Mining Assumptions for Synthesis

Wenchao Li
UC Berkeley
wenchao@berkeley.edu

Lili Dworkin
Haverford College
ldworkin@haverford.edu

Sanjit A. Seshia
UC Berkeley
sseshia@eecs.berkeley.edu

***Abstract*—Automatic synthesis of a reactive system from its formal specification is appealing but often difficult due to the tedium of writing auxiliary specifications, especially on the environment. In several instances, specifications are found unrealizable as a result of insufficient environmental assumptions. We present an approach to this problem for synthesis from LTL based on specification mining. For a satisfiable but unrealizable specification, a counter-strategy can be computed from the synthesis game as a witness to unrealizability. Our algorithm mines environment assumptions from this counter-strategy as well as user scenarios if they are provided. We argue that our approach is a natural way to discover the designer's intent. We demonstrate the effectiveness of our approach on examples from the domains of digital circuits and robotic controllers.**

I. Introduction

Automatic synthesis of a reactive system from high-level specifications is a very attractive proposition. A user describes the target system using a high-level language, such as linear temporal logic (LTL), and the tool will produce a system that automatically satisifies the specification [20], [27]. Such a synthesis tool can enable the human user to specify core design properties while freeing her of the tedious low-level programming tasks. However, synthesis is challenging on multiple dimensions: (i) it requires good-quality specifications on the system; (ii) it requires good models (specifications) of the environment, and (iii) the underlying decision problems often have very high complexity. Industrial experience indicates that these specifications are hard to get right [14]. In this paper, we address the problem of inadequate environment specifications for synthesis from LTL.

A specification is *unsatisfiable* if there does not exist any input and output sequence that can accomplish the objective of the specification. On the other hand, a specification is *unrealizable* if there is no implementation that can produce outputs to satisfy the specification given all possible inputs that can be generated by the environment. Unsatisfiability is a stronger notion of a specification being unsynthesizable than unrealizabilty. The problem that we are tackling in this paper is the case when a specification is *satisfiable but unrealizable*.

This class of specifications has been studied in previous work. For example, the requirements analysis tool RATSY [7] is an interactive tool that can analyze unrealizability for Generalized Reactivity(1) specifications (GR(1)) [26]. The tool uses a game-theoretic approach by taking on the role of the environment while the user takes on the role of the system. The game proceeds by the tool providing inputs to the system and the user determining the outputs as an attempt to satisfy the specification. The move of the tool is a result of the finite-state *counter-strategy* computed for the environment so as to prevent the system from satisfying the specification.

This paper builds upon the above work, using the counter-strategy to further suggest solutions to make the specification realizable. In particular, we use a *template-based specification mining* approach to find LTL properties that are satisfied by the counter-strategy. By asserting the negation of these mined properties as an assumption of the original specification, we effectively rule out such moves by the environment. We iterate this process until either we cannot find a specification in our library of templates that is satisfied by the counter-strategy or the resulting specification becomes realizable.

Our work can be viewed as a debugging approach to unrealizability. Unrealizability typically arises either from an under-constrained environment or from an over-constrained system. We focus on debugging environmental assumptions rather than system guarantees; note however, that these two are complementary, and the proposed approach can easily be adapted to generate guarantees as well. Könighofer et al. [23] identify guarantees that can be weakened or signals that can be less restricted in order to obtain a realizable specification. Our focus is on generating *additional environment assumptions*, since it is the more tricky problem in practice — it is often easier to miss an environment behavior than to miss a system specification because the former is not inherently part of the design and seldom formally defined. Particularly, we want additional assumptions that are tailored to the type of synthesis (GR(1) in this case here) and the existing specification. As also noted in [23], the Boolean formula `false` is also a valid assumption to resolve unrealizability but a trivial and uninteresting one. We propose a template-based specification mining approach to address this problem. By imposing a particular structure on the form of specifications (using templates), we reduce the possibility of generating uninteresting assumptions.

Figure 1 shows the main flow of our method. Our specification mining algorithm takes in as input a library of specification templates, a set of user scenarios (desired I/O traces of the target implementation), and a counter-strategy state machine generated from unrealizability analysis, and produces as output a candidate assumption that can be added to the existing specification to make it realizable.

One may argue in favor of an alternate approach where the additional environment assumption is constructed directly from the counter-strategy so that the resulting specification is realizable. Chatterjee et al. [12] showed that in fact one can

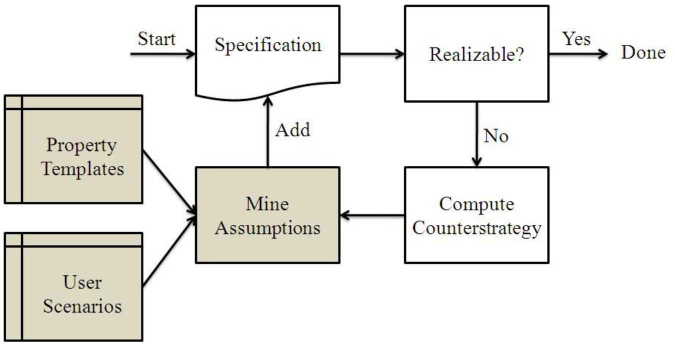

Fig. 1. Flow of Assumption Mining

construct such an assumption by analyzing the game graph that is used to answer the realizability question. In fact, the assumption synthesized (as a Büchi automaton) is minimal in terms of the number of safety and fair edges manipulated in the game graph during synthesis. However, such a monolithic environment assumption (see, e.g., Fig. 3 in [12]) can be difficult for a human user to understand even for correcting a simple specification. Moreover, the Büchi automaton synthesized does not directly translate to a LTL description. In addition, the behavior of the weakest assumption does not neccessarily coincide with the designer's intent. We offer an alternative approach that is simpler from the theoretical viewpoint but very useful in practice.

