

Flow-Based Synthesis of Reactive Tests for Discrete Decision-Making Systems with Temporal Logic Specifications

Josefine B. Graebener*, Apurva S. Badithela*, Denizalp Goktas, Wyatt Ubellacker, Eric V. Mazumdar, Aaron D. Ames, Richard M. Murray

Abstract—Designing tests to evaluate if a given autonomous system satisfies complex specifications is challenging due to the complexity of these systems. This work proposes a flow-based approach for reactive test synthesis from temporal logic specifications, enabling the synthesis of test environments consisting of static and reactive obstacles and dynamic test agents. The temporal logic specifications describe desired test behavior, including system requirements as well as a test objective that is not revealed to the system. The synthesized test strategy places restrictions on system actions in reaction to the system state. The tests are minimally restrictive and accomplish the test objective while ensuring realizability of the system’s objective without aiding it (semi-cooperative setting). Automata theory and flow networks are leveraged to formulate a mixed-integer linear program (MILP) to synthesize the test strategy. For a dynamic test agent, the agent strategy is synthesized for a GR(1) specification constructed from the solution of the MILP. If the specification is unrealizable by the dynamics of the test agent, a counterexample-guided approach is used to resolve the MILP until a strategy is found. This flow-based, reactive test synthesis is conducted offline and is agnostic to the system controller. Finally, the resulting test strategy is demonstrated in simulation and experimentally on a pair of quadrupedal robots for a variety of specifications.

Index Terms—Test and Evaluation, Reactive Test Synthesis, Formal Methods, Network Flows, Optimization

I. INTRODUCTION

Safety is imperative for a wide range of autonomous systems, from self-driving vehicles, to autonomous flight and space missions, to assistive robotics, and medical devices. To ensure safety, various challenges need to be addressed [1]. For example, these systems need to be aware of their own state and adapt their behavior in response to the environment, which requires reasoning over both discrete and continuous inputs and states. Deployment of these safety-critical autonomous

systems requires thorough testing, both in simulation and in the operating environment, which is crucial to validating the system’s performance. Typically, test cases are designed to uncover bugs and corner cases in the system design that lead to safety-critical errors. However, for these tests to be successful, executing them requires setting up a test environment that is consistent with the test case while also allowing for correct system implementations. To make this process efficient, it is equally important to automatically synthesize these test environments for the desired test case, i.e., automatically synthesize test environments that reveal corner cases.

In this work, we focus on synthesizing test environments (e.g., placement of obstacles, agent strategies) to test the discrete decision-making logic in an autonomous system. Tests are synthesized from temporal logic descriptions of desired test behavior which encodes aspects of the test unknown to the system (test objective) in addition to the system requirements (system objective). The test objective is meant to capture the “challenging” aspect of the test in terms of high-level decision-making, and is not revealed to the system. The purpose of testing is to check that the system can take correct decisions despite being given opportunities to fail, i.e., verify correct decision in the presence of “hard tests” or corner cases. Our framework routes the system to the test objective while also giving the system freedom to make decisions and ensuring that the test is fair (i.e., system can satisfy its requirements if it makes correct decisions). Therefore, we synthesize tests that minimally restrict the system’s decision-making to realize the desired test behavior. Fig. 1 provides an overview of the flow-based test synthesis framework.

A. Background on Test and Evaluation

Tests are typically manually designed by test engineers — identifying challenging test cases and manually constructing the test environments either from expert experience or failure reports. Examples of this include the qualification tests in the DARPA Urban Challenge, track testing by self-driving car companies [2], [3], and constructing test scenarios in simulation using tools such as CARLA [4] or Scenic [5], for which test engineers either partially specify the scenarios or recreate them from crash reports [3], [6], [7]. Due to the time-intensive nature of this endeavor, automatically finding challenging tests for safety-critical systems is an active area of research [8], [9]. For self-driving vehicles, there is ongoing effort to standardize the testing process [10], [11].

This work was supported in by the U.S. Air Force Office of Scientific Research (AFOSR) under Grant FA9550-22-1-0333 and Grant FA9550-19-1-0302.

* These authors contributed equally. Corresponding author: A.S. Badithela.

J.B. Graebener is with the Graduate Aerospace Laboratories of California Institute of Technology, Pasadena CA 91125 USA (e-mail: jgraeben@caltech.edu).

A.S. Badithela, W. Ubellacker, A.D. Ames, R.M. Murray are affiliated with Control and Dynamical Systems, California Institute of Technology, Pasadena CA 91125 USA. (e-mail: {apurva, wubellac, ames, murray}@caltech.edu).

D. Goktas is with the Department of Computer Science, Brown University, Providence RI 02912 USA (e-mail: denizalp_goktas@brown.edu).

E.V. Mazumdar is with the Department of Computing and Mathematical Sciences, California Institute of Technology, Pasadena CA 91125 USA (e-mail: mazumdar@caltech.edu).

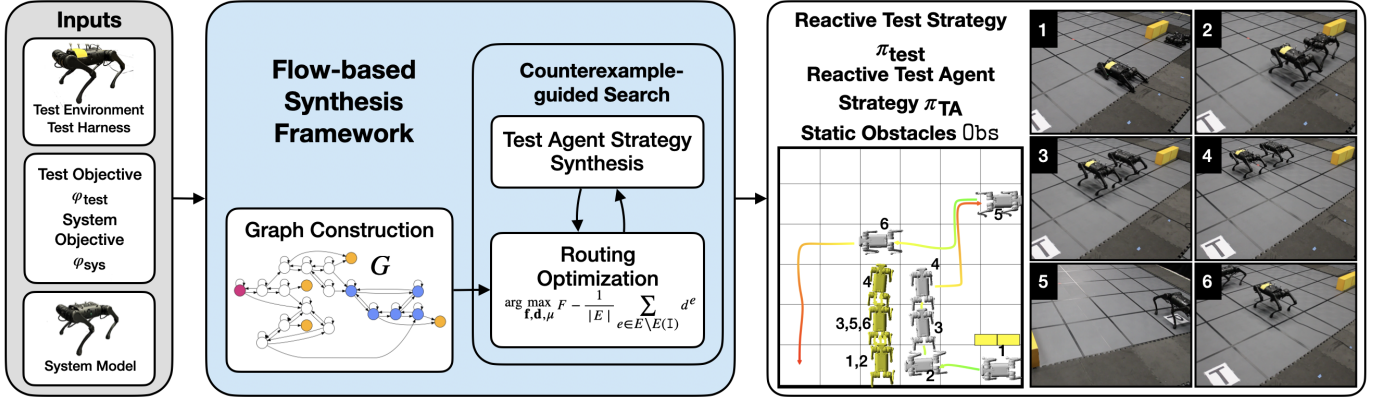


Fig. 1: Overview of the flow-based test synthesis framework which consists of three key parts: i) graph construction, ii) routing optimization, and iii) test environment synthesis (e.g., reactive test strategy / test agent strategy, static obstacles).

Black-box optimization algorithms [12] and reinforcement learning [13], [14] have been used to search over a specified input domain to find a falsifying input that leads to a trajectory that violates a metric of mission success. This metric can be derived from formal temporal logic specifications [15]–[21] or from control barrier functions [22]. However, falsification algorithms typically require a well-defined test environment, and find a falsifying trace by fine-tuning the parameters in that scenario. The framework proposed in this paper is complementary to these approaches — our focus is on synthesizing high-level strategies for the test environment, and continuous parameters of the synthesized test environment (e.g., continuous pose values of test agents, friction coefficients, exact timing of events) can be inputs to falsification algorithms for fine-tuning.

Typically, high-level choices of autonomous robotic systems exhibit discrete decision-making [23], [24]. The use of linear temporal logic (LTL) model checkers for testing has been explored in [25]–[28]. In these works, counterexamples from model-checking are used to construct test cases for deterministic systems and are inconclusive if the system behavior deviates from the expected test case. However, since robotic systems are often reactive, and because we want to generate tests without specific knowledge of the system controller, the generated tests must be able to *adapt* or *react* to system behavior at runtime. Our test synthesis procedure is gray-box in the sense that it requires knowledge of a nondeterministic model of the system but is agnostic to the high-level controller of the system and is completely black-box to models and controllers at lower levels of abstraction.

Adaptive specification-based testing using discrete logics has been explored in [29]–[33]. Particularly in [29], an adaptive test strategy is synthesized using reactive synthesis [34] from LTL specifications of the system and the fault model, both of which are specified by the test engineer. This adaptive test strategy ensures that the resulting test trace demonstrates a fault if the system implementation is faulty according to the fault model. However, these fault models must be carefully specified over the outputs of the system. While this is incredibly useful for specifying and catching sub-system level faults, it becomes intractable for specifying complex system-

level faults resulting from multiple outputs. Our test synthesis framework is also specification-based and adaptive, but we specify desired test behavior in the form of test objectives instead of specifying system-level faults. Furthermore, in [29], the adaptive test strategies are synthesized from fault models that are designed for coverage goals corresponding to specification coverage, without accounting for the freedom of the system to satisfy its own requirements. We seek to synthesize reactive test strategies that demonstrate the test objective while placing minimal restrictions on the system. The automata-theoretic tools used in this paper build on concepts used in correct-by-construction synthesis and model checking [35], [36]. This background is covered in Section II.

In [37], testing of reactive systems was introduced as a game between two players, where the tester and the system try to reveal and hide faults, respectively. Similarly, in [38] the test strategy is found by reasoning over a game graph to optimize reachability and coverage metrics. Testing in cooperative game settings has been explored in [39], [40]. However, the reactive test synthesis problem we consider is neither fully *adversarial* nor fully *cooperative* — a well-designed system is cooperative with the test environment in realizing the system objective, but since the system is agnostic to test objective, it need not cooperate with the test environment in realizing it.

We consider test environments that can consist of the following: static obstacles that restrict the system throughout the test, reactive obstacles and a dynamic test agent that is reactive to system behavior at runtime. In particular, we leverage flow networks to pose the test synthesis problem as a mixed-integer linear program (MILP). In recent years, network flow optimization frameworks with tight convex relaxations have led to massive computational speed-ups in solving robot motion planning problems [41], [42]. Network flow-based mixed integer programs have also been used to synthesize playable game levels in video games [43], which was then applied to construct playable scenarios in robotics settings [44].

In previous work [45], we formulated this problem in a semi-cooperative setting as a min-max Stackelberg game with coupled constraints. Despite being defined over continuous variables with an affine objective and affine constraints, the prior formulation resulted in slow runtimes and did not guar-

antee that the optimal solution would realize the test objective. Furthermore, it could only reactively restrict system actions, and did not characterize how to translate these restrictions to the choice of a test agent strategy. In this work, we present a simpler formulation of the routing optimization as an MILP, which led to an improvement in runtime. Test strategies from optimal solutions of the MILP are guaranteed to realize the test objective in a least restrictive fashion. Finally, we present a formal approach to restricting system actions in the form of static/reactive obstacles and dynamic test agent strategies, including a counterexample-guided approach to synthesize a test agent strategy from the solution of the routing optimization.

B. Contributions

In this work, we study the problem of synthesizing a reactive test strategy for a test environment for discrete decision-making systems given a formal test objective, unknown to the system under test. In particular, we ask whether such a test strategy exists without making it impossible for the system to meet its specification.

To obtain the main results of this paper, we first characterize system and test objectives using a variety of specification patterns commonly used in robotic missions [46]. We formalize both the restrictiveness and the feasibility of a test strategy, i.e., a system should have freedom to make decisions and a correct system should be able to pass the test. Secondly, these conditions are translated into a routing optimization on a flow network to capture the requirement that all test executions that satisfy the system objective should demonstrate the test objective. For each test environment, we set up an MILP to find cuts, corresponding to restrictions on system actions, on the flow network. For static and reactive obstacles, the solution of the MILP is realized in the form of an obstacle placement test strategy. We prove that the optimal solutions to the MILPs solve the aforementioned routing requirement. Third, in the case of dynamic agents, we match the restrictions on system actions to a test agent strategy via GR(1) synthesis [35], [47]. Furthermore, we use a counterexample-guided approach to exclude unrealizable solutions from the MILP until we find a realizable test agent strategy. We prove that test agent strategies synthesized in this manner exactly correspond to the test strategy found from the MILP. In the extended version, we prove that the routing problem is NP-hard via a reduction from 3-SAT. Despite this, our framework can reliably handle medium-sized problems with thousands of integer variables. Empirical runtimes for parametrized problems are also provided.

Finally, the test synthesis framework is demonstrated on simulated grid world settings and on hardware with a pair of quadrupedal robots. For all experiments, our framework synthesizes test strategies that place the fewest possible restrictions on the system over the course of the test either by obstacle placement or a dynamic agent. In experiments with reactive obstacles and dynamic agents, the reactive test strategy results in a different test execution depending on system behavior. Despite this, the system is always routed through the test objective (e.g., being put in low-fuel state or having to walk over challenging terrain).

II. PRELIMINARIES

This section introduces concepts from automata theory and network flows that are relevant to this work.

A. Automata Theory and Temporal Logic

Definition 1 (Finite Transition System). A *finite transition system* (FTS) is the tuple

$$TS := (S, A, \delta, S_0, AP, L),$$

where S denotes a finite set of states, A is a finite set of actions, $\delta : S \times A \rightarrow S$ the transition relation, S_0 the set of initial states, AP the set of atomic propositions, and $L : S \rightarrow 2^{AP}$ denotes the labeling function. We denote the transitions in TS as $TS.E := \{(s, s') \in S \times S \mid \exists a \in A \text{ s.t. } \delta(s, a) = s'\}$. We refer to the states of TS as $TS.S$, and similarly denote the other elements of the tuple. An execution σ is an infinite sequence $\sigma = s_0 s_1 \dots$, where $s_0 \in S_0$ and $s_k \in S$ is the state at time k . We denote the finite prefix of the trace σ up to the current time k as σ_k . A strategy π is a function $\pi : (TS.S)^* TS.S \rightarrow TS.A$.

Definition 2 (System). The *system under test* is modeled as a finite transition system T_{sys} with a single initial state, that is, $|T_{\text{sys}}.S_0| = 1$. Furthermore, at least one of the system states is terminal (i.e., no outgoing edges).

The system designers provide the states S , actions A , transitions δ , and a set of possible initial conditions S_0 , set of atomic propositions, AP_{sys} and a corresponding label function $L_{\text{sys}} : S \rightarrow 2^{AP_{\text{sys}}}$. We require a unique initial condition $s_0 \in S_0$ to synthesize the test. If the test designer wishes to select an initial condition, then they can synthesize the test for each $s_0 \in S_0$ and choose accordingly. In addition to AP_{sys} , the test designer can choose additional atomic propositions AP_{test} and define a corresponding labeling function $L : S \rightarrow 2^{AP}$, where $AP := AP_{\text{sys}} \cup AP_{\text{test}}$. For test synthesis, the system model is $T_{\text{sys}} = (S, A, \delta, \{s_0\}, AP, L)$ is defined for the specific initial condition s_0 chosen by the test designer. The terminal state is used for defining test termination when the system satisfies its objective.

Assumption 1. Except for sink states, transitions between states of the system are bidirectional: $\forall (s, s') \in T_{\text{sys}}.E$ where s' is not a terminal state, we also have $(s', s) \in T_{\text{sys}}.E$.

This assumption is for a simpler presentation, and the framework can be extended to transition systems without this assumption (see Remark 7).

Definition 3 (Test Harness). A *test harness* is used to constrain a state-action (s, a) pair of the system in the sense that the system is prevented from taking action a from state $s \in T_{\text{sys}}.S$. Let the actions $A_H \subseteq T_{\text{sys}}.A$ denote the subset of system actions that can be restricted by the test harness. The test harness $H : T_{\text{sys}}.S \rightarrow 2^{A_H}$ maps states of the transition system to actions that can be restricted from that state.

In the examples in this paper, every state of the system has a self-loop transition corresponding to stay-in-place action, but the proposed framework does not require this. Note that in our examples, A_H does not contain self-loop actions.

Definition 4 (Test Environment). The *test environment* consists of one or more of the following: static obstacles, reactive obstacles, and dynamic test agents. A *static obstacle* on $(s, s') \in T_{\text{sys}}.E$ is a restriction on the system transition (s, s') that remains in place for the entire duration of the test. A *reactive obstacle* on $(s, s') \in T_{\text{sys}}.E$ is a temporary restriction on the system transition (s, s') that can be enabled/disabled over the course of the test. A *dynamic test agent* can occupy states in $T_{\text{sys}}.S$, thus restricting the system from entering the occupied state.

In this work, we synthesize tests for high-level decision-making components of the system under test and therefore model it as a discrete-state system. Linear temporal logic (LTL) has been effective in formally specifying safety and liveness requirements for discrete-decision making [48]–[50]. For our problem, we use LTL to capture the system and test objectives.