We make the following contributions in this paper.

- When LTL specifications are satisfiable but unrealizable, we present a novel *counter-strategy guided synthesis* approach based on specification mining that can produce additional environmental assumptions to make the specification realizable.
- We demonstrate the effectiveness of our approach with examples in digital circuits and robotic controllers.

The rest of the paper is organized as follows. Section II surveys related work. We present basic terminology and background information in Section III. The proposed technique is described in Section IV. Experimental results are given in Section V and we conclude in Section VI.

II. Related Work

Unrealizability can come from overconstrained system assertions or insufficient environment assumptions. Our work assumes that unrealizability is due to insufficient environment assumptions, and tackles this by generating additional assumptions.

Cimatti et al. [13] formally define the notion of (minimal) explanation for unrealizability using an unrealizability core, which is the set of specifications responsible for unrealizabilty. In particular, they suggest using the removal of guarantees as a way to explain and fix unrealizability. Our approach can be viewed as orthogonal to theirs. We aim to add environment assumptions to make the specification realizable. We argue that this is also a natural way to fix unrealizability. In fact, (in)formal descriptions of the environment are often not available or far less accessible than those of the system. The challenge here is finding the right assumptions to add.

Counter-strategies have been used to explain the failure in synthesizing a system that respects the specification, such as in the context of Live Sequence Charts [9]. Könighofer et al. [22] provide an explanation for LTL unrealizability by computing a finite-state counter-strategy for the environment. The counter-strategy is further simplified by removing specifications and variables that are not responsible for unrealizability. A heuristic is also provided for computing from the counter-strategy a *counter-trace* — a fixed infinite input sequence that, regardless of what the system outputs, will still ensure that the system specification will be violated. The paper finally illustrates how to compute this counter-strategy specifically for GR(1) specifications. Our work builds upon this work, mining specifications from these counter-strategies and counter-traces. Similar to this work, we also focus our attention on GR(1) specifications.

Könighofer et al. [23] follow up on their previous work with a model-based diagnosis technique that identifies components (specification or variables) in the system guarantees which are overconstrained. For GR(1) specifications, the authors also show how to compute the realizable and unrealizable core quickly using approximations. In our work, we focus on analyzing the weakness in the environment assumptions instead of the constraints in the system guarantees. Moreover, we produce additional assumptions instead of localizing the error.

The problem of correcting the assumption of an unrealizable LTL specification has also been studied in depth in [12]. The authors construct an additional assumption that constrains only the environment as weakly as possible, and makes the resulting specification realizable. The approach proceeds by first computing a safety assumption that removes a minimal set of environment edges from the game graph, and then computing a liveness assumption that puts fairness on the remaining environment edges. Finding a minimal set of fair edges was shown to be NP-hard. The authors use probabilistic games to compute a locally minimal fairness assumption and implemented their approach in the tool GIST [11]. Their work can synthesize general environment assumptions (as an intersection of safety and liveness assumptions [4]) for any LTL synthesis problem. Our work provides a simpler but practical approach by restricting the form of missing assumptions and uses specification mining to identify a set of assumptions such that it restricts the environment in a reasonable way to make the specification realizable.

Previous work by Hagihara et al. [18] has also attempted to extract environment constraints of simple forms to make specifications strong satisfiable, where strong satisfiability means that for all input sequences which are given in advance, there exists an output sequence such that the specification is satisfied. Their method is based on deriving these constraints from Büchi automata representing the specifications. Our method is different because we tackle realizability directly instead of strong satisfiability. In addition, we use a counter-strategy guided approach instead of the constructive derivation used in this previous work.

The realizability problem for general LTL formulas has been shown to be 2EXPTIME-complete [28]. This high complexity of LTL synthesis has prohibited its wide-scale adoption in actual practice. However, more recently, a subclass of LTL known as Generalized Reactivity(1) formulas (GR(1)) has been shown to be very amenable to synthesis. Piterman et al. [26] show that the realizability and synthesis problem of GR(1) specifications can be solved efficiently in polynomial time. Building on this result, real world problems from several application domains have been approached using synthesis from GR(1) specifications, such as planning for autonomous vehicles in an urban environment [31]. Leveraging the expressiveness of the GR(1) class of specifications and its efficient synthesis algorithm, our paper focuses on mining assumptions in GR(1). However, our template-based approach can also be extended to handle more complicated specifications.

Our approach is motivated by the extensive work done in the area of specification mining. The study of automatically generating specifications goes back as early as 1974 [30]. Since then, many techniques have been proposed and they seek to either mine specifications dynamically from execution traces [16] or infer them directly from the program through static analysis [5]. DAIKON [16] is one of the earliest template-based specification mining tools that generates single-state invariants or pre-/post-conditions in programs. More recently, Li et al. [24] proposed a scalable technique for mining temporal specifications from traces produced by digital systems, and showed that the mined specifications are effective in localizing bugs in designs. Our approach is similar in spirit but presents a novel application of specification mining - the generation of environment assumptions for LTL synthesis.

III. Preliminaries

A. LTL Synthesis

1) Linear Temporal Logic: Given a finite set of propositional (Boolean) variables P, LTL formulas are constructed as follows.

$$\psi ::= p \,|\, \neg\psi \,|\, \psi \vee \psi \,|\, \mathbf{X}\,\psi \,|\, \psi \mathbf{U}\,\psi$$

where $p \in P$ is a propositional variable, $\mathbf{X}$ is the temporal operator *next* and $\mathbf{U}$ is the temporal operator *until*. Other temporal operators can be derived using these two temporal operators and Boolean operators. $\mathbf{F}\,\psi = \texttt{true}\,\mathbf{U}\,\psi$. $\mathbf{G}\,\psi = \neg\mathbf{F}\,\neg\psi$. We interpret LTL formulas over infinite words $w \in (2^P)^\omega$. Then the language of a LTL formula ψ is the set of infinite words that satisfy ψ, given by $\mathcal{L}(\psi) = \{w \in (2^P)^\omega \,|\, w \models \psi\}$. One classic example is the LTL formula $\mathbf{G}\,(p \rightarrow \mathbf{F}\,q)$, which means every p must be followed by some q in the future.

2) Finite-state Transducers: Given a set of atomic propositions P, we partition P into two disjoint sets I and O that represent the set of input signals and the set of output signals respectively. A *Moore* transducer is a tuple $M = (\mathcal{I}, \mathcal{O}, S, s_0, \delta, \theta)$, where $\mathcal{I} = 2^I$ is the input alphabet, $\mathcal{O} = 2^O$ is the output alphabet, S is the set of states, $s_0 \in S$ is the initial state, $\delta : S \times \mathcal{I} \rightarrow S$ is the transition function, and $\theta : S \rightarrow \mathcal{O}$ is the state output function. A *Mealy* transducer is similar, except that the state output funciton is $\theta : S \times \mathcal{I} \rightarrow \mathcal{O}$. Given a word $w \in \mathcal{I}^\omega$, a run of M is the infinite sequence $\pi \in S^\omega$ of states such that $\pi_0 = s_0$, and $\pi_{i+1} = \delta(\pi_i, w_i)$ for all $i \geq 0$. The run π on w produces an infinite word $M(w) \in 2^P$ such that $M(w)_i = \theta(\pi_i) \cup w_i$ for all $i \geq 0$. The language of M is then the set $\mathcal{L}(M) = \{M(w) \,|\, w \in \mathcal{I}^\omega\}$.

3) Satisfiability and Realizability: A LTL formula ψ is *satisfiable* if there exists an infinite word that satisfies ψ, i.e. $\exists w \in (2^P)^\omega$ such that $w \models \psi$. A Moore transducer M *satisfies* a LTL formula ψ if $\mathcal{L}(M) \subseteq \mathcal{L}(\psi)$. We write this as $M \models \psi$. Then *realizability* is the the problem of checking whether there exists a Moore transducer M that satisfies the LTL specification ψ.

B. Synthesis as a Game

A *two-player deterministic game graph* is a tuple $\mathcal{G} = (Q, Q_0, E)$ where $Q = Q_1 \cup Q_2$ can be split into two disjoint sets of states. Q_1 is the set of *player 1* states, and Q_2 is the set of *player 2* states. Q_0 is the set of initial states. $E = Q \times Q$ is the set of edges. The game is played by each player taking turn to decide the successor state. If we are currently at a state in Q_1, player 1 decides the successor state. Otherwise, player 2 decides the successor state.

A *play* of the game graph $\mathcal{G}$ is an infinite sequence of states $\pi = s_0 s_1 \ldots$ such that $(q_i, q_{i+1}) \in E$ for all i. We write Π as the set of plays. A strategy for player 1 is the function $\alpha : Q^* \times Q_1 \rightarrow Q$ that decides the successor state given a finite sequence of states ending at a player 1 state. Similary, we can define a strategy for player 2 using the function $\beta : Q^* \times Q_2 \rightarrow Q$. A strategy is *memoryless* if it depends only on the current state of the play. For example, a memoryless strategy for play-1 is $\alpha : Q_1 \rightarrow Q$. Given two strategies α and β, and a state $q \in Q$, the *outcome* is the play starting at q and executed by α and β, denoted as $o(q, \alpha, \beta)$.

We consider *parity* games in the context of LTL synthesis. Let $\texttt{Inf}(\pi) = \{q \in Q \,|\, \forall_i, \exists_j \quad j > i \quad \wedge \quad q = q_j\}$ be the set of states visited infinitely many times in π. A parity game consists of a game graph $\mathcal{G}$ and a parity objective. Given a function $p : Q \rightarrow \{0, 1, \ldots, m\}$ that maps every state to a priority, a parity objective is the set of plays $PO(p) = \{\pi \in \Pi \,|\, \texttt{min}\{p(\texttt{Inf}(\pi))\}$ is even $\}$ requiring that the least priority is even among the sets of states that are visited infinitely oftenn.

Given a parity game, we say a strategy α is winning for player 1 for some state q if for every strategy β of player 2, we have $o(q, \alpha, \beta) \in PO(p)$. Memoryless strategies exist for both players for parity game [15].

Given a LTL formula ψ, and a partition of the set of propositions P into I and O, *synthesis* is the problem of finding a Moore transducer $M = (\mathcal{I}, \mathcal{O}, S, s_0, \delta, \theta)$ such that $M \models \psi$. This problem can be solved by first constructing a nondeterministic Büchi automaton that accepts $\mathcal{L}(\psi)$ [29]. This automaton is then translated into a deterministic parity automaton that accepts $\mathcal{L}(\psi)$ [25]. By splitting the states in the parity automaton according to the inputs I and outputs O, we can obtain a parity game. We use the function $\kappa_1 : Q_1 \rightarrow \mathcal{I}$ and $\kappa_2 : Q_2 \rightarrow \mathcal{O}$ to label states in the game graph. Every memoryless winning strategy for player 1 (the system) in this game can be represented by a Moore transducer $M = (\mathcal{I}, \mathcal{O}, S, s_0, \delta, \theta)$ satisfying ψ such that $S = Q_1$, $s_o = q_I$ where q_I is the initial state, $\delta(s, i) = s'$ where $\kappa_1(s') = i$, and $\theta(s) = \kappa_2(\alpha(s))$ where $(\alpha(s), s') \in E$.