Definition 5 (Linear Temporal Logic [36]). *Linear temporal logic* (LTL) is a temporal logic specification language that allows reasoning over linear-time trace properties. The syntax of LTL is given as:

$$\varphi ::= \text{True} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg \varphi \mid \bigcirc \varphi \mid \varphi_1 \mathcal{U} \varphi_2,$$

with $a \in AP$, where AP is the set of atomic propositions, $\wedge$ (conjunction) and $\neg$ (negation) are the Boolean connectors from which other Boolean connectives such as $\rightarrow$ can be defined, and $\bigcirc$ (next) and $\mathcal{U}$ (until) are temporal operators. Let φ be an LTL formula over AP . We can define the operators $\Diamond$ (eventually) and $\Box$ (always) as $\Diamond \varphi = \text{True} \mathcal{U} \varphi$ and $\Box \varphi = \neg \Diamond \neg \varphi$. For an execution $\sigma = s_0 s_1 \dots$ and an LTL formula φ , $s_i \models \varphi$ iff φ holds at $i \geq 0$ of σ . More formally, the semantics of LTL formula φ are inductively defined over an execution $\sigma = s_0 s_1 \dots$ as follows,

$$\begin{aligned} &\text{for } a \in AP, s_i \models a \text{ iff } a \text{ evaluates to True at } s_i, \\ &s_i \models \varphi_1 \wedge \varphi_2 \text{ iff } s_i \models \varphi_1 \text{ and } s_i \models \varphi_2, \\ &s_i \models \neg \varphi \text{ iff } \neg(s_i \models \varphi), \\ &s_i \models \bigcirc \varphi \text{ iff } s_{i+1} \models \varphi, \text{ and} \\ &s_i \models \varphi_1 \mathcal{U} \varphi_2 \text{ iff } \exists k \geq i, s_k \models \varphi_2 \text{ and } s_j \models \varphi_1, \text{ for all } i \leq j < k. \end{aligned}$$

An execution/trace $\sigma = s_0 s_1 \dots$ satisfies formula φ , denoted by $\sigma \models \varphi$, iff $s_0 \models \varphi$. A *strategy* π is correct (satisfies formula φ), if the trace σ_π resulting from the strategy satisfies φ .

Every LTL formula can be transformed into an equivalent non-deterministic Büchi automaton, which can then be converted to a deterministic Büchi automaton [36].

Definition 6 (Deterministic Büchi Automaton). A *non-deterministic Büchi automaton* (NBA) [36], [51] is a tuple $\mathcal{B} := (Q, \Omega, \delta, Q_0, F)$, where Q denotes the states, $\Omega := 2^{AP}$ is the set of alphabet for the set of atomic propositions AP , $\delta : Q \times \Omega \rightarrow Q$ denotes the transition function, $Q_0 \subseteq Q$ represents the initial states, and $F \subseteq Q$ is the set of acceptance states. The automaton is a *deterministic Büchi automaton* (DBA) iff $|Q_0| \leq 1$ and $|\delta(q, A)| \leq 1$ for all $q \in Q$ and $A \in \Omega$.

Remark 1. We use *deterministic* Büchi automata since each input word corresponding to an execution should have a

unique run on the automaton. While there are several different automata representations, deterministic Büchi automata are a natural choice for LTL specifications.

A *product* of two deterministic Büchi automata, $\mathcal{B}_1$ and $\mathcal{B}_2$ over the alphabet Ω , is defined as $\mathcal{B}_1 \otimes \mathcal{B}_2 := (Q, \Omega, \delta, Q_0, F)$, with states $Q := \mathcal{B}_1.Q \times \mathcal{B}_2.Q$, initial state $Q_0 := \mathcal{B}_1.Q_0 \times \mathcal{B}_2.Q_0$, acceptance states $F := \mathcal{B}_1.F \times \mathcal{B}_2.F$. The transition relation δ is defined as follows, for all $(q_1, q_2) \in Q$, for all $A \in \Omega$, $\delta((q_1, q_2), A) = (q'_1, q'_2)$ where $\mathcal{B}_1.\delta(q_1, A) = q'_1$ and $\mathcal{B}_2.\delta(q_2, A) = q'_2$.

The desired test behavior can be captured via sub-tasks that are defined over atomic propositions AP . Table I lists the sub-task specification patterns that are considered. These specification patterns are commonly used to specify robotic missions [46]. The desired test behavior is characterized by the system and test objectives, defined over the set of atomic propositions AP that can be evaluated on system states $T_{\text{sys}}.S$.

TABLE I: Sub-task specification patterns defined on atomic propositions.

Name	Formula	
Visit	$\bigwedge_{i=1}^m \Diamond p^i$	(s1)
Sequenced Visit	$\Diamond(p^0 \wedge (\Diamond(p^1 \wedge \dots \wedge p^m)))$	(s2)
Safety	$\Box \neg p$	(s3)
Instantaneous Reaction	$\Box(p \rightarrow q)$	(s4)
Delayed Reaction	$\Box(p \rightarrow \Diamond q)$	(s5)

Definition 7 (Test Objective). The *test objective* φ_{test} consists of at least one visit or sequenced visit sub-task or a conjunction of these sub-tasks. The Büchi automaton $\mathcal{B}_{\text{test}}$ corresponds to the test objective φ_{test} .

Definition 8 (System Objective). The *system objective* φ_{sys} consists of at least one visit or sequenced visit sub-task. The final visit proposition should be a terminal state of the system. In addition, it can also contain some conjunction of safety, instantaneous and/or delayed reaction, and visit and/or sequenced visit sub-tasks. The Büchi automaton $\mathcal{B}_{\text{sys}}$ corresponds to the system objective φ_{sys} . We say that the *system reaches its goal* or that the *test execution satisfies the system objective* if the system trace is accepted $\mathcal{B}_{\text{sys}}$.

Typically, some aspects of a test are not revealed to the system until test time such as testing the persistence of a robot or prompting it to exhibit a difficult maneuver by placing obstacles in its path. This is formalized as a test objective which is not known to the system. In contrast, the system is aware of the system objective, which captures its requirements. For example, to test for *safety*, the system should know to avoid unsafe areas (s3). To test a *reaction*, $\Box(p \rightarrow q)$, the system needs to be aware of the reaction requirement (s4), and the test objective needs to contain the corresponding visit requirement $\Diamond p$ to trigger the reaction. Furthermore, the test objective can contain standalone reachability (visit and/or

sequenced visit) sub-tasks that are not associated with a system reaction sub-task, but require the system to reach/visit certain states. The test objective is accomplished by restricting system actions in reaction to the system state.

In addition to the system objective, the system must interact safely with the test environment. The system must also obey the initial condition set by the test designer. For each obstacle/agent of the test environment, the system controller must respect the corresponding restrictions on its actions (i.e., cannot crash into obstacles/agents). Furthermore, for a valid system implementation, all lower-level planners and controllers of the system must simulate transitions on T_{sys} .

Definition 9 (System Guarantees). The system guarantees are a conjunction of the system objective, initial condition, safe interaction with the test environment, and a system implementation respecting the model T_{sys} .

Definition 10 (System Assumptions). The system *assumes* that the test environment satisfies the following conditions:

A1. The test environment can consist of: i) static obstacles (e.g., wall), ii) reactive obstacles (e.g., door), and iii) test agents whose dynamics are provided to the system.

A2. The test environment will not take any action that will inevitably lead to unsafe behavior (e.g., not restricting a system action after the system has committed to it, test agents not colliding into the system).

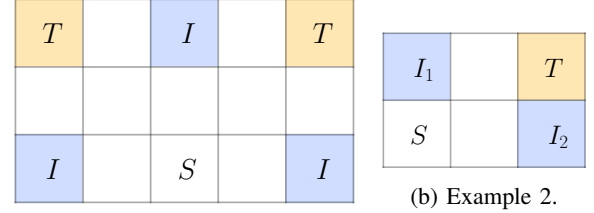
A3. The test environment will not take any action that will inevitably block all paths for the system to reach its goal (e.g., restrictions will not completely enclose the system or block it from progressing to its goal).

A4. If the system and test environment are in a livelock, the system will have the option to break the livelock and take a different path toward its goal.

A *correct system strategy* satisfies the system guarantees when the test environment satisfies the system assumptions. This full system specification cannot always be expressed as an LTL formula. This is because, in an LTL synthesis setting, the system can assume that the test environment can behave in a worst-case manner and will never synthesize a satisfying controller. However, the system can assume that the test environment will always ensure that a path to achieving the system specification remains. For many examples, expressing that a satisfying path exists is not possible in LTL.

Definition 11 (Specification Product). The *specification product* is the product $\mathcal{B}_\pi := \mathcal{B}_{\text{sys}} \otimes \mathcal{B}_{\text{test}}$, where $\mathcal{B}_{\text{sys}}$ is the Büchi automaton corresponding to the system specification, and $\mathcal{B}_{\text{test}}$ is the Büchi automaton corresponding to the test objective. The states $(q_{\text{sys}}, q_{\text{test}}) \in \mathcal{B}_\pi \cdot Q$, where $q_{\text{sys}} \in \mathcal{B}_{\text{sys}} \cdot Q$ and $q_{\text{test}} \in \mathcal{B}_{\text{test}} \cdot Q$, capture the event-based progression of the test and are referred to as history variables.

The system reaching its goal would typically mark the end of a test execution. However, the test engineer can also decide to terminate the test if the system appears to be stuck or enters an unsafe state. Tests that are terminated prematurely might result in inconclusive results [52], so we rely on the test engineer to determine the termination condition. We assume



(a) Example 1.

Fig. 2: Grid world layouts for examples.

that the test engineer gives the system a reasonable amount of time to complete the test. Upon test termination in state s_n , we augment the trace σ with the infinite suffix s_n^ω for evaluation purposes.

Remark 2. As tests have a defined start and end point, we need to bridge the gap between the finiteness of test executions and the infinite traces that are needed to evaluate LTL formulae. Augmenting the trace with the infinite suffix allows us to leverage useful tools available for LTL. Other research on interpreting LTL over finite traces can be found in [29], [53], [54].

Remark 3. The states of the specification product automaton track the states of the individual Büchi automata, $\mathcal{B}_{\text{sys}}$ and $\mathcal{B}_{\text{test}}$, in the form of the Cartesian product to remember accepting states of the individual automata, which will be necessary for our framework (see Definitions 11, 19).

Example 1. The system under test can transition (N-S-E-W) on the grid world as illustrated in Fig. 2a. The initial condition of the system is marked by S , and the system is required to visit one of the terminal goal states marked by T , $\varphi_{\text{sys}} = \Diamond T$. The test objective is to observe the system visit at least one of the I states before the system reaches its goal, encoded as $\varphi_{\text{test}} = \Diamond I$.

Example 2. In this example, the system under test can transition (N-S-E-W) on the grid world as illustrated in Fig. 2b. The initial condition of the system is marked by S , and the system objective is to visit terminal state T , $\varphi_{\text{sys}} = \Diamond T$. The test objective is to observe the system visit states I_1 and I_2 : $\varphi_{\text{test}} = \Diamond I_1 \wedge \Diamond I_2$. The corresponding Büchi automata are illustrated in Fig. 3.

The synchronous product operator is used to construct a product of a transition system and a Büchi automaton. In particular, we will use this operator to construct the virtual product graph and the system product graph (see Section III).

Definition 12 (Synchronous Product). The *synchronous product* of a DBA $\mathcal{B}$ and a FTS T_{sys} , where the alphabet of $\mathcal{B}$ is the

labels of T_{sys} , is the transition system $P := T_{\text{sys}} \otimes \mathcal{B}$, where:

$$\begin{aligned}
 P.S &:= T_{\text{sys}}.S \times \mathcal{B}.Q, \\
 P.\delta((s, q), a) &:= (s', q') \text{ if } \forall s, s' \in T_{\text{sys}}.S, \forall q, q' \in \mathcal{B}.Q, \\
 &\quad \exists a \in T_{\text{sys}}.A, \text{ s.t. } T_{\text{sys}}.\delta(s, a) = s' \text{ and } \mathcal{B}.\delta(q, T_{\text{sys}}.L(s')) = q', \\
 P.S_0 &:= \{(s_0, q) \mid s_0 \in T_{\text{sys}}.S_0, \exists q_0 \in \mathcal{B}.Q_0 \text{ s.t.} \\
 &\quad \mathcal{B}.\delta(q_0, T_{\text{sys}}.L(s_0)) = q\}, \\
 P.AP &:= \mathcal{B}.Q, \\
 P.L((s, q)) &:= \{q\}, \quad \forall (s, q) \in P.S.
 \end{aligned}$$

We denote the transitions in P as $P.E := \{(s, s') \mid s, s' \in P.S \text{ if } \exists a \in P.A \text{ s.t. } P.\delta(s, a) = s'\}$. An infinite sequence on P corresponds to a state-history trace $\vartheta = (s, q)_0, (s, q)_1, \dots, (s, q)_n^\omega$. We refer to $(s, q) \in P.S$ as the state-history pair and define the corresponding path to be the finite prefix: $\vartheta_n = (s, q)_0, (s, q)_1, \dots, (s, q)_n$.

B. Network Flows

Definition 13 (Flow Network [55]). A *flow network* is a tuple $\mathcal{N} = (V, E, (V_s, V_t))$, where V denotes the set of nodes, $E \subseteq V \times V$ the set of edges excluding self-loops, $V_s \subseteq V$ the source nodes, and $V_t \subseteq V$ the sink nodes. We assume unit capacity for all edges. On the flow network $\mathcal{N}$, we can define the *flow vector* $\mathbf{f} \in \mathbb{R}_{\geq 0}^{|E|}$ to satisfy the following constraints:

$$0 \leq f^e \leq 1, \forall e \in E, \quad (6)$$

ii) the conservation constraint

$$\sum_{u \in V} f^{(u, v)} = \sum_{u \in V} f^{(v, u)}, \forall v \in V \setminus \{V_s, V_t\}, \text{ and} \quad (7)$$

iii) no flow into the source or out of the sink

$$f^{(u, v)} = 0 \text{ if } u \in V_t \text{ or } v \in V_s. \quad (8)$$

The flow value on the network $\mathcal{N}$ is defined as

$$F := \sum_{\substack{(u, v) \in E, \\ u \in V_s}} f^{(u, v)}. \quad (9)$$

III. PROBLEM STATEMENT

In this section, we will state the test environment synthesis problem. The test engineer provides a system objective and a test objective, which describes the desired test behavior. Then, we find a reactive test strategy for which every test execution that satisfies the system objective also satisfies the test objective.

Definition 14 (Reactive Test Strategy). A *reactive test strategy* $\pi_{\text{test}} : (T_{\text{sys}}.S)^* T_{\text{sys}}.S \rightarrow 2^{A_H}$ defines the set of restricted system actions at each state during its execution σ . For some finite prefix $s_0 \dots s_i$ of execution σ starting from initial state $s_0 \in T_{\text{sys}}.S_0$, $\pi_{\text{test}}(s_0 \dots s_i) \subseteq H(s_i)$ is the set of actions that the system cannot take from state s_i . A test environment is said to *realize* a reactive test strategy π_{test} if it restricts system actions according to π_{test} .

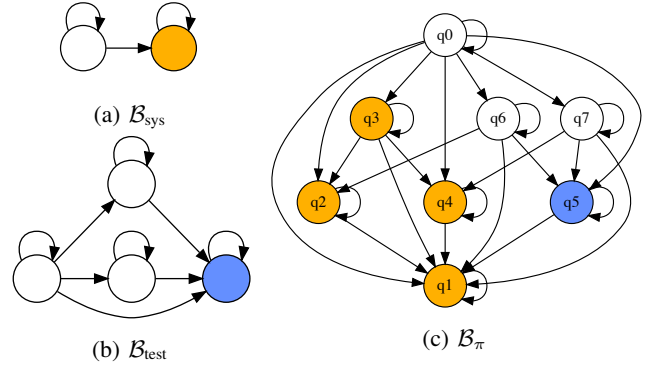


Fig. 3: Automata for Example 2. Yellow $\bullet$ and blue $\bullet$ nodes in $\mathcal{B}_{\text{sys}}$ and $\mathcal{B}_{\text{test}}$ are the respective accepting states. In the product $\mathcal{B}_{\pi}$, we continue to track these states for the system and test objectives. States in the product $\mathcal{B}_{\pi}$ that are accepting to both objectives (e.g., q_1) are also shaded yellow.

Let $\Sigma_{\text{fin}} := (T_{\text{sys}}.S)^* T_{\text{sys}}.S$ be the set of all finite prefixes of system traces. At each time step $k \geq 0$, a correct system strategy $\pi_{\text{sys}} : \Sigma_{\text{fin}} \rightarrow T_{\text{sys}}.A \setminus \pi_{\text{test}}(\Sigma_{\text{fin}})$ must pick from available actions at state s_k . The resulting execution is denoted as $\sigma(\pi_{\text{sys}} \times \pi_{\text{test}})$.

Remark 4. Note that the test environment externally blocks system transitions, and as a consequence, restricts corresponding actions that the system can safely take. When actions are restricted by the test environment, the system strategy π_{sys} should select from the available actions at each state. Since these restrictions can be placed during the test execution, the system might have to re-plan and choose a different action than originally planned.

Definition 15 (Feasibility of a Test Strategy). Given a test environment, system T_{sys} , system and test objectives, φ_{sys} and φ_{test} , a reactive test strategy π_{test} is said to be *feasible* iff: i) the test environment can realize π_{test} , ii) there exists a correct system strategy π_{sys} , and iii) any execution corresponding to a correct π_{sys} satisfies the system and test objectives: $\sigma(\pi_{\text{sys}} \times \pi_{\text{test}}) \models \varphi_{\text{test}} \wedge \varphi_{\text{sys}}$.

Note that the test strategy is not aiding the system in achieving the system objective; it only restricts system actions such that the test objective is realized. That is, the system is free to choose an incorrect strategy, in which case there are no guarantees. Furthermore, the test strategy should allow the system to make multiple decisions at each step of the execution, if possible, as opposed to leaving a single allowed action. For any system trace $\sigma = s_0 s_1 \dots$, every finite prefix of σ maps to a history variable $q \in \mathcal{B}_{\pi}.Q$. For each σ , we can define a corresponding state-history trace $\vartheta = (s, q)_0, (s, q)_1, \dots$, where history variable q at time step i corresponds to the prefix of $s_0 \dots s_i$ of σ . From now on, we will refer to σ and the associated ϑ as the test execution, and clarify the context if necessary.

Definition 16 (Restrictiveness of a Test Strategy). State-history traces ϑ_1 and ϑ_2 are *unique* if they do not share any consecutive state-history pairs. For a feasible π_{test} , let Σ be the

set of all executions corresponding to correct system strategies, and let Θ be the set of all state-history traces corresponding to Σ . Let $\Theta_u \subseteq \Theta$ be a set of unique state-history traces. A test strategy π_{test} is *least restrictive* if the cardinality of Θ_u is maximized.

Remark 5. Note that the set of all state history traces Θ can be infinite. However, the set Θ_u is finite because: i) the system has a finite number of states and the specification product has a finite number of history variables, and ii) every state-history trace in Θ_u is *unique* with respect to any other trace in Θ_u .

Problem 1 (Finding a Reactive Test Strategy). Given a high-level abstraction of the system model T_{sys} , test harness H , system objective φ_{sys} , test objective φ_{test} , find a feasible, reactive test strategy π_{test} that is least restrictive.

The restrictions on system actions placed by the test strategy can be realized in several ways in the test environment. For example, a dynamic test agent, together with any static obstacles, can be used to enforce the test strategy. This leads to the second problem of synthesizing a reactive strategy for a test agent to realize the test strategy. That is, at each time step of the test execution, the test environment consisting of an agent and static obstacles restricts the system actions according to π_{test} .

Problem 2 (Reactive Test Agent Strategy Synthesis). Given a high-level abstraction of the system model T_{sys} , test harness H , system objective φ_{sys} , test objective φ_{test} , and a test agent modeled by transition system T_{TA} . Find the test agent strategy π_{TA} and the set of static obstacles Obs that: i) satisfy the system's assumptions on its environment, and ii) realize a reactive test strategy π_{test} that is least-restrictive and feasible.

IV. GRAPH CONSTRUCTION

To reason about executions of the system in relation to the system and test objectives, we leverage automata theory to construct the following graphs.

Definition 17 (Virtual Product Graph and System Product Graph). A *virtual product graph* is the product transition system $G := T_{\text{sys}} \otimes \mathcal{B}_{\pi}$. Similarly, the system product graph is defined as $G_{\text{sys}} := T_{\text{sys}} \otimes \mathcal{B}_{\text{sys}}$.

The virtual product graph G tracks the test execution in relation to both the system and test objectives while the system product graph G_{sys} tracks the system objective. We will find the restrictions on system actions on G , while G_{sys} represents the system's perspective concerning the system objective during the test execution. For each node $u = (s, q) \in G.S$, we denote the corresponding state in $s \in T_{\text{sys}}.S$ as $u.s := s$. Similarly, the state corresponding to $v \in G_{\text{sys}}.S$ is denoted by $v.s := s$. For practical implementation, we remove nodes on the product graphs that are not reachable from the corresponding initial states, $G.S_0$ or $G_{\text{sys}}.S_0$.

Definition 18 (Projection). We map states from G to G_{sys} using the *projection* $\mathcal{P}_{G \rightarrow G_{\text{sys}}} : G.S \rightarrow G_{\text{sys}}.S$ as

$$\mathcal{P}_{G \rightarrow G_{\text{sys}}}(s, (q_{\text{sys}}, q_{\text{test}})) = (s, q_{\text{sys}}). \quad (10)$$

These projections help us to reason about how restrictions found on G map to the system T_{sys} and the system product graph G_{sys} . We can now define the edges on G that we can restrict with the test harness as follows,

$$E_H = \{((s, q), (s', q')) \in G.E \mid \forall s \in T_{\text{sys}}.S, \forall a \in H(s) \text{ s.t. } s' = T_{\text{sys}}.\delta(s, a)\}. \quad (11)$$

Lemma 1. For every path $(s, q_{\text{sys}})_0, (s, q_{\text{sys}})_1, \dots, (s, q_{\text{sys}})_n$ on G_{sys} , there exists at least one corresponding path on G .

Proof. Suppose there exists some $q_{\text{test } 0}, \dots, q_{\text{test } n} \in \mathcal{B}_{\text{test}}.Q$ such that $(s, (q_{\text{sys}}, q_{\text{test}}))_0, \dots, (s, (q_{\text{sys}}, q_{\text{test}}))_n$ is a path on G . Then, by construction, there exists a path on G_{sys} where $(s, (q_{\text{sys}}, q_{\text{test}}))_k$ maps to $(s, q_{\text{sys}})_k$ for all $0 \leq k \leq n$. $\square$

Paths on the virtual product graph G correspond to possible test executions. We identify the nodes on G that capture the acceptance conditions for the system and test objectives.

Definition 19 (Source, Intermediate, and Target Nodes). The *source node* S represents the initial condition of the system. The *intermediate nodes* I correspond to system states in which the test objective acceptance conditions are met. Finally, the *target nodes* T represent the system states for which the acceptance condition for the system objective is satisfied. Formally, these nodes are defined as follows,

$$\begin{aligned} S &:= \{(s_0, q_0) \in G.S \mid s_0 \in T_{\text{sys}}.S_0, q_0 \in \mathcal{B}_{\pi}.Q_0\}, \\ I &:= \{(s, (q_{\text{sys}}, q_{\text{test}})) \in G.S \mid q_{\text{test}} \in \mathcal{B}_{\text{test}}.F, q_{\text{sys}} \notin \mathcal{B}_{\text{sys}}.F\}, \\ T &:= \{(s, (q_{\text{sys}}, q_{\text{test}})) \in G.S \mid q_{\text{sys}} \in \mathcal{B}_{\text{sys}}.F\}. \end{aligned}$$

In addition, we define the set of states corresponding to the system acceptance condition on G_{sys} as $T_{\text{sys}} := \{(s, q) \in G_{\text{sys}}.S \mid q \in \mathcal{B}_{\text{sys}}.F\}$.

Proposition 1. Every test execution corresponds to a path $\vartheta_n = (s, q)_0, (s, q)_1, \dots, (s, q)_n$ on G where $(s, q)_0 \in S$. The corresponding system trace σ_n satisfies the system objective, $\sigma \models \varphi_{\text{sys}}$ iff $(s, q)_n \in T$. Furthermore, if $\sigma \models \varphi_{\text{test}}$, then the path ϑ_n contains a state-history pair $(s, q)_i \in I$ for some $0 \leq i \leq n$.

Provided that there exists a path on G from S to T , identifying a feasible reactive test strategy corresponds to identifying edges to cut on G . These edge cuts correspond to restricted system actions. In particular, these edge cuts are such that all paths on G from source S to target T visit the intermediate I .

V. NETWORK FLOW OPTIMIZATION FOR IDENTIFYING RESTRICTIONS ON SYSTEM ACTIONS

To identify which edges to cut on G , we use network flow optimization, a commonly used paradigm for flow-cut problems on graphs. On G , which characterizes all possible test executions, all paths from the initial condition S to the system goal T must be routed through the intermediate I . Furthermore, the edge cuts should be least-restrictive and such that the system can satisfy the test objective and system objective. Maximum flow can be a proxy for freedom of the system under test to make decisions — a higher network flow corresponds to more unique paths on G . Since we use flow

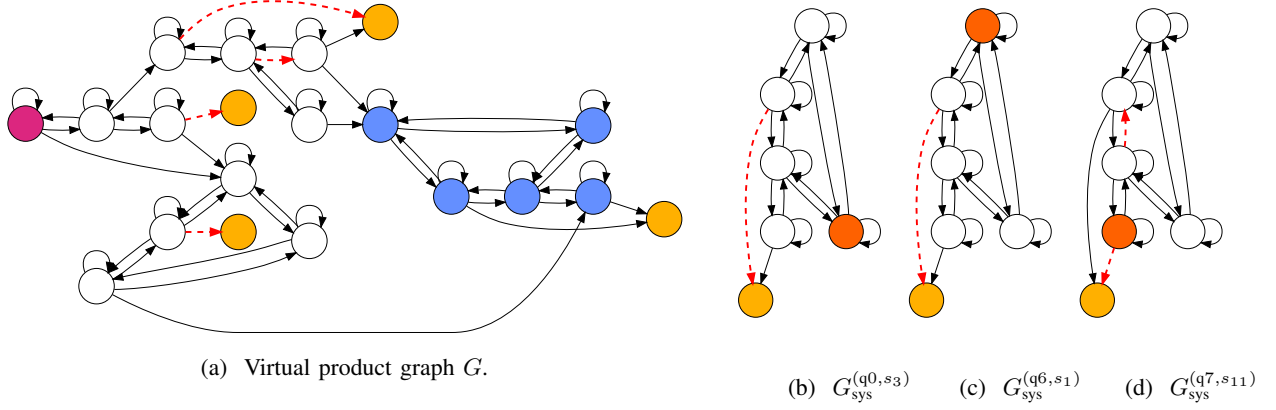


Fig. 4: Virtual product graph and system product graphs for Example 2. Fig. 4a shows the virtual product graph G , with the source S (magenta $\bullet$), the intermediate nodes I (blue $\bullet$), and the target nodes (yellow $\bullet$). Edge cut values for each edge in G are grouped by their history variable q and projected to the corresponding copy of G_{sys} . Red dashed lines indicate edge cuts. Figs. 4b-4d show the copies of G_{sys} with their source (s_3 , s_6 or s_{11} in orange $\bullet$) and target nodes (yellow $\bullet$). The graphs in Figs. 4b-4d correspond to the history variables q_0 , q_6 , and q_7 from $\mathcal{B}_\pi$ shown in Fig. 3c. The constraints (c6)-(c8) ensure that the edge cuts are such that a path from each source to the target node exists for each history variable q .

networks with unit edge capacities, a realization of maximum flow corresponds to a set of paths that do not share an edge. Furthermore, this flow should be achieved with the fewest possible cuts to not unnecessarily restrict system actions. A high network flow with the minimum possible edge cuts corresponds to a least restrictive test for the system.

A. Optimization Setup

We define the flow network $\mathcal{G} := (V, E, (S, T))$, where $V := G.S$, $E := G.E$, source and target nodes correspond to S and T , with the corresponding flow $\mathbf{f} \in \mathbb{R}^{|E|}$. For simplicity, we use the same notation to refer to nodes and edges on the graph and the corresponding flow network. The Boolean edge cut vector $\mathbf{d} \in \mathbb{B}^{|E|}$ represents whether edges are cut or not. That is, $d^e = 1$ refers to edge $e \in E$ being cut, and $d^e = 0$ implies that edge e is not cut,

$$d^e \in \{0, 1\}, \quad \forall e \in E, \text{ and } d^e = 0, \quad \forall e \notin E_H. \quad (\text{c1})$$

The edges into and out of the intermediate I nodes are denoted as $E(I) := \{(u, v) \in E | u \in I \text{ or } v \in I\}$. To solve Problem 1, we formulate a mixed-integer linear program (MILP).

Objective. To find the least restrictive test, we want to maximize the system's freedom in satisfying the test objective. To capture this, we optimize for edge cuts that maximize the flow value on $\mathcal{G}$. However, a realization of maximum flow on a network is not unique. To ensure that we do not cut any edges unnecessarily, we subtract the sum of the edge cuts from the flow value:

$$\sum_{\substack{(u,v) \in E, \\ u \in S}} f^{(u,v)} - \frac{1}{|E|} \sum_{e \in E} d^e. \quad (12)$$

The regularizer $\frac{1}{|E|}$ on the sum of edge cuts is chosen such that it will not compete with the maximum flow value on the network. The weighted sum $\frac{1}{|E|} \sum_{e \in E} d^e$ is always between 0 and 1, and binary edge cuts and unit capacity will always result

in maximum flow being integer-valued. Thus, the optimization will always favor increasing the maximum flow value rather than reducing edge cuts.

Network flow constraints. First, the network flow optimization is subject to the following standard constraints on flow $\mathbf{f}$:

Flow constraints (6), (7), and (8) on flow network $\mathcal{G}$. (c2)

An edge that is cut restricts flow completely, while an edge that is not cut may or may not have flow,

$$\forall e \in E, \quad d^e + f^e \leq 1. \quad (\text{c3})$$

Partition constraints. The following constraints ensure that all flow across the network will be routed through I . To accomplish this, we adapt the partitioning conditions given in [56] as follows. Except for the I nodes, we divide the remaining nodes into two groups defined by the partition variable $\mu \in \mathbb{R}^{|V \setminus I|}$, and ensure that the nodes S belong to one group, and T belong to the other:

$$0 \leq \mu^v \leq 1, \quad \mu^S - \mu^T \geq 1, \forall v \in V \setminus I. \quad (\text{c4})$$

The two groups are partitioned by the edge cut vector $\mathbf{d}$, where this constraint is only defined over the edges that do not go into or out of nodes in I ,

$$d^{(u,v)} - \mu^u + \mu^v \geq 0, \forall (u, v) \in E \setminus E(I). \quad (\text{c5})$$

Feasibility constraints. To ensure that the test is not impossible from the system's perspective, we map restrictions found on G to G_{sys} via the following feasibility constraints. For each history variable $q \in \mathcal{B}_\pi.Q$, we define the set of state-history pairs that captures the possible first observations of the history variable in a test execution via the function $S_G : \mathcal{B}_\pi.Q \rightarrow G.S$ defined as follows,

$$S_G(q) := \{(s, q) \in G.S \mid \forall ((\bar{s}, \bar{q}), (s, q)) \in G.E, \bar{q} \neq q\}. \quad (13)$$

These sets of states are mapped to G_{sys} as follows:

$$\mathbf{S}_{G_{\text{sys}}}(q) := \{u \in G_{\text{sys}}.S \mid u = \mathcal{P}_{G \rightarrow G_{\text{sys}}}(v), \\ v \in \mathbf{S}_G(q), \text{ and } \exists \text{ path}(u, \mathbf{T}_{\text{sys}})\}, \quad (14)$$

where this set is empty if no path from the node u to $\mathbf{T}_{\text{sys}}$ exists on G_{sys} . For each $q \in \mathcal{B}_{\pi}.Q$, for each source in $\mathbf{s} \in \mathbf{S}_{G_{\text{sys}}}(q)$, we define a flow network $\mathcal{G}_{\text{sys}}^{(q,\mathbf{s})} := (V_{\text{sys}}, E_{\text{sys}}, c, (\mathbf{s}, \mathbf{T}_{\text{sys}}))$, where $V_{\text{sys}} := G_{\text{sys}}.S$, and $E_{\text{sys}} := G_{\text{sys}}.E$, with the corresponding flow variable $\mathbf{f}_{\text{sys}}^{(q,\mathbf{s})}$. For each of these flow networks, we define a flow subject to the standard flow constraints:

$$\forall q \in \mathcal{B}_{\pi}.Q, \forall \mathbf{s} \in \mathbf{S}_{G_{\text{sys}}}(q), \quad (c6)$$

Flow constraints (6), (7), and (8) on network $\mathcal{G}_{\text{sys}}^{(q,\mathbf{s})}$.