1) Counter-strategies: A *counter-strategy* to the synthesis problem is a strategy for the environment (player 2) such that it can force the specification to be violated. Hence, we can construct a Mealy transducer M_c corresponding to this counter-strategy for player 2, which satisfies $\neg\psi$. Alternatively, we can treat the synthesis problem as finding a Mealy transducer that satisfies the LTL specification ψ and the counter-strategy for player 2 would be a Moore machine.

C. Synthesis of GR(1) specifications

Consider a LTL specification ψ that can be expressed in this form, $\psi = \psi_e \rightarrow \psi_s$. ψ_e is the environment assumption and ψ_s is the system guarantee. We require ψ_l for $l \in \{e, s\}$ to be a conjunction of sub-formulas in the following forms:

- ψ_l^i: a Boolean formula that characterizes the initial states
- ψ_l^t: a LTL formula that characterizes the transition, in the form $\mathbf{G}\ B$, where B is a Boolean combination of variables in $I \cup O$ and expression $\mathbf{X}\ u$ where $u \in I$ if $l = e$ and $u \in I \cup O$ if $l = s$.
- ψ_l^f: a LTL formula that characterizes fairness, in the form $\mathbf{G}\ \mathbf{F}\ B$, where B is a Boolean formula

The GR(1) synthesis problem can also be reduced to solving a two-player deterministic game between the system and the environment. Piterman et al. [26] describe a fixpoint formulation that solves the realizability problem in cubic time. We omit details of the algorithm in this paper. The form of GR(1) specifications allows us to impose structure on the assumptions that we seek to mine. Starting with a set of GR(1) specifications, the goal is to see if we can find assumptions that are also in the GR(1) class to make the resulting specification realizable.

IV. Specification Mining

Our method first computes a finite-state counter-strategy using the approach proposed in [22]. The counter-strategy state-machine summarizes the next moves of the environment in response to the current output of the system, which will force a violation of the specification. We then use a template-based mining approach to find specifications that are satisfied by the counter-strategy. By asserting the negation of such a specification as an assumption to the original specification, we effectively rule out such moves by the environment. We iterate this process until either we cannot find a specification in our library of templates that is satisfied by the counter-strategy or the resulting specification becomes realizable.

A. Overall Algorithm

We give an overview of our mining algorithm in this section. There are four main procedures used by our approach.

Algorithm 1 Mine Assumptions for $\psi = \psi_e \rightarrow \psi_s$

```
Input: $B = I \cup O$: set of Boolean signals
Input: $\psi$: initial specification
Input: $\Gamma$: set of specification templates
Input: $\Omega_u$: set of scenario traces for the target system
Output: $\phi$: additional assumption required for realizability
$\Gamma(B)$ := GenerateCandidates($\Gamma$, $B$)
while $\neg$Realizable($\psi$) do
    $M_c$ := CounterStrategy($\psi$)
    $\rho$ := Mine($M_c$, $\Gamma(B)$, $\Omega_u$, $\psi_e$)
    if $\rho$ = false then
        return Insufficient Template
        Quit
    end if
    $\psi_e := \psi_e \wedge \rho$
    $\psi := \psi_e \rightarrow \psi_s$
end while
```

GenerateCandidates(Γ, B) generates a set of template instantiations $\Gamma(B)$ in a particular order as candidates of the additional environment assumptions.

Realizable(ψ) checks if the specification ψ is realizable [20], [7].

CounterStrategy(ψ) returns a counter-strategy as a Moore machine M_c for the environment to force a violation of the specification if ψ is not realizable [7].

Mine(M_c, $\Gamma(B)$, Ω_u, ψ_e), which is our contribution, returns a formula ρ as an additional assumption. We give the details of this procedure in Section IV-B.

It should be noted that if we are able to mine a set of assumptions that are sufficient to make the specification realizable, we still want to remove redundant assumptions from this set. We point the readers to [13] for finding a set of assumptions that is minimally sufficient.

Example 1: Consider the following simple example, with $I = \{r, c\}$ and $O = \{g, v\}$. We start with no assumptions, i.e. $\psi_e = \texttt{true}$, and no user traces. The system guarantee ψ_s is a conjunction of the following properties.

- **G1**: $\mathbf{G}\ (r = 1 \rightarrow \mathbf{X}\ (\mathbf{F}\ g = 1))$
- **G2**: $\mathbf{G}\ ((c = 1 \vee g = 1) \rightarrow \mathbf{X}\ g = 0)$
- **G3**: $\mathbf{G}\ (c = 1 \rightarrow v = 0)$
- **G4**: $\mathbf{G}\ \mathbf{F}\ (g = 1 \wedge v = 1)$

Specifications **G1** and **G2** are borrowed from the first example in [12]. **G1** says every request r has to be granted eventually by setting g to high starting from the next time step. **G2** says if either the cancel signal c is high or grant g is high, then g has to be low in the next time step. **G3** says if c is high, then the valid signal v has to be low. **G4** says that g and v are both true (a valid grant) infinitely often. Our algorithm produces assumptions that achieves realizability in the following order.