For each $\mathcal{G}_{\text{sys}}^{(q,\mathbf{s})}$, we map the edge cuts $\mathbf{d}$ and check that there is still a path from $\mathbf{s}$ to some node in $\mathbf{T}_{\text{sys}}$. This ensures that reactively placing restrictions on system actions does not make it impossible for a correct system strategy to make progress toward its goal. Intuitively, the edge cuts are grouped by the history variable q and checked to ensure that the system has a feasible path when these restrictions are placed on system actions. The edges are grouped by their history variable using the mapping $\text{Gr} : \mathcal{B}_{\pi}.Q \rightarrow 2^{G.E}$, defined as follows:

$$\text{Gr}(q) := \{(s, q), (s', q')\} \in G.E\}. \quad (15)$$

The edge cuts are mapped onto the corresponding $\mathcal{G}_{\text{sys}}^{(q,\mathbf{s})}$ to cut the corresponding flow $\mathbf{f}_{\text{sys}}^{(q,\mathbf{s})}$ as follows:

$$\forall q \in \mathcal{B}_{\pi}.Q, \forall \mathbf{s} \in \mathbf{S}_{G_{\text{sys}}}(q), \forall (u, v) \in \text{Gr}(q), \forall (u', v') \in E_{\text{sys}}, \\ d^{(u,v)} + f_{\text{sys}}^{(q,\mathbf{s})}(u', v') \leq 1, \text{ if } u'.s = u.s \text{ and } v'.s = v.s. \quad (c7)$$

Since we are agnostic to the system controller, we need to ensure that a path to the system's goal exists at all times during the test execution. To enforce this, we require a flow of at least 1 on each system flow network $\mathcal{G}_{\text{sys}}^{(q,\mathbf{s})}$,

$$\sum_{(\mathbf{s}, v) \in E_{\text{sys}}} f_{\text{sys}}^{(q,\mathbf{s})}(\mathbf{s}, v) \geq 1, \forall q \in \mathcal{B}_{\pi}.Q, \forall \mathbf{s} \in \mathbf{S}_{G_{\text{sys}}}(q). \quad (c8)$$

These feasibility cuts correspond to the reactive constraint setting since edge cuts are placed on $\mathcal{G}$ and depend on the history variable q . For an illustrated explanation for Example 2, refer to Fig. 4. Finally, the optimization to identify edge cuts for the reactive test strategy is characterized by the following mixed-integer linear program (MILP) with the cuts $\mathbf{d}$ as the integer variables, and the flow and partition variables taking continuous values.

MILP-REACTIVE:

$$\max_{\mathbf{f}, \mathbf{d}, \mu, \mathbf{f}_{\text{sys}}^{(q,\mathbf{s})} \forall q \in \mathcal{B}_{\pi}.Q \forall \mathbf{s} \in \mathbf{S}_{G_{\text{sys}}}(q)} F - \frac{1}{|E|} \sum_{e \in E} d^e \quad (16)$$

s.t. (c1)-(c3), (c4)-(c5), (c6)-(c8).

Static Constraints. We can simplify the feasibility constraints in the case of static obstacles. This corresponds to the requirement that any transition that is restricted will remain restricted for the entire duration of the test. From the system's perspective, the restrictions will not change depending on the history variable q . That is, edges in G corresponding to the same transition in $T_{\text{sys}}.E$ are grouped and share the same cut value:

$$d^{(u,v)} = d^{(u',v')}, \forall (u, v), (u', v') \in E, \\ \text{if } u.s = u'.s \text{ and } v.s = v'.s. \quad (c9)$$

Similarly, the optimization to find edge cuts in a static setting is as follows.

MILP-STATIC:

$$\max_{\mathbf{f}, \mathbf{d}, \mu} F - \frac{1}{|E|} \sum_{e \in E} d^e \quad (17)$$

s.t. (c1)-(c3), (c4)-(c5), (c9).

Lemma 2. For the case of static constraints, due to (c9), ensuring feasibility from the system's perspective is guaranteed by checking $F > 0$ on G . That is, $F > 0$ on G is equivalent to checking (c6)-(c8).

Proof. Under (c9), the edge groupings $\text{Gr}(q)$ become the same for all $q \in \mathcal{B}_{\pi}.Q$. Thus, the constraints (c6)-(c8) can be reduced onto a single flow network $\mathcal{G}_{\text{sys}} = (V_{\text{sys}}, E_{\text{sys}}, (\mathbf{S}_{\text{sys}}, \mathbf{T}_{\text{sys}}))$, where $\mathbf{S}_{\text{sys}} := G_{\text{sys}}.I$. Equation (c8) being satisfied on $\mathcal{G}_{\text{sys}}$ implies that there is a path on G from $\mathbf{S}$ to $\mathbf{T}$ via Lemma 1. Additionally, if there is a path on G from $\mathbf{S}$ to $\mathbf{T}$ with the static constraints (c9), then it must be that there exists a path from $\mathbf{S}_{\text{sys}}$ to $\mathbf{T}_{\text{sys}}$ on G_{sys} . $\square$

Remark 6. For the reactive constraint setting, we can replace the feasibility constraints (c6)-(c8) by several static constraints. That is, we introduce a copy of $\mathcal{G}$ for each history variable $q \in \mathcal{B}_{\pi}.Q$ and each source $\mathbf{s} \in \mathbf{S}_G(q)$, denoted $\mathcal{G}^{(q,\mathbf{s})} = (V, E, \mathbf{s}, \mathbf{T})$, and require a path from $\mathbf{s}$ to $\mathbf{T}$ to exist under a static mapping of the edges in the group $\text{Gr}(q)$ by constraint (c9). We choose the former since it reduces the number of variables and constraints in the optimization.

Mixed Constraints. In some cases, it might be desirable to define specific transitions $T_{\text{sys}}.E_{\text{static}} \subseteq T_{\text{sys}}.E$ which require static constraints. The mixed setting of reactive and static transition restrictions can be implemented by enforcing the feasibility constraints (c6)-(c8), and the static constraints (c9) on edges $(u, v) \in E$, where the corresponding transition $(u.s, v.s) \in T_{\text{sys}}.E_{\text{static}}$. Finally, the optimization for the mixed constraint setting is as follows.

MILP-MIXED:

$$\max_{\mathbf{f}, \mathbf{d}, \mu, \mathbf{f}_{\text{sys}}^{(q,\mathbf{s})} \forall q \in \mathcal{B}_{\pi}.Q \forall \mathbf{s} \in \mathbf{S}_{G_{\text{sys}}}(q)} F - \frac{1}{|E|} \sum_{e \in E} d^e \quad (18)$$

s.t. (c1)-(c3), (c4)-(c5), (c6)-(c8), (c9).

Auxiliary Constraints. Additional constraints can be added to the optimization depending on the test harness or the desired test setup. For example, it might be required to enforce that if an edge is cut, the transition will be blocked in both directions. This can be enforced as follows,

$$d^{(u,v)} = d^{(u',v')}, \forall (u,v), (u',v') \in E, \quad (\text{c14})$$

$$\text{if } u.s = v'.s \text{ and } v.s = u'.s.$$

Algorithm 1: Finding the test strategy π_{test}

```

1: procedure FINDTESTSTRATEGY( $T_{\text{sys}}, H, \varphi_{\text{sys}}, \varphi_{\text{test}}$ )
   Input: transition system  $T_{\text{sys}}$ , test harness  $H$ , system
   objective  $\varphi_{\text{sys}}$ , test objective  $\varphi_{\text{test}}$ 
   Output: test strategy  $\pi_{\text{test}}$ 
2:  $\mathcal{B}_{\text{sys}} \leftarrow \text{BA}(\varphi_{\text{sys}})$  ▷ System Büchi automaton
3:  $\mathcal{B}_{\text{test}} \leftarrow \text{BA}(\varphi_{\text{test}})$  ▷ Tester Büchi automaton
4:  $\mathcal{B}_{\pi} \leftarrow \mathcal{B}_{\text{sys}} \otimes \mathcal{B}_{\text{test}}$  ▷ Specification product
5:  $G_{\text{sys}} \leftarrow T_{\text{sys}} \otimes \mathcal{B}_{\text{sys}}$  ▷ System product
6:  $G \leftarrow T_{\text{sys}} \otimes \mathcal{B}_{\pi}$  ▷ Virtual Product Graph
7:  $S, I, T \leftarrow \text{IDENTIFYNODES}(G, \mathcal{B}_{\text{sys}}, \mathcal{B}_{\text{test}})$ 
8:  $\mathcal{G} \leftarrow \text{DEFINENETWORK}(G, S, T)$ 
9:  $\mathcal{G} \leftarrow \text{set}()$  ▷ System Perspective Graphs
10: for  $q \in \mathcal{B}_{\pi}.Q$  do
11:   for  $s \in S_{G_{\text{sys}}}(q)$  do
12:      $\mathcal{G}_{\text{sys}}^{(s,q)} \leftarrow \text{DEFINENETWORK}(G_{\text{sys}}, s, T_{\text{sys}})$ 
13:      $\mathcal{G} \leftarrow \mathcal{G} \cup \mathcal{G}_{\text{sys}}^{(s,q)}$ 
14:    $\mathbf{d}^* \leftarrow \text{MILP}(\mathcal{G}, T_{\text{sys}}, \mathcal{G}, I, H)$  ▷ Reactive, static, or mixed.
15:    $C \leftarrow \{(u,v) \in G.E \mid \mathbf{d}^{(u,v)} = 1\}$  ▷ Cuts on  $G$ 
16:    $\pi_{\text{test}} \leftarrow \text{Define test strategy according to equation (20)}$ 
17: return  $\pi_{\text{test}}$ 

```

B. Characterizing Optimization Results

The flow value (Eq. (9)) of the network is always integer-valued since the edge cuts are binary and edges have unit capacities, and therefore, any strictly positive flow value corresponds to at least one valid test execution. In the following cases, the problem data are *inconsistent* and a flow value ≥ 1 cannot be found.

Case 1: There is no path from S to T on G (and equivalently, no path from S_{sys} to T_{sys} on G_{sys}). In this case, the optimization will not have to place any cuts because the only possible maximum flow value is 0.

Case 2: There is a path from S to T on G , but there is no path S to T in G visiting an intermediate node in I . In this case, the partition constraints will cut all paths from S to T , while by Lemma 1 the feasibility constraints require a path to exist from S to T —a contradiction. The routing optimization is infeasible in this instance.

For each MILP, the set of edges that are cut are found from the optimal $\mathbf{d}^*$ as follows, $C := \{(u,v) \in E \setminus E(I) \mid \mathbf{d}^{(u,v)} = 1\}$, resulting in the cut network $\mathcal{G}_{\text{cut}} = (V, E \setminus C, S, T)$. The bypass flow value is computed on the network $\mathcal{G}_{\text{byp}} := (V_{\text{byp}}, E_{\text{byp}}, S, T)$, where $V_{\text{byp}} := V \setminus I$, and $E_{\text{byp}} := E \setminus (E(I) \cup C)$. A strictly positive bypass flow value

indicates the existence of a $\text{Path}(S, T)$ on $\mathcal{G}_{\text{cut}}$ that does not visit an intermediate node in I .

Theorem 1. For each MILP, the optimal cuts C result in a bypass flow value of 0.

Proof. The partition constraints (c4) and (c5) partition the set of vertices $V \setminus I$ into two groups: nodes with potential $\mu = 0$ (e.g., T) and nodes with potential $\mu = 1$ (e.g., S). On any path $v_0 \dots v_k$ on $\mathcal{G}_{\text{byp}}$, where $v_0 = S$ and $v_k = T$, the difference in potential values can be expressed as a telescoping sum: $\sum_{i=0}^{k-1} (\mu^i - \mu^{i+1}) = \mu^S - \mu^T$. Then, by partition constraints (c4) and (c5),

$$\sum_{i=0}^{k-1} d^{(v_i, v_{i+1})} \geq \sum_{i=0}^{k-1} (\mu^i - \mu^{i+1}) = \mu^S - \mu^T \geq 1.$$

Therefore, for at least one edge (v_i, v_{i+1}) on the path, where $0 \leq i \leq k-1$, the corresponding cut value is $d^{(v_i, v_{i+1})} = 1$. These edges belong to the set of cut edges C . Thus, the flow value on $\mathcal{G}_{\text{byp}}$ is zero. $\square$

Theorem 2. For each MILP, the optimal cuts C are such that there always exists a path to the goal from the system's perspective.

Proof. First, consider the MILP in the reactive setting. The optimal cuts C satisfy the feasibility constraints (c6), (c7), and (c8). These constraints ensure that for each history variable $q \in \mathcal{B}_{\pi}.Q$, there exists a path for the system from each state $s \in S_{G_{\text{sys}}}(q)$ to T_{sys} on G_{sys} . The edge cuts C are grouped by their history variable (see equation (15)) and mapped to the corresponding $\mathcal{G}_{\text{sys}}^{(q,s)}$ (see equation (c7)). Then, each copy $\mathcal{G}_{\text{sys}}^{(q,s)}$ represents all the cuts that can be simultaneously applied when the state of the test execution is at history variable q . Thus, all restrictions on system actions at history q are captured by the cuts on $\mathcal{G}_{\text{sys}}^{(q,s)}$. Since this is true for every q and every source state s at which the test execution enters into q , there always exists a path to the goal by equation (c8). The proof for the static and mixed settings follows similarly. $\square$

Lemma 3. For each MILP, the optimal cuts C correspond to maximizing the cardinality of Θ_u .

Proof. By construction, a realization of the flow $\mathbf{f}$ on $\mathcal{G}$ corresponds to a set of unique state-history traces Θ_u . The MILP objective maximizes the flow, and therefore the cardinality of Θ_u is maximized. $\square$

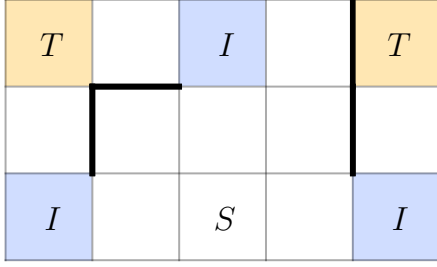
VI. TEST STRATEGY SYNTHESIS

In this section, we will outline how to find the reactive test strategy from the optimization result in the different settings.

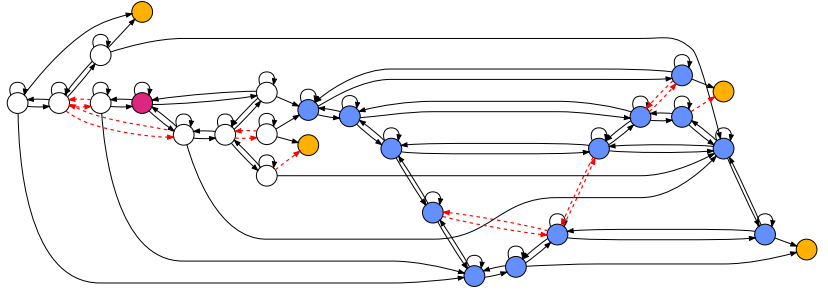
A. Test Environments with Static and/or Reactive Obstacles

For each setting (static, reactive, and mixed), the optimal cuts from solving the corresponding MILP are used to realize a test strategy with static and/or reactive obstacles. The optimal cuts C for each MILP are parsed into a reactive map $\mathcal{C} : \mathcal{B}_{\pi}.Q \rightarrow T_{\text{sys}}.E$, where

$$\mathcal{C}(q) := \{(s, s') \in T_{\text{sys}}.E \mid ((s, q), (s', q')) \in C\}. \quad (19)$$



(a) Static obstacles in black.



(b) Virtual product graph with edge cuts in dashed red.

Fig. 5: Static obstacles in (a) corresponding to edge cuts found on the virtual product graph (b) for Example 1. States marked S , I , and T illustrated in (a) correspond to states S (magenta ●), I (blue ●), and T (yellow ●) on G as shown in (b).

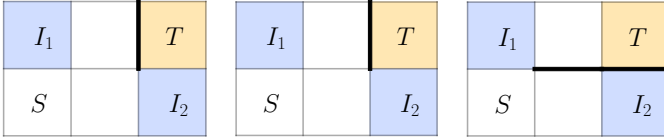
(a) q_0 (b) q_6 (c) q_7

Fig. 6: Test environment implementation of a reactive test strategy for Example 2.

The set $\mathcal{C}(q)$ captures cuts that will be used to restrict the system when the state of the test execution is at the history variable q . When the test execution ϑ reaches a state-history pair (s, q) at time step $k \geq 0$, and $\mathcal{C}(q)$ contains a system transition $(s, s') \in T_{\text{sys}}.E$, then the reactive test strategy π_{test} will restrict the system action corresponding to this transition. That is, the set of restrictions on the system is given by

$$\pi_{\text{test}}(\sigma_k) := \{a \in T_{\text{sys}}.A \mid s' \in T_{\text{sys}}.\delta(s, a) \text{ and } (s, s') \in \mathcal{C}(q)\}. \quad (20)$$

In practice, the reactive test strategy can be realized by the test environment by placing obstacles during the test execution. The set of *active obstacles* $\text{Obs}(\sigma_k)$ at time step $k \geq 0$ is defined as the set of all state-action restrictions at time k . The test environment uses the test strategy π_{test} to determine Obs in the following settings.

Instantaneous: In this setting, the test environment *instantaneously* places obstacles for the current history variable q . For any $k \geq 0$, let (s, q) be the state-history pair at time step k of the test execution. Therefore, the set of active obstacles at σ_k is given as, $\text{Obs}(\sigma_k) = \{(s', a) \mid \forall s'' \in T_{\text{sys}}.\delta(s', a) \text{ and } (s', s'') \in \mathcal{C}(q)\}$.

Accumulative: In this setting, the test environment *accumulates* obstacles according to the system state during the test execution. For any $k \geq 0$, let $(\bar{s}, \bar{q})$ and (s, q) be the state-history pairs at time steps $k-1$ and k of the test execution, respectively. If $\bar{q} \neq q$, we set active obstacles to be $\text{Obs}(\sigma_k) = \{(s, a) \mid \forall a \in \pi_{\text{test}}(\sigma_k)\}$. As the test execution progresses to state-history pair (s', q) at time step $l > k$, any transition restricted by the test strategy is added to the set of active obstacles $\text{Obs}(\sigma_l) = \bigcup_{i=k}^l \text{Obs}(\sigma_i)$ and is restricted by the test environment. These obstacles remain in place until the test execution reaches a state history pair

(s'', q') at time step $m > k$, where $q \neq q'$, at which point the test environment resets the set of active obstacles to be $\text{Obs}(\sigma_m) = \{(s'', a) \mid \forall a \in \pi_{\text{test}}(\sigma_m)\}$ and restrictions are accumulated until a different history variable is reached.

Remark 7. Assumption 1 can be relaxed if we can ensure that cuts C do not introduce any livelocks, in which the system has no path the goal. The feasibility constraints ensure that there always exists a path to the goal from every source $s \in S_{G_{\text{sys}}}(q)$ for every history variable q , and under the bidirectional setting of Assumption 1, the system can navigate back to the corresponding source. Without Assumption 1 we need to check for every cut that the MILP returns, that a path to the goal still exists. If that is not the case, we can exclude the solution and re-solve the MILP in a counterexample-guided search similar to the approach presented in Section VI-B.

Proposition 2. In both the instantaneous and accumulative settings, as long as no new restrictions that are not in $\mathcal{C}(q)$ are introduced, the flow value F remains the same.

Example 2 (Small Reactive (continued)). Fig. 6 illustrates the test environment implementing a reactive test strategy. The reactive test strategy is constructed from the optimal cuts (as depicted in Fig. 4a) on $\mathcal{G}$ found by solving **MILP (REACTIVE)**. The test starts in history variable q_0 and the system transitions are restricted according to Fig. 6a. If the system decides to visit I_1 first, the test execution moves to history variable q_6 shown in Fig. 6b, whereas if the system decides to visit I_2 first, the test execution moves to q_7 , as depicted in Fig. 6c. This test environment can be implemented in either the instantaneous or the accumulative setting.

Static and Mixed Test Environments. The cuts found from **MILP-STATIC** result in a reactive map $\mathcal{C}$ in which $\mathcal{C}(q) = \mathcal{C}(q')$ for all $q, q' \in \mathcal{B}_{\pi}.Q$. That is, restrictions on system actions remain in place for the entire duration of the test, and do not change depending on the history variable q . In this fully static setting, every edge is in the static area, that is $T_{\text{sys}}.E_{\text{static}} = T_{\text{sys}}.E$. Therefore, the test environment realizes the test strategy by restricting all system actions corresponding to any cut in $\mathcal{C}(q)$ for all $q \in \mathcal{B}_{\pi}.Q$ with static obstacles simultaneously,

$$\text{Obs} := \{(u, s, v, s) \in T_{\text{sys}}.E_{\text{static}} \mid (u, v) \in C\}. \quad (21)$$

In the mixed setting of static and reactive obstacles, the test strategy resulting from **MILP-MIXED** is implemented similarly to the reactive setting, except for system transitions in $T_{\text{sys}}.E_{\text{static}}$ that are blocked by static obstacles.

Example 1 (continued). For the grid world example, Fig. 5a illustrates the static test on the grid world, and Fig. 5b shows the corresponding cuts C^* on the virtual product network $\mathcal{G}$. Here, the 14 cuts on $\mathcal{G}$ map to 4 static obstacles since multiple edges on $\mathcal{G}$ correspond to the same transition in T_{sys} . The optimal flow value is $F^* = 3$ and there is no bypass flow. Thus, as the system navigates from source S to target T, it must visit at least one of the intermediate nodes I.

Remark 8. The instantaneous and accumulative implementations of the test environment guide when the obstacles are placed by the test environment. However, this does not have to be the same as when the system senses or observes these restrictions on its actions. We assume that the system can observe all restricted actions on its current state before it commits to an action.

The graph construction, network flow optimization, and finding the reactive test strategy are summarized in Algorithm 1.

Theorem 3. If the problem data are not inconsistent (see Section V-B), the reactive test strategy π_{test} found by Algorithm 1 solves Problem 1.

Proof. The test environment informs the choice of the MILP (static, reactive, or mixed). Therefore, the resulting π_{test} will be realizable by the test environment. By construction of G_{sys} , any correct system strategy corresponds to a $\text{Path}(S_{\text{sys}}, T_{\text{sys}})$. By Theorem 2, at any point during the test execution, if the system has not violated its guarantees, there exists a path on G_{sys} to T_{sys} . Therefore, there exists a correct system strategy π_{sys} , and resulting trace $\sigma(\pi_{\text{sys}} \times \pi_{\text{test}})$, which corresponds to the path $\vartheta_{\text{sys},n} = (s, q)_0 \dots (s, q)_n$ on G_{sys} , where $(s, q)_0 \in S_{\text{sys}}$ to $(s, q)_n \in T_{\text{sys}}$. By Lemma 1 any $\text{Path}(S_{\text{sys}}, T_{\text{sys}})$ on G_{sys} has a corresponding $\text{Path}(S, T)$ on G and by Theorem 1, the cuts ensure that all such paths on G are routed through the intermediate I. Therefore, for a correct system strategy π_{sys} , the trace $\sigma(\pi_{\text{sys}} \times \pi_{\text{test}}) \models \varphi_{\text{sys}} \wedge \varphi_{\text{test}}$. Thus, π_{test} is feasible and by Proposition 2 and Lemma 3, π_{test} is least-restrictive. Thus, Problem 1 is solved. $\square$

This framework results in a test that is not impossible (with respect to the system objective) for a correctly implemented system. On the other hand, a poorly designed system can still fail since the system is not aided in satisfying its guarantees.

B. Synthesizing a Dynamic Test Strategy

In some test scenarios, it might be beneficial to make use of an available dynamic test agent. Thus, the challenge is to find a test agent strategy that corresponds to $\mathcal{C}$ while ensuring that the system's operational environment assumptions are satisfied. To accomplish this, we adapt the **MILP-MIXED** using information about the dynamic test agent. Then, we find

Algorithm 2: Reactive Test Synthesis

```

1: procedure TEST SYNTHESIS( $T_{\text{sys}}, T_{\text{TA}}, H, \varphi_{\text{sys}}, \varphi_{\text{test}}$ )
   Input: system  $T_{\text{sys}}$ , test agent  $T_{\text{TA}}$ , test harness  $H$ ,
   system objective  $\varphi_{\text{sys}}$ , test objective  $\varphi_{\text{test}}$ 
   Output: test agent strategy  $\pi_{\text{TA}}$ 
2:  $T_{\text{sys}}.E_{\text{static}} \leftarrow \text{Define from } T_{\text{sys}}, T_{\text{TA}}$   $\triangleright$  Static area (Eq. (22))
3:  $\mathcal{G}, \mathcal{O}, \mathbf{I}, G \leftarrow \text{Setup arguments}$   $\triangleright$  Lines 2-13 in Alg. 1
4:  $C_{\text{ex}} \leftarrow \emptyset$   $\triangleright$  Initialize empty set of excluded solutions
5: while True do
6:    $\mathbf{d}^* \leftarrow \text{Solve MILP-AGENT}(\mathcal{G}, \mathcal{O}, \mathbf{I}, T_{\text{sys}}, H, C_{\text{ex}})$ 
7:   if STATUS(MILP) = infeasible then
8:     return infeasible
9:    $C \leftarrow \{(u, v) \in G.E \mid \mathbf{d}^{*(u,v)} = 1\}$   $\triangleright$  Cuts on  $G$ 
10:   $\text{Obs} \leftarrow \text{Define from } C \triangleright$  Static Obstacles (Eq. (21))
11:   $\mathcal{R} \leftarrow \text{Define from } C \triangleright$  Reactive map (Eq. (23))
12:   $\mathbf{A} \leftarrow \text{Assumptions (a1)–(a5) from } T_{\text{sys}}, T_{\text{TA}}, G,$ 
    $\varphi_{\text{sys}}$ 
13:   $\mathbf{G} \leftarrow \text{Guarantees (g1)–(g7) from } T_{\text{sys}}, T_{\text{TA}}, \mathcal{R}$ 
14:   $\varphi \leftarrow (\mathbf{A} \rightarrow \mathbf{G})$   $\triangleright$  Construct GR(1) formula
15:  if REALIZABLE( $\varphi$ ) then
16:     $\pi_{\text{TA}} \leftarrow \text{GR1Solve}(\varphi)$ 
17:    return  $\pi_{\text{TA}}, \text{Obs}$ 
18:   $C_{\text{ex}} \leftarrow C_{\text{ex}} \cup C$ 

```

the test agent strategy using reactive synthesis and counterexample guided search. From the optimal cuts of **MILP-MIXED** and the resulting reactive map $\mathcal{C}$, we can find states that the test agent must occupy in reaction to the system state. Then, we synthesize a strategy for the dynamic test agent using the Temporal Logic and Planning Toolbox (TuLiP) [57]. If we cannot synthesize a strategy, we use a counterexample-guided approach to exclude the current solution and resolve the MILP to return a different set of optimal cuts until a strategy can be synthesized. Suppose we are given a test agent whose dynamics are given by the finite transition system T_{TA} , where $T_{\text{TA}}.S$ contains at least one state that is not in $T_{\text{sys}}.S$, denoted as park. During the test execution, the test agent can navigate to these park states, if necessary. These states are required to synthesize a test agent strategy. From the test agent's transition system T_{TA} , we determine which states in T_{sys} the test agent can occupy. From these states, we can define the static area as,

$$T_{\text{sys}}.E_{\text{static}} := \{(u, v) \in T_{\text{sys}}.E \mid v \notin T_{\text{TA}}.S\}. \quad (22)$$

Adapting the MILP: Since an agent can only occupy a single state at a time, we incentivize solutions in which multiple edge cuts can be realized by occupying the same state. For this, we introduce the variable $\mathbf{d}_{\text{state}} \in \mathbb{R}_+^{|V|}$, which represents whether an incoming edge into a state is cut. This is captured by the constraint

$$\forall (u, v) \in E, \quad \mathbf{d}^{(u,v)} \leq \mathbf{d}_{\text{state}}^v, \quad (\text{c10})$$

where $\mathbf{d}_{\text{state}}^v \geq 1$ corresponds to at least one incoming edge

being cut. The adapted objective is then defined as

$$F - \frac{1}{|E|} \sum_{v \in V} d_{\text{state}}^v - \frac{1}{|E|^2} \sum_{e \in E} d^e.$$

The objective is chosen such that the number of states that need to be blocked is minimized with the fewest possible edge cuts. The regularizers are chosen to reflect this order of priority. The optimal cuts from the resulting MILP are used to synthesize a reactive test agent strategy as follows. From the optimal cuts C , we find the set of static obstacles $\text{Obs} \subseteq T_{\text{sys}} \cdot E_{\text{static}}$ according to Eq. (21) and the reactive map $\mathcal{R} : \mathcal{B}_\pi \cdot Q \rightarrow T_{\text{sys}} \cdot E$ as follows:

$$\mathcal{R}(q) := \{(s, s') \in T_{\text{sys}} \cdot E \mid (s, s') \notin T_{\text{sys}} \cdot E_{\text{static}} \text{ and } ((s, q), (s', q')) \in C\}. \quad (23)$$

The reactive map $\mathcal{R}$ is used to synthesize a strategy for the test agent. If no strategy can be found, a counter-example guided approach is used to resolve the MILP.

Reactive Synthesis: From the solution of the MILP, we now construct the specification to synthesize the test agent strategy using TuLiP. In particular, we construct a GR(1) formula with assumptions being our model of the system and the guarantees capturing requirements on the test agent. Note that we are synthesizing a strategy for the test agent, where the environment is the system under test. The variables needed to define the GR(1) formula consist of variables capturing the system's state $\mathbf{x}_{\text{sys}} \in T_{\text{sys}} \cdot S$ and $\mathbf{q}_{\text{hist}} \in \mathcal{B}_\pi \cdot Q$, which track how system transitions affect the history variable q . The test agent state is represented in the variable $\mathbf{x}_{\text{TA}} \in T_{\text{TA}} \cdot S$.

First, we set up the subformulae constituting the assumptions on the system model. The initial conditions of the system are defined as

$$(\mathbf{x}_{\text{sys}} = s_0 \wedge \mathbf{q}_{\text{hist}} = q_0), \quad (a1)$$

where $s_0 \in T_{\text{sys}} \cdot S_0$ and $\mathcal{B}_\pi \cdot Q_0$. We define the dynamics of the system and the history variable for each state $(s, q) \in G \cdot S$ as follows:

$$\Box \left((\mathbf{x}_{\text{sys}} = s \wedge \mathbf{q}_{\text{hist}} = q) \rightarrow \bigvee_{\substack{(s', q') \in \\ \text{succ}(s, q)}} \bigcirc (\mathbf{x}_{\text{sys}} = s' \wedge \mathbf{q}_{\text{hist}} = q') \right), \quad (a2)$$

where $\text{succ}(s, q)$ denotes the successors of state $(s, q) \in G \cdot S$. For simplicity, we choose a turn-based setting, in which each player will only take their action if it is their turn. To track this, we introduce the variable $\text{turn} \in \mathbb{B}$ as a test agent variable. For the system, this is encoded as remaining in place when $\text{turn} = 1$:

$$\bigwedge_{s \in T_{\text{sys}} \cdot S} \Box \left((\mathbf{x}_{\text{sys}} = s \wedge \text{turn} = 1) \rightarrow \bigcirc (\mathbf{x}_{\text{sys}} = s) \right). \quad (a3)$$

If a turn-based setup is not used, we need to synthesize a Moore strategy for the test agent since it should account for all possible system actions. The system objective φ_{sys} can be encoded as the formula

$$\Box \Diamond (\mathbf{x}_{\text{sys}} = x_{\text{goal}}) \wedge \varphi_{\text{aux}}, \quad (a4)$$

where x_{goal} is the terminal state of the system and a reachability objective specified in φ_{sys} . The other objectives specified

in φ_{sys} are transformed to their respective GR(1) forms in φ_{aux} . This transformation of LTL formulae into GR(1) form is detailed in [58]. In addition, the system is expected to safely operate in the test agent's presence. The set of states where collision is possible is denoted by $S_\cap := T_{\text{sys}} \cdot S \cap T_{\text{TA}} \cdot S$. Thus, the safety formula encoding that the system will not collide into the tester is given as:

$$\bigwedge_{s \in S_\cap} \Box \left(\mathbf{x}_{\text{TA}} = s \rightarrow \bigcirc \neg (\mathbf{x}_{\text{sys}} = s) \right). \quad (a5)$$

Equations (a1)–(a5) represent the test agent's assumptions on the system model. Next, we describe the subformulae for the guarantees of the GR(1) specification. The initial conditions for the test agent are

$$\bigvee_{s \in T_{\text{TA}} \cdot S_0} \mathbf{x}_{\text{TA}} = s. \quad (g1)$$

The test agent dynamics are represented by

$$\Box \left((\mathbf{x}_{\text{TA}} = s) \rightarrow \bigvee_{(s, s') \in T_{\text{TA}} \cdot E} \bigcirc (\mathbf{x}_{\text{TA}} = s') \right). \quad (g2)$$

The test agent can also move only in its turn and will remain stationary when $\text{turn} = 0$:

$$\bigwedge_{s \in T_{\text{TA}} \cdot S} \Box \left((\mathbf{x}_{\text{TA}} = s \wedge \text{turn} = 0) \rightarrow \bigcirc (\mathbf{x}_{\text{TA}} = s) \right). \quad (g3)$$

The turn variable alternates at each step:

$$\begin{aligned} (\text{turn} = 1) &\rightarrow \bigcirc (\text{turn} = 0) \wedge \\ (\text{turn} = 0) &\rightarrow \bigcirc (\text{turn} = 1). \end{aligned} \quad (g4)$$

To satisfy the system assumptions (Def. 10), the test agent should not adversarially collide into the system. This is captured via the following safety formula,

$$\bigwedge_{s \in S_\cap} \Box \left(\mathbf{x}_{\text{sys}} = s \rightarrow \bigcirc \neg (\mathbf{x}_{\text{TA}} = s) \right). \quad (g5)$$

Now, we enforce the optimal cuts found from the MILP. To enforce cuts reactively during the test execution, the states occupied by the system are defined as follows,

$$\begin{aligned} \bigwedge_{q \in \mathcal{B}_\pi \cdot Q} \bigwedge_{(s, s') \in \mathcal{R}(q)} \Box \left((\mathbf{x}_{\text{sys}} = s \wedge \mathbf{q}_{\text{hist}} = q \wedge \text{turn} = 0) \right. \\ \left. \rightarrow (\mathbf{x}_{\text{TA}} = s') \right). \end{aligned} \quad (g6)$$

Essentially, for some history variable q , if $(s, s') \in \mathcal{R}(q)$ is an edge cut, then the test agent must occupy the state s' when the system is in the state s when the test execution is at history variable q . However, the test agent should not introduce any additional restrictions on the system, which is formulated as

$$\begin{aligned} \bigwedge_{q \in \mathcal{B}_\pi \cdot Q} \bigwedge_{\substack{(s, s') \in T_{\text{sys}} \cdot E \\ (s, s') \notin \mathcal{R}(q)}} \Box \left((\mathbf{x}_{\text{sys}} = s \wedge \mathbf{q}_{\text{hist}} = q \wedge \text{turn} = 0) \right. \\ \left. \rightarrow \neg (\mathbf{x}_{\text{TA}} = s') \right). \end{aligned} \quad (g7)$$

Intuitively, this corresponds to the requirement that the tester agent shall not restrict system transitions that are not part of the

reactive map $\mathcal{R}$. A test agent strategy that satisfies the above specifications is guaranteed to not restrict any system action unnecessarily. However, the test agent can occupy a state that is not adjacent to the system and block all paths to the goal from the system's perspective. This could lead the system to not making any progress towards the goal at all, resulting in a livelock. To avoid this, we characterize the livelock condition as a safety constraint that the test agent must satisfy (e.g., if it occupies a livelock state, it must not occupy it in the next step). The specific safety formula that captures the livelock depends on the example. We find the states where the tester would block the system from reaching its goal $T_{\text{sys}} \cdot S_{\text{block}} \subseteq T_{\text{TA}} \cdot S$. The following condition ensures that it will only transiently occupy blocking states:

$$\bigwedge_{s \in T_{\text{sys}} \cdot S_{\text{block}}} \square \left(x_{\text{TA}} = s \rightarrow \bigcirc \neg (x_{\text{TA}} = s) \right). \quad (\text{g8})$$

Therefore, we synthesize a test agent strategy π_{TA} for the GR(1) formula with assumptions (a1)–(a5) and guarantees (g1)–(g8).