- **A1**: $\mathbf{G}\ (\mathbf{F}\ c = 0)$
- **A2**: $\mathbf{G}\ (r = 0 \rightarrow \mathbf{X}\ c = 0)$
- **A3**: $\mathbf{G}\ (r = 0 \rightarrow c = 0)$

- **A4**: $\mathbf{G}\ (\mathbf{F}\ r = 0)$

A1 requires that the cancel signal c to be low infinitely often. This is necessary because otherwise setting c permanently to high can force the grant signal to stay low because of **G2**, and this is in conflict with **G1**.

A2 requires that cancel signal c to be low the cycle after the request signal r is set to low. This is also necessary because the environment can generate a sequence of inputs in which r and c are set to high in an alternating fashion. According to **G2** and **G3**, at least one of v and g is low at every cycle. This is in conflict with **G4**.

A3 requires that the cancel signal c to be low in the same cycle as the request signal r is set to low. This assumption together with **A2** prevents the scenario of alternating r and c described above. However, the resulting specification is still unrealizable. This is because the environment can force r to stay high from some point onwards and set c in an alternating fashion such that **G4** is again violated.

A4 requires that the request signal r is set low infinitely often. Adding this assumption resolves the above problem and the specification is finally realizable.

This example illustrates an instance of which our specification mining approach iteratively adds assumptions to the original specification. At each iteration, the added assumption removes certain behaviors of the environment which are causes of unrealizability. This set, however, is in no way the only set or the minimal set of assumptions that can achieve realizabiliy. In fact, we can replace **A1** with $\mathbf{G}\ \mathbf{F}\ (c = 0 \vee r = 0)$ and the resulting specification is still realizable. However, each element in this set of assumptions is a simple specification in GR(1) that contains environment behaviors which are necessary for the realizability of the specification and can be readily analyzed by the user. If a particular assumption is deemed too strong or erroneous, the user can simply remove that assumption and our method will seek alternative replacements until either the resulting specification is realizable or the set of candidate assumptions is depleted.

Our approach can be viewed as a recommendation system for the user, which can incorporate the user's design intent in an interactive way. The user can engineer the templates based on his knowledge of the environment (possibly another system) or inspect and rule out any candidate assumption as the mining algorithm proceeds. To come up with multiple recommendations, one can simply restart the algorithm with a reduced set of template instantiations (in which the first additional assumption found in constructing the previous ϕ is discarded) and try to find another set that achieves realizability.

B. Template-based Mining

In this section, we describe the procedure **Mine** in detail. First, we formally define the set of templates Γ.

Definition 1: **[Templates]** Given a set of Boolean variables B, a template γ is a LTL formula with at least one placeholder $?_b$, such that each $?_b$ can be replaced by a literal l_b of Boolean variable $b \in B$ and replacing all the $?_b$(s) results in a valid LTL formula. We call this an *instantiation* of γ with the l_b(s). We write $\gamma(B)$ as the set of all unique instantiations of γ using variables in B.

1) Templates in GR(1): In our experiments, we focus on mining assumptions that are in GR(1). The set Γ is constructed using the following types of templates.

- $\gamma^1 : \mathbf{G}\ \mathbf{F}\ ?_b$, where $b \in I$
- $\gamma^2 : \mathbf{G}\ (?_{b1} \rightarrow \mathbf{X}\ ?_{b2})$, where $b_1 \in I \cup O$ and $b_2 \in I$
- $\gamma^3 : \mathbf{G}\ (?_{b1} \vee ?_{b2})$, where $b_1, b_2 \in I$

The negation of each template has the following form. We omit the negation on the placeholder for the simplicity of notations.

- $\phi^1 : \mathbf{F}\ \mathbf{G}\ ?_b$
- $\phi^2 : \mathbf{F}\ (?_{b1} \wedge \mathbf{X}\ ?_{b2})$
- $\phi^3 : \mathbf{F}\ (?_{b1} \wedge ?_{b2})$

Also, notice that we do not mine assumptions that characterize the initial state of the system. This is just because these assumptions are easier to write in practice, and, moreover, it is straightforward to discover them from the counter-strategy.

2) Generate Candidate Assumptions from Templates: We generate a set of candidate assumptions in procedure **GenerateCandidates** of Algorithm 1 by first generating a set of candidates $\gamma^i(B)$ for each of the three templates. Each set contains all unique instantiations of the corresponding template and boolean signals. For example, if $B = \{a\}$, then $\gamma^1(B) = \{\mathbf{G}\ \mathbf{F}\ a, \mathbf{G}\ \mathbf{F}\ \neg a\}$. Next, we construct $\Gamma(B) = \{\gamma^1(B), \gamma^2(B), \gamma^3(B)\}$.

3) Mining assumptions: Algorithm 2 describes the series of tests for checking whether a candidate assumption should be added to the current specification ψ.