Counterexample-guided Approach: The MILP can have multiple optimal solutions, some of which may not be realizable for the test agent. If the GR(1) formula is unrealizable, we exclude the solution and re-solve the MILP until we find a realizable GR(1) formula. In particular, every new set of optimal cuts C that is unrealizable is added to the set $\mathcal{C}_{\text{ex}}$. Then, the MILP is resolved with an additional set of affine constraints as follows,

$$\sum_{e \in C} d^e \leq |C| - 1, \forall C \in \mathcal{C}_{\text{ex}}. \quad (\text{c15})$$

This corresponds to preventing all edges in an excluded solution C from being cut at the same time. The adapted MILP is then defined as follows:

MILP-AGENT:

$$\begin{aligned} \max_{\substack{\mathbf{f}, \mathbf{d}, \mathbf{d}_{\text{state}}, \boldsymbol{\mu}, \\ \mathbf{f}_{\text{sys}}^{(q,s)} \forall q \in \mathcal{B}_{\pi} \cdot Q, \\ \forall s \in \mathcal{S}_{\mathcal{G}_{\text{sys}}}(q)}}} & F - \frac{1}{|E|} \sum_{v \in V} d_{\text{state}}^v - \frac{1}{|E|^2} \sum_{e \in E} d^e \\ \text{s.t.} & (\text{c1})\text{--}(\text{c9}), (\text{c10}), (\text{c15}). \end{aligned} \quad (24)$$

This process is repeated until a strategy is synthesized or the **MILP-AGENT** becomes infeasible.

Lemma 4. Let π_{TA} be the test agent strategy and let Obs be the set of static obstacles synthesized from the optimal solution C^* of **MILP-AGENT** according to the GR(1) formula with assumptions (a1)–(a5) and guarantees (g1)–(g8). Let π_{test} be the reactive test strategy corresponding to the optimal cuts C^* . Then π_{TA} and Obs realize π_{test} .

Proof. By construction in Eqs. (19), (21), (23), we have that $\mathcal{C}(q) = \mathcal{R}(q) \cup \text{Obs}$ for all history variables $q \in \mathcal{B}_{\pi} \cdot Q$. Due to guarantee (g6), the synthesized test agent strategy restricts the transitions in $\mathcal{R}(q)$. The test agent is also prohibited from restricting any other transitions by the guarantee (g7). Therefore, at each step of the test execution, the system actions

restricted as a result of π_{TA} and static obstacles Obs exactly correspond to those restricted by the test strategy π_{test} . $\square$

Theorem 4. Algorithm 2 solves Problem 2.

Proof. The test agent strategy is synthesized to satisfy guarantees (g1)–(g8). The guarantees (g1)–(g4) specify the dynamics of the test agent, which satisfies **A1**. The safety guarantee (g5) satisfies **A2**. Guarantees (g6) and (g7) realize the optimal cuts from **MILP-AGENT**. Due to constraint (c8) the optimal cuts ensure that there always exists a path on $\mathcal{G}_{\text{sys}}$. Together with guarantee (g8), this results in π_{TA} satisfying assumptions **A3** and **A4**. By Lemma 4, π_{TA} is a realization of a least-restrictive feasible π_{test} . $\square$

The test agent strategy and obstacles, π_{TA} and Obs correspond to the least-restrictive reactive test strategy π_{test} possible for that test environment. Other test environments might result in different least-restrictive reactive test strategies.

VII. COMPLEXITY

Our framework comprises three parts: i) graph construction, ii) routing optimization, and iii) reactive synthesis. For graph construction, we first need to construct Büchi automata from specifications. In the worst case, this construction has doubly-exponential complexity, $2^{2^{|\phi|}}$, in the length of the formula ϕ [36]. Then, graph construction involves computing a Cartesian product of two graphs T_{sys} and $\mathcal{B}_{\pi}$, and has a worst-case time complexity of $O(|T_{\text{sys}} \cdot S|^2 \cdot |\mathcal{B}_{\pi} \cdot Q|^2)$. For a more efficient implementation, we construct this product by expanding into states that are reachable from the source S . In the reactive synthesis part of the framework, we use GR(1) synthesis which is known to have a complexity of $O(|N|)^3$, where N is the number of states required to define the formula. In this section, we will establish the computational complexity of the routing optimization and show that the associated decision problem is NP-hard.

To prove the computational complexity of finding the cuts on the graph, we first prove the computational complexity in the special case of static obstacles. As defined in sections IV and V, the problem data is a graph $G = (V, E)$ with node groups S, I, T , and the corresponding flow network $\mathcal{G}$. For some edge $e \in E \setminus E(I)$, the binary variable d^e indicates whether the edge is cut: $d^e = 1$. The set $C \subseteq E$ represents the set of edges with $d^e = 1$. For static obstacles, the edges are grouped by the corresponding transition in T_{sys} . The grouping $\text{Gr}_{\text{static}} : T_{\text{sys}} \cdot E \rightarrow G \cdot E$, and defined as follows,

$$\text{Gr}_{\text{static}}((s, s')) := \{(u, v) \in G \cdot E \mid u.s = s, v.s = s'\}. \quad (25)$$

For some $(s, s') \in T_{\text{sys}} \cdot E$, all edges $e \in \text{Gr}_{\text{static}}((s, s'))$ have the same d^e value, i.e., if $d^e = 1$ for some edge e in the group, then all edges in this group will have d^e set to 1. A bypass path on G is some $\text{Path}(S, T)$ which does not visit the intermediate I . The flow value F on $\mathcal{G}$ is defined from the source S to target T , with each edge having unit capacity. A valid set of edge cuts C is such that i) there does not exist a bypass path, ii) there exists a path from S to T , and iii) edges respect the grouping $\text{Gr}_{\text{static}}$.

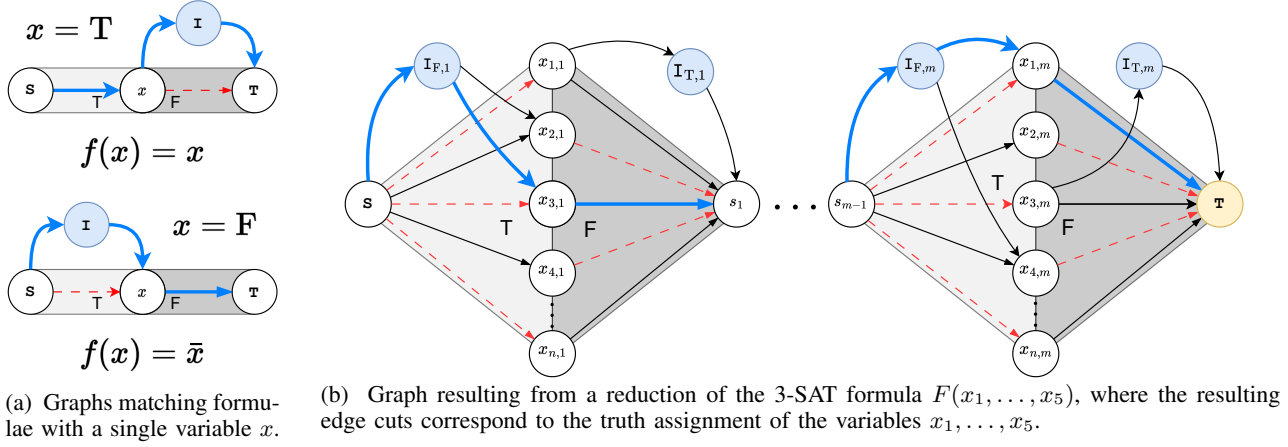


Fig. 7: Graphs constructed from a 3-SAT formula, where a truth assignment for the variables can be found using the network flow approach for static obstacles.

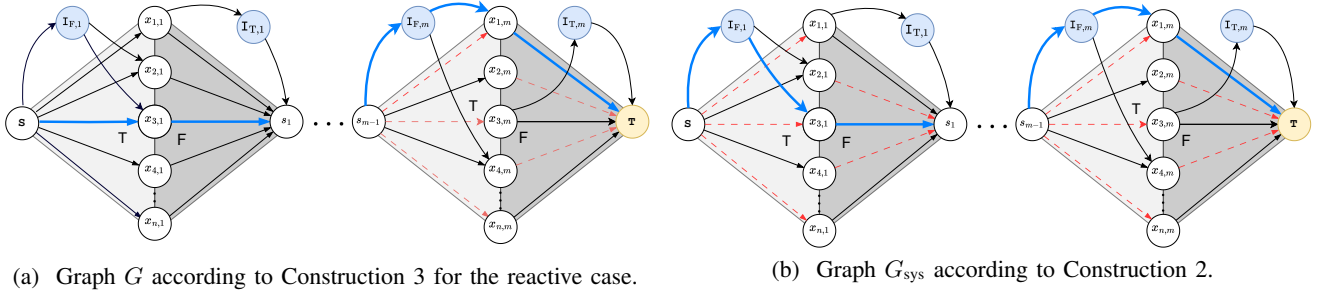


Fig. 8: Graphs G and G_{sys} constructed from a 3-SAT formula, where a truth assignment for the variables can be found using the flow approach for reactive obstacles.

Problem 3 (Static Obstacles Optimization Problem). Given a graph G , find a valid set of edge cuts C such that the resulting maximum flow F is maximized over all possible sets of edge cuts, and such that $|C|$ is minimized for the flow F .

This corresponds to finding the valid set of edge cuts C that as first priority, maximizes the flow F , and subsequently chooses the set of edge cuts C with the smallest cardinality $|C|$ (i.e. breaking ties between all valid edge cuts that realize F). For static obstacles, Problem 3 corresponds to the following decision problem.

Problem 4 (Static Obstacles Decision Problem). Given a graph G and an integer $M \geq 0$, does there exist a valid set of edge cuts C such that $|C| \leq M$?

Lemma 5. Any solution to Problem 3 can be used to construct a solution for Problem 4 in polynomial time.

Lemma 5 implies that if there exists a polynomial-time algorithm to compute a solution to Problem 3, then there also exists a polynomial-time algorithm to solve Problem 4. Thus, if we can show that Problem 4 belongs to the class of NP-hard problems (i.e., there exists a polynomial-time reduction from any arbitrary problem in NP to Problem 4), that would imply that there exists a polynomial-time algorithm to solve Problem 4 only if $P = NP$. This in turn would support the MILP approach we provide to solve Problem 3. To show

that Problem 4 is NP-hard, we construct a polynomial-time reduction from 3-SAT to Problem 4. This polynomial-time reduction maps any instance of 3-SAT to Problem 4 such that the solution of the constructed instance of Problem 4 corresponds to a solution of the instance of the 3-SAT problem.

Definition 20 (3-SAT [59]). Let $f(x_1, \dots, x_n) := \bigwedge_{j=1}^m c_j$ be a propositional logic formula over Boolean propositions $x_1, \dots, x_n$ in conjunctive normal form (CNF) in which each clause c_j is a disjunction of three Boolean propositions or their negations. A solution to the 3-SAT problem is an algorithm that returns *True* if there exists a satisfying Boolean assignment to $f(x_1, \dots, x_n)$ and *False* otherwise.

We first introduce a construction which maps any clause in a propositional logic formula to some sub-graph of a graph. We will then connect these sub-graphs to obtain the graph which will allow a reduction of any instance of 3-SAT to Problem 4. In turn, we will show that we can use any algorithm that solves Problem 4 to solve the 3-SAT problem, showing that Problem 4 is polynomial-time only if there exists a polynomial-time algorithm to solve 3-SAT, implying $P=NP$.

Construction 1 (Clause to Sub-graph). Given a 3-SAT clause c_j , we can construct a sub-graph representing this clause as follows. For each clause c_j , we introduce nodes s_{j-1} and s_j . Then, we add the nodes $x_{1,j}, \dots, x_{n,j}$ corresponding

to variables $x_1, \dots, x_n$ in the 3-SAT formula. We add the following directed edges for each $x_{i,j}$ node — an incoming edge from node s_{j-1} to $x_{i,j}$, and an outgoing edge from $x_{i,j}$ node to node s_j . Then we add two nodes, denoted by $I_{T,j}$ and $I_{F,j}$, to this sub-graph. If x_i appears in the clause c_j , then we connect the $I_{T,j}$ node by bypassing the edge from $x_{i,j}$ to s_j , and if $\bar{x}_i$ appears in c_j , then we connect $I_{F,j}$ to bypass the edge from s_{j-1} to $x_{i,j}$ (as shown in Fig. 7a).

Constructing a sub-graph for a clause c_j via Construction 1 allows us to relate the edge cuts to the Boolean assignment for the variables $x_0, \dots, x_n$. If the incoming edge into $x_{i,j}$ is cut, then the corresponding Boolean assignment to x_i is *False*, and if the outgoing edge from $x_{i,j}$ is cut, then the corresponding Boolean assignment to x_i is *True*. This ensures that a satisfying assignment for the clause corresponds to edge cuts such that all $Paths(s_{j-1}, s_j)$ are routed through intermediate nodes $\{I_{T,j}, I_{F,j}\}$. An assignment that evaluates the clause c_j to *False* corresponds to edge cuts in the sub-graph such that there is no path from s_{j-1} to s_j .

Construction 2 (Reduction of 3-SAT to Problem 4). Suppose we have an instance of the 3-SAT problem with n variables $x_1, \dots, x_n$ and m clauses $c_1, \dots, c_m$. First, we construct the sub-graphs for each clause according to Construction 1. Let $M := m \times n$. We denote the node s_0 as the source S, and s_m as the sink T. The resulting graph is a series of sub-graphs representing each clause c_j of the 3-SAT formula. For every variable x_i in the formula, we maintain two groups of edges: i) incoming edges $\{(s_{j-1}, x_{i,j}) \mid 1 \leq j \leq m\}$, and ii) outgoing edges $\{(x_{i,j}, s_j) \mid 1 \leq j \leq m\}$. All edges in a group share the same edge cut value, corresponding to Gr_{static} . This ensures that every variable has the same Boolean assignment across clauses.

This allows us to construct a graph corresponding to a 3-SAT formula in polynomial time via the procedure outlined in Construction 2, also illustrated in Fig. 7.

Theorem 5. Problem 4 is NP-complete.

Proof. We will show that Problem 4 is NP-hard by showing that Construction 2 is a correct polynomial-time reduction of the 3-SAT problem to Problem 4 i.e., any polynomial-time algorithm to solve Problem 4 can be used to solve 3-SAT in polynomial-time. Consider the graph constructed by Construction 2 for any propositional logic formula. The valid set of edge cuts C on this graph with cardinality $|C| \leq M$ is a witness for Problem 4. A witness for the 3-SAT formula is an assignment of the variables $x_1, \dots, x_n$. A witness to a problem is *satisfying* if the problem evaluates to *True* under that witness. Next, we show that a valid set of edge cuts C is a satisfying witness for Problem 4 iff the corresponding assignment to variables $x_1, \dots, x_n$ is a satisfying witness for the 3-SAT formula.

First, consider a satisfying witness for Problem 4. By Construction 2, the cardinality of the witness, $|C| = m \times n$ will be exactly M , which is the minimum number of edge cuts required to ensure no bypass paths on the constructed graph. This implies that each variable x_i has a Boolean assignment.

By Construction 1, a strictly positive flow on the sub-graph of clause c_j implies that c_j is satisfied. By Construction 2, a strictly positive flow through the entire graph implies that all clauses in the 3-SAT formula are satisfied. Therefore, a satisfying witness to the 3-SAT formula can be constructed in polynomial-time from a satisfying witness for an instance of Problem 4.

Next, we consider a satisfying witness for the 3-SAT formula. The Boolean assignment for each variable x_i corresponds to edge cuts on the graph (see Fig. 7b). Any Boolean assignment ensures that there is no bypass path on the graph since either all incoming edges or all outgoing edges for each variable x_i are cut. This also corresponds to the minimum number of edge cuts required to cut all bypass paths, corresponding to $|C| = m \times n$. By Construction 1, a satisfying witness corresponds to a $Path(s_{j-1}, s_j)$ on the sub-graph for each clause c_j . By Construction 2, observe that there exists a strictly positive flow on the graph. Thus, we can construct a satisfying witness to an instance of Problem 4 in polynomial time from a satisfying witness to the 3-SAT formula. Therefore, any 3-SAT problem reduces to an instance of Problem 4, and thus, Problem 4 is NP-hard. Additionally, Problem 4 is NP-complete since we can check the cardinality of C , and whether C is a valid set of edge cuts in polynomial time. $\square$

Corollary 1. Problem 3 is NP-hard [60].

Proof. By Theorem 5, Problem 4 is NP-complete, and therefore by Lemma 5, Problem 3 is NP-hard. $\square$

Additionally, we can identify the computational complexity for the reactive setting. For the reactive setting, a valid set of edge cuts is similar to the static setting, except in how edges are grouped, which is discussed in Remark 6. Fig. 8 illustrates the graphs used for establishing the computational complexity in this setting. The optimization problem and its corresponding decision problem can be stated as follows.

Problem 5 (Reactive Obstacles Optimization Problem). Given a graph G , identify a valid set of edge cuts C such that the resulting flow F is maximized over all possible sets of edge cuts, and such that $|C|$ is minimized for the flow F .

Note that a *valid* set of edge cuts for the reactive problem is different from a valid set of edge cuts for the static problem.

Problem 6 (Reactive Obstacles Decision Problem). Given a graph G , and an integer $M \geq 0$, does there exist a valid set of cuts C such that $|C| \leq M$?

Once again, we prove a reduction from 3-SAT, but to an instance of Problem 6 with a single history variable q . Given a 3-SAT formula, the construction of the graph follows from the static setting, but with a few key differences.

Construction 3 (Reduction from 3-SAT to Problem 6 with single history variable q). Suppose we have an instance of the 3-SAT problem with n variables $x_1, \dots, x_n$ and m clauses $c_1, \dots, c_m$. Let $M := n$. Using Construction 2, setup two graphs: G and a copy $G^{(q,S)}$ for source S and the single history variable q . The key difference is that $G^{(q,S)}$ follows

Construction 2 exactly, while in G , edges in a group need not have the same cut value. Furthermore, for each group in $G^{(q,s)}$, the cut value is set to the maximum edge-cut value in the corresponding group in G .

Theorem 6. Problem 6 is NP-complete and Problem 5 is NP-hard.

Proof. The proof follows similarly from Theorem 5. In this setting, a witness for Problem 6 comprises the maximum edge cut value of each group in G . Construction 3 relates edge cuts on G and $G^{(q,s)}$. This implies that edge cuts on G are found under the condition that there is a strictly positive flow on $G^{(q,s)}$ under a static mapping of edges. The minimum set of edge cuts which ensures no bypass paths on G has cardinality n , corresponding to only one of the sub-graphs having edge cuts. Furthermore, for each x_i , there will be one edge-cut in one of the two groups (incoming or outgoing edges). Therefore, for each x_i , only the incoming or the outgoing edge group will have a maximum edge cut value of 1, corresponding to the Boolean assignment for x_i . A minimum cut on G found under the conditions of no bypass paths on G and a positive flow on $G^{(q,s)}$ results in a Boolean assignment that is a satisfying witness to the 3-SAT formula. Thus, we have polynomial-time construction of a satisfying witness to the 3-SAT formula from a satisfying witness to Problem 6. This follows similarly to Theorem 5.

Likewise, a satisfying witness to the 3-SAT formula can be mapped to edge cuts on one of the sub-graphs of G . These edge cuts will be such that there is no bypass path on G , and will be the minimum set of edge cuts to accomplish this task, corresponding to $|C| = n$. Additionally, by construction of the graphs, this will correspond to a strictly positive flow on $G^{(q,s)}$. Thus, we can construct a satisfying witness to Problem 6 in polynomial time from a satisfying witness of the 3-SAT formula. Therefore, any 3-SAT problem reduces to an instance of Problem 6. As a result, Problem 6 is NP-complete and following similarly to Corollary (1), Problem 5 is NP-hard. $\square$

VIII. EXPERIMENTS

In this section, we demonstrate our framework on simulated and hardware experiments, and include runtime analysis. In the following experiments, examples with static test environments solve the routing optimization **MILP-STATIC** to find the test strategy. Similarly, examples with reactive test environments solve **MILP-REACTIVE**, and those with reactive dynamic agents solve **MILP-AGENT**, unless otherwise stated. These optimizations are solved using Gurobi [61]. The reactive test agent strategies are synthesized using the temporal logic planning toolbox TuLiP [47].

In simulations and hardware experiments, we utilize Unitree A1 quadrupeds as both the system and test agents. The low-level control of the quadruped is managed through a motion primitive layer, which abstracts the underlying dynamics and facilitates transitions between primitives as described in [62]. This includes behaviors such as lying down, standing, walking, jumping, and reduced-order model-based waypoint tracking

using a unicycle or single integrator model. These behaviors can be directly commanded by the autonomy layer provided by TuLiP. Individual motion primitives are implemented within our C++ motion primitive framework, with control laws, sensing, and estimation executed at 1kHz.

A. Simulation

Reactive Test Environment: The following two reactive examples were demonstrated on hardware in previous work in [45]. The updated framework in this paper resulted in simulated test traces (see Figs. 9a, 9b) that are qualitatively similar to the hardware demo in [45]. Additionally, using the updated framework reduced the time to solve the optimization by three orders of magnitude.

1) *Beaver Rescue:* The quadruped's task is to rescue the beaver from the hallway and return it to the lab. The system objective is given as $\varphi_{\text{sys}} = \Diamond(\text{beaver} \wedge \Diamond \text{goal})$, where 'beaver' corresponds to the quadruped reaching the beaver, and 'goal' corresponds to the quadruped and the beaver reaching the safe location in the lab. The test objective is given as $\varphi_{\text{test}} = \Diamond \text{door}_1 \wedge \Diamond \text{door}_2$, ensuring that the quadruped will use different doors on the way to the beaver and back into the lab. The resulting test execution first shows the quadruped using door_2 to exit the lab into the hallway, then after it reaches the beaver, door_2 is shut and the quadruped walks to door_1 to finally return to the lab. The reactive aspect here can be observed as follows — if the quadruped chose to enter the hallway through door_1 , then the resulting test execution would constrain access to door_1 when the quadruped is attempting to re-enter the lab with the beaver. The simulated test trace is shown in Fig. 9a.

2) *Motion Primitive Example:* In this example, we test the motion primitives of the quadruped given as 'lie', 'jump', and 'stand'. The goal for the quadruped is to reach the beaver in the hallway. The test objective is given as $\varphi_{\text{test}} = \Diamond \text{jump} \wedge \Diamond \text{lie} \wedge \Diamond \text{stand}$, which ensures that each motion primitive is tested at least once; and the system objective is $\varphi_{\text{sys}} = \Diamond \text{goal}$, where 'goal' corresponds to the beaver location. The test setup includes doors that might be unlocked by the system demonstrating specific motion primitives. Our framework will decide whether the doors will be locked or unlocked according to which motion primitives have already been observed during the test. This is where the reactivity of this framework becomes apparent, if the quadruped chose a different set of doors and motion primitives, the resulting test execution would have been different. The simulated test trace is shown in Fig. 9b.

Test Environment with Dynamic Agent

3) *Maze 1:* The system (gray quadruped) wants to reach its goal location in the top left corner of the grid, and the test agent wants to route it through a series of states, labeled I_1 , I_2 , and I_3 , shown in Fig. 9c. The system specification and test objective are given as $\varphi_{\text{sys}} = \Diamond \text{goal}$ and $\varphi_{\text{test}} = \Diamond I_1 \wedge \Diamond I_2 \wedge \Diamond I_3$. The test agent (yellow quadruped) can move up on the center column of the grid, and its strategy is found using the flow-based synthesis framework. Observe that it blocks specific cells such that the quadruped cannot directly navigate to its goal through the center of the grid. Instead, the system

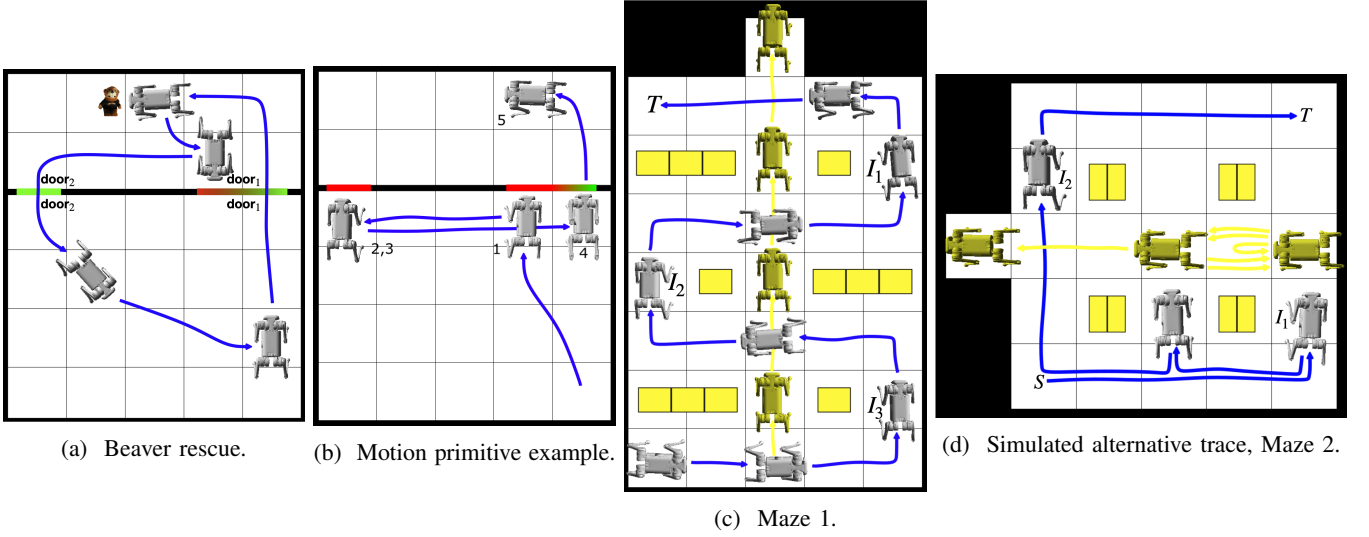


Fig. 9: Simulated experiment results. Yellow boxes are obstacles to indicate states that are not navigable in T_{sys} . Gray quadruped is the system, and yellow quadruped in (c) and (d) is the test agent. In (b), system demonstrates primitives in the order: stand (1), stand (2), jump (3), and lie (4), before advancing to goal (5). In (c) and (d), the test agent chooses to navigate off-grid after the test objective is realized.

quadruped is forced to visit the labeled cells, and only then, the test agent moves into the parking state off the grid to not excessively constrain the system. The resulting test execution is shown in Fig. 9c.

B. Hardware Experiments

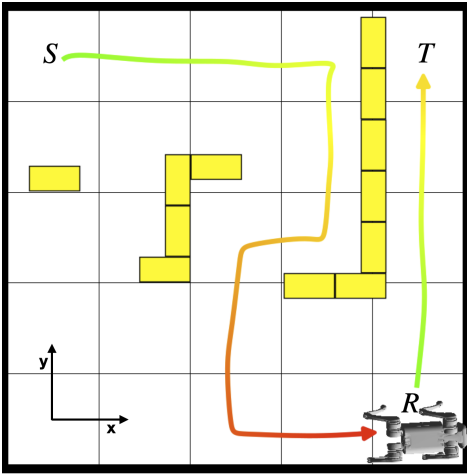


Fig. 10: Refueling example experiment trace with yellow boxes representing static obstacles Obs.

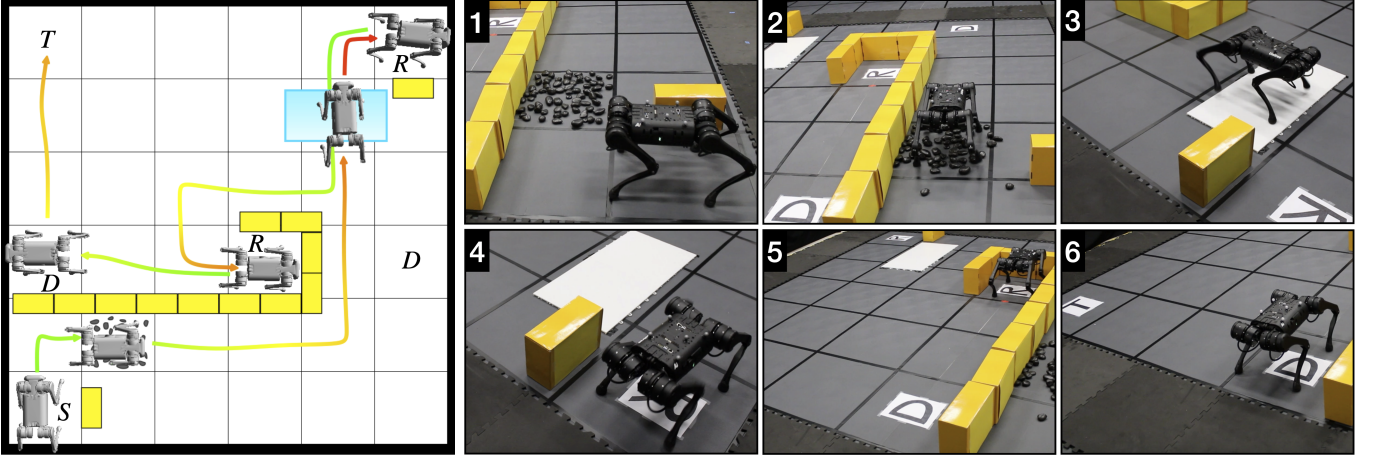
Static Test Environment:

1) *Running Example*: For this experiment we implemented Example 1 on the quadruped. The resulting test trace is shown in Fig. 12.

For the following two examples, the system state also contains the fuel level. Thus, the auxiliary bidirectional constraints in **MILP-STATIC** are such that the fuel level is abstracted away, meaning if a transition is cut for a specific fuel level, it is cut for all fuel levels.

2) *Refueling*: This example highlights that intermediate nodes need not always represent poses of the system. In addition to the coordinates $\mathbf{x} = (x, y)$, the quadruped state also tracks the fuel level f . A full fuel tank consists of 10 units of fuel. Every move on the grid reduces the fuel level by 1 and reaching the refueling station (in the bottom right corner of the grid) resets the fuel tank to full. The desired test behavior is to have the system visit a state that is too far away to reach the goal state with its available fuel, specifically we want to see the system be in the lower three rows of the grid with a fuel level of lower than 2. The system objective is given as $\varphi_{\text{sys}} = \Diamond \text{goal} \wedge \Box \neg (f = 0)$ and the test objective is $\varphi_{\text{test}} = \Diamond (y < 4 \wedge f < 2)$. Note that this test objective also includes states where the fuel tank is empty, $f = 0$, but the MILP will not route the test execution through these *unsafe* states, but will automatically only route it through the states where $f = 1$ instead. Snapshots and the trace of the test execution are shown in Fig. 10. The color of the trace corresponds to the fuel level, and we observe that the obstacle configuration is such that for the quadruped to successfully reach its goal location it is required to visit the refueling station.

3) *Mars Exploration*: In this example the system is tested for a combination of reachability, reaction and avoidance sub-tasks. This example is inspired by a planetary rover's exploration of the Martian surface. Consequently, the grid world has states designated as 'rock', 'ice', and 'drop-off', denoting sample locations and the drop-off position, respectively. In addition to the coordinates $\mathbf{x}$, the quadruped state also contains the fuel level f that decreases by 1 for every transition on the grid. The maximum fuel capacity is 10 units and is reset to full at the refueling locations labeled 'R'. The system objective states that the quadruped must reach its goal location, labeled 'T', and if it picks up a sample, it shall drop it off at the drop-off location, while not running out of fuel. This is captured in



(a) Mars exploration experiment trace.

(b) Mars exploration experiment snapshots.

Fig. 11: Resulting test execution on the Unitree A1 quadruped for static test environments.