Algorithm 2 Procedure **Mine**

```
Input: $M_c$: counter-strategy state machine
Input: $\Gamma(B)$: set of candidate assumptions
Input: $\Omega_u$: set of scenario traces for the target system
Input: $\psi_e$: current assumption
Output: $\rho$: additional assumption
while $\Gamma(B) \neq \emptyset$ do
    $\rho_c$ = pop($\Gamma(B)$)
    if $\Omega_u \models \rho_c$ then
        if $M_c \models \neg\rho_c$ then
            if Satisfiable($\rho_c \wedge \psi_e$) then
                return $\rho_c$
            end if
        end if
    end if
end while
return false
```

The procedure **pop** will remove and return a candidate assumption from $\Gamma(B)$ based on the last returned candidate by **Mine**. The idea is to keep trying candidate assumptions from a particular template until one is successfully added to the existing specification and then move on to a different template. We choose the following order for selecting the templates, $\gamma^1, \gamma^2, \gamma^3, \gamma^1, \gamma^2, \ldots$ and corresponding

$\gamma^1(B), \gamma^2(B), \gamma^3(B), \gamma^1(B), \gamma^2(B), \ldots$ for the sets of candidates. If a current set is depleted, then we move on to the next set in the aforementioned order. The order for which a candidate assumption is popped from a set $\gamma^i(B)$ is arbitrary. The assumptions generated in Example 1 are based on this template order. We choose this order to select candidate assumptions of different forms based on our experience but we do not claim that it is general for all problems.

Eliminate False Assumptions

Given a candidate assumption ρ_c, we check if it matches the user's intent by checking whether ω_u satisfies ρ_c, for all $\omega_u \in \Omega_u$, where Ω_u is the set of traces representing some known behaviors of the target design and the environment. Each ω_u can be finite or infinite. For the different types of templates, if ω_u is finite, we only check the validity of instantiations of γ^1 and γ^3 in ω_u up to the last time step and instantiations of γ^2 in ω_u up to the second last time step.

Eliminate Counter-Strategy

Our method follows a *counter-strategy guided synthesis* approach in which at every iteration, we constrain the environment more by adding an additional assumption and move closer to realizability. The key idea is to eliminate the counter-strategy generated in unrealizabilty analysis by first finding a formula that is satisfied by the counter-strategy state machine, and then assert the negation of the formula as an additional assumption. This step is equivalent to model checking the counter-strategy state machine with the candidate specification. In the cases where RATSY produces a *countertrace* τ_c, we use τ_c in place of M_c by encoding it as a Moore machine.

Notice that our template-based mining approach has a lot in common with query checking. Query checking was first proposed by Chan [10]. A *temporal logic query* is a temporal logic formula containing a special symbol $?_x$, called the *placeholder*, which can be replaced by any propositional formula. The job of query checking is to find the strongest solutions to $?_x$ such that the resulting expression is true for a given model. For example, the query $\mathbf{A}\,\mathbf{G}\,?_x$ is to find the strongest propositional formula that is true at all points of computation of the model. One can also restrict the query to propositions of interest. In the previous example, one may be only interested in formulas that contain only atomic propositions p and q. We write this modified query as $\mathbf{A}\ \mathbf{G}\ ?_x\{p, q\}$. Given the set of atomic propositions of interest of size n, the query checking problem can be solved by searching all 2^{2^n} propositional formulas (modulo logic equivalence) as replacements for the placeholders, and individually verifying each resulting temporal logic formula. Gurfinkel et al. [17] implemented one such temporal logic query checker for CTL based on multi-valued model checking. More recently, Chockler et al. [19] studied its LTL variants.

In our approach, we try to avoid the potential high complexity of query checking by further imposing structure on the Boolean formula. However, query checking has the advantage of finding the strongest formula for a particular query that can be satisfied by the system. As a result, by asserting the negation of the resulting match as an additional assumption, we are less likely to rule out desirable behaviors of the environment. We plan to leverage this technique in the future.

Avoid Trivial Assumptions

To avoid synthesizing trivial systems as a result of the additional assumptions, each candidate match is checked with the current assumption ψ_e for inconsistency. If the conjunction of the match and ψ_e is unsatisfiable, we eliminate this candidate. Moreover, each candidate match can be examined by the user to check if it makes sense. In our experiments, no user inspection is sought; instead, we simply return the assumptions mined by the procedure.

V. Experimental Results

In this section, we present case studies from the domains of digital circuit design and robotic controller synthesis. We use RATSY [7] to generate the counter-strategy in case of unrealizability. Additionaly, we let RATSY perform minimization of the unrealizable core. Hence, the counter-strategy we use to mine assumptions is computed from the reduced unrealizable core. We use the Cadence SMV model checker [1] to (1) check if the current set of assumptions is satisfiable, (2) generate witnesses (as representations of user scenarios) to the original specification and (3) model check counter-strategy state machines with candidate assumptions. Our experiment proceeds as follows. Starting with a known realizable specification ψ (with a minimally sufficient set of assumptions [13]),

1) Remove an arbitrary assumption τ from ψ to get ψ';
2) Proceed with Algorithm 1 to generate a replacement assumption ϕ;
3) Evaluate the relationship between ϕ and τ;
4) Restart from Step 1 by removing a different assumption.

A. AMBA AHB Bus Protocol

ARM's Advanced Microcontroller Bus Architecture (AMBA) Advanced High Performance Bus (AHB) is an on-chip communication protocol. The specification allows for up to 16 masters and 16 slaves. The masters initiate communication (reading or writing) with the slaves, and the slaves respond to a master's request. There is an address bus and a data bus, each of which can only be accessed by one master at a time. An arbiter controls access to the address bus. A bus access can either be a single transfer, or a burst, which is a transaction consisting of multiple transfers. A bus access can also either be locked (incapable of being interrupted) or unlocked. Our specifications for this protocol are taken from the example files provided with RATSY [7]. For details of this protocol, we refer the readers to [6]. There are four environment signals. The first three are driven by the masters and the last one is driven by the slaves.

- `HBUSREQ[i]` - master `i` requests access to the bus
- `HLOCK[i]` - master `i` requests locked access to the bus (used in combination with `HUBUSREQ[i]`)
- `HBURST[1:0]` - one of following: single transfer (SINGLE), 4-transfer burst (BURST4), or unspecified size burst (INCR) Signals driven by the slaves:

- HREADY - high if the slave has finished working with the current data; needs to be high before bus ownership can change or transfers can begin

Our experiment was on the configuration of the protocol with 1 master and 2 slaves (“amba02.rat”). We ignored the assumptions that characterize the initial states. In fact, the removal of any of them does not lead to unrealizability. First, we remove the following environment fairness.

$$\psi_e^f = \mathbf{G}\,(\mathbf{F}\,\texttt{HREADY} = 1)$$

We used a single satisfying trace (witness) of the original specification as a representation of the user's knowledge of the design. The trace is shown below, with parentheses indicating the start and end of the loop and input signals in bold.

cycle:	(1	2	3)
HBUSREQ[0]:	0	0	0
HBUSREQ[1]:	0	0	1
HBURST[0]:	0	0	0
HBURST[1]:	0	0	0
HLOCK[0]:	0	0	0
HLOCK[1]:	0	0	0
HREADY:	0	1	1
HMASTER[0]:	0	0	0
HGRANT[0]:	1	1	1
HGRANT[1]:	0	0	0
HMASTLOCK:	0	0	0
LOCKED:	0	0	0
BUSREQ:	0	0	0
DECIDE:	1	0	0
START:	1	0	0

TABLE I
A SINGLE USER SCENARIO FOR THE AMBA AHB BUS EXAMPLE

We applied our algorithm on the remaining specification and we found ψ_e^f in exactly one iteration. In fact, we can choose a different order to find candidate assumptions. The following is another possible valid replacement.

$$\mathbf{G}\,(\texttt{HREADY} = 0 \rightarrow \mathbf{X}\,\texttt{HBUSREQ[0]} = 0) \wedge$$
$$\mathbf{G}\,(\mathbf{F}\,\texttt{HREADY} = 1)$$

In this example, the second candidate assumption $\mathbf{G}\,(\texttt{HREADY} = 0 \rightarrow \texttt{HBUSREQ[0]} = 1)$ was refuted by the witness. After removing redundant assumptions, we again get back ψ_e^f.

Next, we removed the following environment transition.

$$\psi_e^t : \mathbf{G}\,(\texttt{HLOCK[0]} = 1 \rightarrow \texttt{HBUSREQ[0]} = 1)$$

Following our algorithm, we obtained $\mathbf{G}\,(\mathbf{F}\,\texttt{HLOCK[0]} = 0)$ as a replacement. The original assumption says whenever master 0 sets HLOCK[0] to high, it should be sending a request at the same time by setting HBUSREQ[0] to high. Our replacement says master 0 should de-assert HLOCK[0] infinitely often. This assumption may not be completely desirable because it does not pinpoint the condition on which the signal should be de-asserted. However, it clearly indicates the need to prevent a master from having a locked access to the bus permanently, and the fact that adding this requirement will make the specification realizable.

In general, we may get a number of possible replacements. It is debatable which of these replacements best represents the designer's intent. We are simply offering a recommendation system in which the user can choose from a pool of possible assumptions. The quality of this pool is better if the user can provide more information to the synthesis process in the form of desired traces of the target design. Our experiments show that even with limited user input such as a single desired trace in this case here, our method is able to generate good quality assumptions that achieve realizability.

B. Generalized Buffer

IBM's generalized buffer (GenBuf) transmits data from n senders to two receivers. The senders provide data in any order, but the receivers must receive the data in turns. The handshake protocol between sender, buffer, and receiver is as follows. The senders request permission from the buffer to send their data. The buffer must acknowledge each sender's request. The buffer then sends a request to a receiver for permission to transmit the data. The receiver must also acknowledge the buffer's request. We used the specifications provided with the tool ANZU [21]. For details of this example, we refer the readers to [8]. It has the following environment signals.

- StoB_REQi - sender i requests a send from the buffer
- RtoB_ACKj - receiver j acknowledges the buffer's request
- FULL - the FIFO is ready to send data
- EMPTY - FIFO is ready to receive data

We performed a similar experiment as the one in Section V-A. For the complete list of results of this benchmark, we refer the readers to [2]. We used two witnesses of the original specification for refuting false candidate assumptions. In the first experiment, we removed the following fairness assumption.

$$\psi_e^f : \mathbf{G}\,(\mathbf{F}\,(\texttt{BtoR\_REQ0} = 1 \leftrightarrow \texttt{RtoB\_ACK0} = 1))$$

Our algorithm produced the following replacement.

$$\mathbf{G}\,(\mathbf{F}\,\texttt{RtoB\_ACK0} = 1)$$

The first candidate $\mathbf{G}\,(\mathbf{F}\,\texttt{StoB\_REQ0=0})$ was refuted by the witnesses. In this example, we were not able to recover the missing assumption. Our replacement represents an environment where the receiver acknowledges infinitely often. The original assumption states that only true request (when BtoR_REQ0 is high) is acknowledged infinitely often. The replacement assumption is stronger than the original, but still provides a hint to the desired behavior of the environment.

C. Robotic Vehicle Controller

Our example of a robotic vehicle controller is motivated by the work done by Wongpiromsarn et al. [31]. Their work aims to synthesize a discrete planner for an autonomous vehicles to navigate in an urban environment while following traffic rules and avoiding obstacles. We use the following simplified variant in our experiment. Given a rectangular grid of length X (length of the road) and width Y (number of lanes), define the coordinates of where the vehicle is located as $l_{x,y}$. We use $o_{x,y}$ to denote if there is an obstacle at position (x, y) at every time step ($o_{x,y} = 1$ if there is one). This is used to simulate moving obstacles such as other cars in the urban environment. The requirements are the following.

- **A**: All squares are clear of obstacles infinitely often: $\mathbf{G}\,\mathbf{F}\; o_{x,y} = 0$.
- **G1**: The car is at initial position $l^i_{x,y}$ and there is no obstacle at the initial position.
- **G2**: The vehicle can move to an adjacent square or stay in the current square at each time step.
- **G3**: The vehicle cannot move into a square occupied by an obstacle.
- **G4**: The vehicle eventually reaches its final destination.

We expressed these requirements in GR(1) formulas [3]. One way to make the specification unrealizable is to have the destination square permanently blocked (by removing the corresponding fairness assumption). We followed Algorithm 1 and were able to recover the assumption in one iteration.

VI. Conclusion

We have proposed a template-based approach that can mine a set of assumptions to correct an unrealizable specification for LTL synthesis. Our approach uses both the counter-strategy which can be computed as a result of unrealizability and exemplary traces from the user for the target design to generate candidate assumptions. The assumptions produced are of simple forms and can be readily understood and analyzed by an user. Our experiments indicate that our proposed method is practical. For a number of examples taken from existing literature, we show that we can either reverse-engineer the missing assumption or produce reasonable replacements. In the future, we would like to incorporate query checking into the mining algorithm and evaluate its effectiveness.

References

[1] Cadence smv. http://www.kenmcmil.com/smv.html.

[2] http://www.eecs.berkeley.edu/∼wenchaol/files/genbuf.result.

[3] http://www.eecs.berkeley.edu/∼wenchaol/files/robotic.rat.

[4] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987. 10.1007/BF01782772.

[5] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 98–109, New York, USA, 2005. ACM.

[6] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6, April 2007.

[7] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Knighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. Ratsy - a new requirements analysis tool with synthesis. In *CAV'10*, pages 425–429, 2010.

[8] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from psl. *Electron. Notes Theor. Comput. Sci.*, 190:3–16, November 2007.

[9] Yves Bontemps, Pierre-Yves Schobbens, and Christof Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundam. Inf.*, 62:139–169, February 2004.

[10] William Chan. Temporal-logic queries. In E. Emerson and A. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 450–463. Springer Berlin / Heidelberg, 2000.

[11] Krishnendu Chatterjee, Thomas Henzinger, Barbara Jobstmann, and Arjun Radhakrishna. A solver for probabilistic games. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 665–669. Springer Berlin / Heidelberg, 2010.

[12] Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Environment assumptions for synthesis. In *Proceedings of the 19th international conference on Concurrency Theory*, CONCUR '08, pages 147–161, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In *Proceedings of the 9th international conference on Verification, model checking, and abstract interpretation*, VMCAI'08, pages 52–67, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] Fady Copty, Amitai Irron, Osnat Weissberg, Nathan P. Kropp, and Gila Kamhi. Efficient debugging in a formal verification environment. In *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '01, pages 275–292, London, UK, 2001. Springer-Verlag.

[15] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proceedings of the 32nd annual symposium on Foundations of computer science*, SFCS '91, pages 368–377, Washington, DC, USA, 1991. IEEE Computer Society.

[16] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[17] Arie Gurfinkel, Benet Devereux, and Marsha Chechik. Model exploration with temporal logic query checking. In *Proceedings of Sigsoft Conference on Foundations of Software Engineering (FSE'02*, pages 139–148. ACM Press, 2002.

[18] S. Hagihara, Y. Kitamura, M. Shimakawa, and N. Yonezaki. Extracting environmental constraints to make reactive system specifications realizable. In *Software Engineering Conference, 2009. APSEC '09. Asia-Pacific*, pages 61 –68, dec. 2009.

[19] Arie Gurfinkel Hana Chockler and Ofer Strichman. Variants of ltl query checking. Lecture Notes in Computer Science. Springer, 2010. in press.

[20] Barbara Jobstmann and Roderick Bloem. Optimizations for ltl synthesis. In *Proceedings of the Formal Methods in Computer Aided Design*, FMCAD '06, pages 117–124, Washington, DC, USA, 2006. IEEE Computer Society.

[21] Barbara Jobstmann, Stefan Galler, Martin Weiglhofer, and Roderick Bloem. Anzu: a tool for property synthesis. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 258–262, Berlin, Heidelberg, 2007. Springer-Verlag.

[22] R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 152 –159, November 2009.

[23] Robert Könighofer, Georg Hofferek, and Roderick Paul Bloem. Debugging unrealizable specifications with model-based diagnosis. In *Haifa Verification Conference (HVC)*, Lecture Notes in Computer Science. Springer, 2010.

[24] Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. Scalable specification mining for verification and diagnosis. In *DAC '10*, pages 755–760, 2010.

[25] Nir Piterman. From nondeterministic büchi and streett automata to deterministic parity automata. In *21st Symposium on Logic in Computer Science (LICS06)*, pages 255–264. IEEE Computer Society, 2006.

[26] Nir Piterman and Amir Pnueli. Synthesis of reactive(1) designs. In *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI06*, pages 364–380. Springer, 2006.

[27] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.

[28] R. Rosner. Modular synthesis of reactive systems. *Ph.D. dissertation, Weizmann Institute of Science*, 1992.

[29] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.

[30] Ben Wegbreit. The synthesis of loop predicates. *Commun. ACM*, 17(2):102–113, 1974.

[31] T. Wongpiromsarn. Formal methods for design and verification of embedded control systems: application to an autonomous vehicle. 2010.