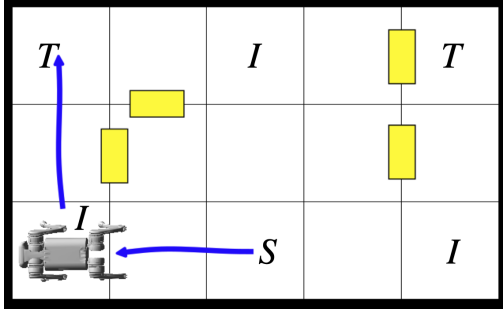


Fig. 12: Experiment trace for Example 1.

the system objective

$$\varphi_{\text{sys}} = \Diamond T \wedge \Box \neg(f = 0) \wedge \Box(\text{ice} \vee \text{rock} \rightarrow \Diamond \text{drop-off}).$$

The test objective corresponds to the triggers of the reaction sub-task. Specifically, the quadruped is required to collect a ‘rock’ sample and an ‘ice’ sample, and is routed such that a successful run requires the quadruped to refuel:

$$\varphi_{\text{test}} = \Diamond \text{rock} \wedge \Diamond \text{ice} \wedge \Diamond(d > f),$$

where $d = |\mathbf{x} - \mathbf{x}_{\text{goal}}|$ is the distance to the goal and f is the fuel level. The experiment trace and snapshots of the hardware test execution are shown in Figs. 11a and 11b. From the experiment trace, the static obstacles are placed such that the quadruped has to pick up rock and ice samples, refuel twice, and then drop off samples before reaching its goal. The test environment for the hardware run in Fig. 11 corresponds to a sub-optimal solution of **MILP-STATIC** with a flow of 1. This sub-optimal solution still ensures that the system is routed in a manner that the test objective is still satisfied. In Table V, we list the runtimes for getting the optimal solution for this example.

Reactive Dynamic Agent:

4) *Patrolling*: This example is similar to the static refueling example, except that the test environment now consists of a test agent and static obstacles (see Fig. 1). The system (gray quadruped) starts in the lower right corner and must reach

its goal in the lower left corner of the grid without running out of fuel, which is encoded in the system objective: $\varphi_{\text{sys}} = \Diamond T \wedge \Box \neg(f = 0)$. The refueling station is denoted ‘R’ in Fig. 1. Once again, the test objective routes the system through a state from which a successful test execution requires it to refuel,

$$\varphi_{\text{test}} = \Diamond(d > f),$$

where $d = |\mathbf{x} - \mathbf{x}_{\text{goal}}|$ is the distance to the goal. The test agent can move up and down the third column of the grid, and can leave the grid from the first and last rows to a parking state. As shown in the trace and hardware snapshots in Fig. 1, our framework chooses to place a static obstacle near the start state, and the test agent blocks the system from directly navigating to the goal (see panels 2, 3 and 4 in Fig. 1) until its fuel level is low enough, thus requiring it to refuel. For this experiment, we solve **MILP-AGENT** with the objective (12) for numerical stability in Gurobi.

5) *Maze 2*: In this example, the system quadruped starts in the bottom left corner of the grid, and must reach its goal location in the top right corner. The grid world is a 5×5 grid, with a symmetric obstacle configuration shown in yellow in Fig. 13a. In this example, the test environment consists of a test agent that can traverse along the center row and center column of the grid. While the test environment can also place static obstacles, it realizes the test strategy entirely via the test agent. The system objective is given as follows $\varphi_{\text{sys}} = \Diamond T$. The test objective consists of two visit tasks in arbitrary order, encoded as

$$\varphi_{\text{test}} = \Diamond I_1 \wedge \Diamond I_2,$$

where I_1 and I_2 correspond to the designated locations on the grid. The specification product is the same as shown in Fig. 3c, we can see that to route the test execution through the test objective acceptance states, we need to find cuts for the history variables q_0 , q_6 , and q_7 . The reactive cuts found by the flow-based synthesis procedure are shown in Figs. 13b-13d. The trace and snapshots of the resulting test execution is shown in Figs. 14a and 14b. We observe that the system quadruped decides to take the top path first, visits I_2 (see panel 2 in Fig. 14b), and is blocked by the test agent (see

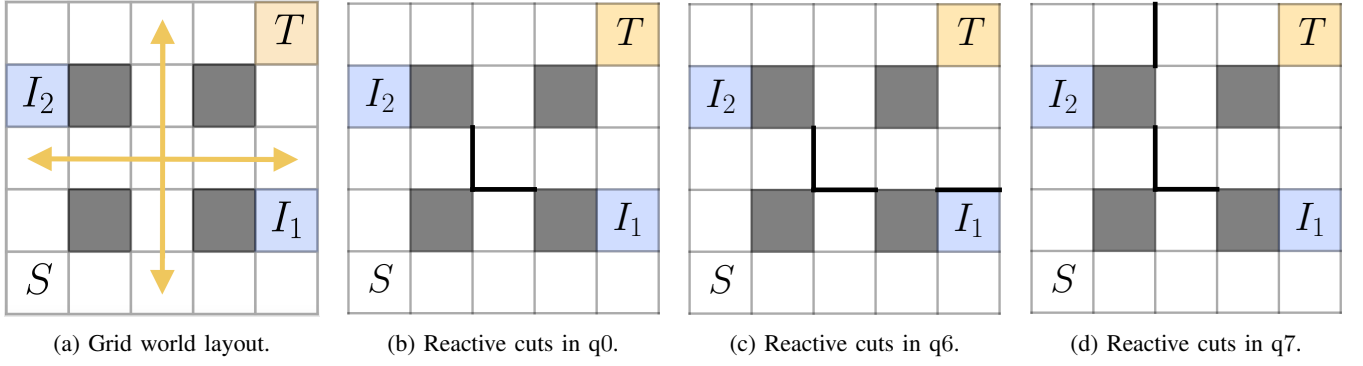


Fig. 13: (a) Grid world layout with cells traversable by the test agent marked. Dark gray cells are not traversable by either agent. (b) Black edges indicate reactive cuts corresponding to the history variables for the Maze 2 experiment. Note that the cuts are not bidirectional. The history variable states q_0 , q_6 , and q_7 can be inferred from $\mathcal{B}_\pi$ illustrated in Fig. 3c, and correspond to initial state, visiting I_1 first, and visiting I_2 first.

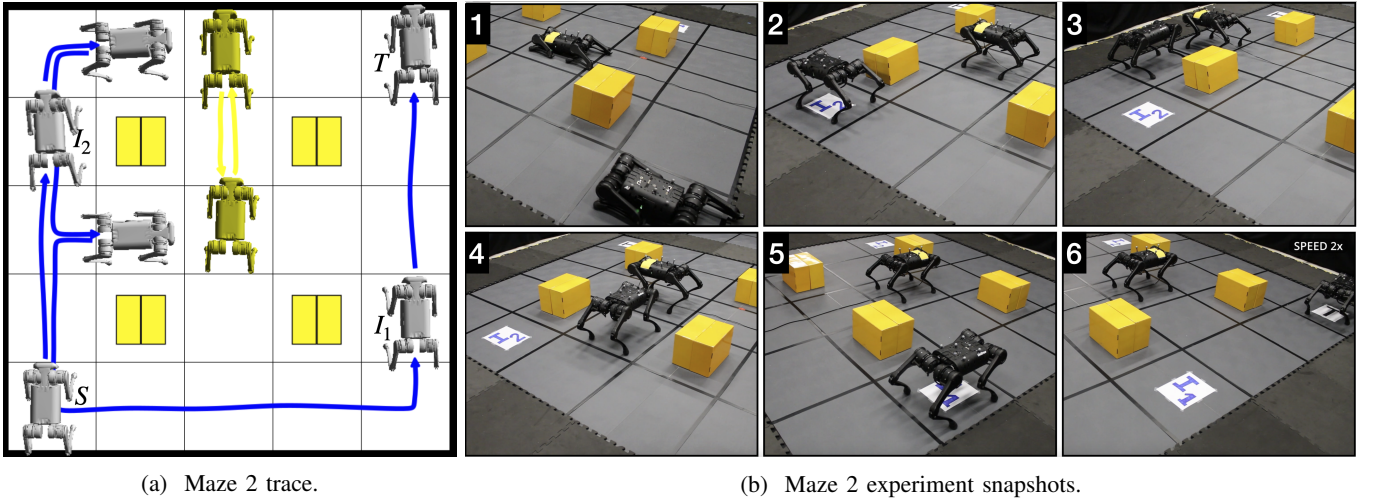


Fig. 14: Resulting test execution for the Maze 2 experiment with a dynamic test agent.

panel 3). It then decides to try navigating through the center of the grid, and is again blocked by the test agent (see panel 4). Subsequently, it decides to try the bottom path, visits I_1 (see panel 5), and successfully reaches the goal without any further test agent intervention. If the system decided to visit I_1 first, the adaptive test agent strategy would have blocked the system from reaching the goal directly from I_1 until it visits I_2 . This is an example with a maximum flow of $F = 2$, corresponding to the two unique ways for the system to reach the goal. For an alternative system controller in which the system chooses to approach the goal through I_1 , the simulated trace resulting from the test agent strategy is shown in Fig. 9d.

C. Runtimes

Table V showcases runtimes for simulated and hardware experiments involving static or reactive obstacles. Table VI shows runtimes for simulated and hardware experiments with a dynamic test agent. The size of the automata and graphs reported in these tables corresponds to the tuple $(|V|, |E|)$, where V is the number of nodes, and E the number of edges. To evaluate the scalability of this framework, we include runtimes on randomized grid worlds for specification sub-tasks in Table IV for static obstacles, and in Table III for

reactive obstacles. These experiments were conducted on an Apple M2 Pro with 16 GB of RAM. The Mars exploration example corresponds to solving an MILP with over 13,000 binary variables, for which the solver takes 46.6s to find the optimal solution. For examples involving a dynamic agent such as Maze 1 and Maze 2, our framework iterates through counterexamples that are not dynamically feasible for the test agent until it finds a solution.

For randomized experiments, we time out if the Gurobi fails to find a feasible solution to the MILP within 10 min. If it finds a feasible solution within 10 minutes, we allocate an additional minute for the optimizer reach the optimal, otherwise terminating the optimization with a feasible solution. For these experiments, we increase the length of the system and test objective for the following three classes of specification patterns: i) reachability, ii) reachability and reaction, and iii) reachability and safety. For reachability patterns, the set AP comprises of atomic propositions needed to describe the system and test objectives as follows, $\varphi_{\text{sys}} = \Diamond p_0$ and $\varphi_{\text{test}} = \bigwedge_{i=1}^n \Diamond p_i$, and the total number of atomic propositions are $|AP| = |\{p_0, \dots, p_n\}| = n + 1$. Similarly, for reachability and reaction patterns (case ii), we have $\varphi_{\text{sys}} = \Diamond p_1 \wedge \bigwedge_{i=2}^n \Box(p_i \rightarrow \Diamond q_i)$ and $\varphi_{\text{test}} = \Diamond p_0 \wedge$

TABLE II: Graph Construction Runtimes (with mean and standard deviation) for Random Grid World Experiments

Experiment		5×5	10×10	15×15	20×20
$ AP $	$ \mathcal{B}_\pi $	Graph Construction [s]			
Reachability:					
2	(4, 9)	0.046 ± 0.001	0.224 ± 0.0056	0.554 ± 0.009	1.078 ± 0.011
3	(8, 27)	0.344 ± 0.007	1.661 ± 0.022	4.004 ± 0.048	7.376 ± 0.061
4	(16, 81)	1.997 ± 0.077	9.895 ± 0.109	23.512 ± 0.179	43.188 ± 0.454
Reachability & Reaction:					
3	(6, 21)	0.090 ± 0.001	0.424 ± 0.016	1.037 ± 0.004	2.044 ± 0.013
5	(20, 155)	1.628 ± 0.087	7.560 ± 0.023	18.019 ± 0.129	33.539 ± 0.144
7	(68, 1065)	44.809 ± 0.996	209.612 ± 1.732	488.611 ± 6.308	869.060 ± 16.870
Reachability & Safety:					
3	(6, 18)	0.102 ± 0.002	0.508 ± 0.010	1.278 ± 0.022	2.557 ± 0.023
4	(6, 18)	0.116 ± 0.002	0.590 ± 0.009	1.485 ± 0.024	2.918 ± 0.046
5	(6,18)	0.179 ± 0.027	0.960 ± 0.037	2.329 ± 0.072	4.482 ± 0.116

TABLE III: Run Times (with mean and standard deviation) for Random Grid World Experiments solving **MILP-REACTIVE**

Experiment		5×5	10×10	15×15	20×20
$ AP $	$ \mathcal{B}_\pi $	Optimization[s], Success Rate (%)			
Reachability:					
2	(4, 9)	5.63 ± 13.43 100	64.62 ± 38.75 100	67.38 ± 25.47 100	68.63 ± 31.12 100
3	(8, 27)	23.36 ± 38.15 100	61.68 ± 35.12 100	91.54 ± 31.41 100	117.82 ± 34.89 100
4	(16, 81)	22.49 ± 36.33 100	83.52 ± 29.25 100	171.49 ± 50.72 100	317.62 ± 89.08 100
Reachability & Reaction:					
3	(6, 21)	5.97 ± 13.21 100	61.06 ± 34.67 100	71.64 ± 41.03 100	85.20 ± 19.49 100
5	(20, 155)	17.19 ± 25.51 100	78.44 ± 34.71 100	159.91 ± 76.63 100	279.86 ± 148.23 90
7	(68, 1065)	52.71 ± 41.23 100	331.32 ± 187.28 90	585.21 ± 67.58 15	600.00 ± 0.00 0
Reachability & Safety:					
3	(6, 18)	0.76 ± 1.52 100	70.82 ± 89.70 100	63.68 ± 27.54 100	80.58 ± 20.79 100
4	(6, 18)	0.15 ± 0.29 100	71.47 ± 80.61 100	59.59 ± 38.92 100	76.02 ± 27.11 100
5	(6, 18)	0.12 ± 0.18 100	94.68 ± 88.04 100	71.34 ± 30.89 100	82.54 ± 22.69 100

TABLE IV: Run Times (with mean and standard deviation) for Random Grid World Experiments solving **MILP-STATIC**.

Experiment		5 × 5		10 × 10		15 × 15		20 × 20	
AP	B _π	Optimization [s], Success Rate (%)							
Reachability:									
2	(4, 9)	8.17 ± 13.14	100	54.07 ± 17.98	100	60.17 ± 0.12	100	60.17 ± 0.10	100
3	(8, 27)	27.78 ± 21.71	100	60.17 ± 0.10	100	60.48 ± 0.86	100	74.02 ± 38.70	100
4	(16, 81)	52.60 ± 14.05	100	60.42 ± 0.34	100	82.02 ± 41.26	100	265.41 ± 203.51	80
Reachability & Reaction:									
3	(6, 21)	10.62 ± 14.85	100	60.09 ± 0.06	100	60.23 ± 0.24	100	60.34 ± 0.46	100
5	(20, 155)	20.41 ± 19.21	100	67.77 ± 31.90	100	95.31 ± 116.65	95	268.50 ± 222.14	75
7	(68, 1065)	36.64 ± 23.34	100	110.63 ± 92.81	100	419.77 ± 214.30	55	556.38 ± 131.06	10
Reachability & Safety:									
3	(6, 18)	1.27 ± 1.47	100	60.08 ± 0.06	100	57.27 ± 12.61	100	60.32 ± 0.24	100
4	(6, 18)	0.17 ± 0.23	100	60.06 ± 0.05	100	60.14 ± 0.10	100	60.30 ± 0.19	100
5	(6, 18)	0.11 ± 0.16	100	54.15 ± 17.80	100	60.17 ± 0.09	100	60.29 ± 0.26	100

TABLE V: Runtimes for Simulated and Hardware Experiments showing sizes of the automata and graphs

Experiment	$ \mathcal{B}_\pi $	$ T_{\text{sys}} $	$ G $	$G[s]$	$ \text{BinVars} $	$ \text{ContVars} $	$ \text{Constraints} $	Opt[s]	Flow	$ C $
Example 1	(4, 9)	(15, 53)	(27, 96)	0.0270	73	87	540	0.0003	3.0	14
Refueling	(6, 18)	(265, 1047)	(332, 1346)	0.6655	1014	1261	19819	0.8682	2.0	199
Mars Exploration	(36, 354)	(376, 1522)	(4073, 17251)	75.8313	13178	16604	1646480	46.6209	2.0	1641
Example 2	(8, 27)	(6, 17)	(20, 56)	0.0452	25	115	409	0.0003	2.0	4
Beaver Rescue	(12, 54)	(7, 19)	(15, 39)	0.0470	8	154	441	0.0001	2.0	2
Motion Primitives	(16, 81)	(15, 42)	(72, 207)	0.4286	106	761	2606	0.0005	3.0	15

TABLE VI: Runtimes for Simulated and Hardware Experiments with Dynamic Agents

Experiment	$ \mathcal{B}_\pi $	$ T_{\text{sys}} $	$ G $	$G[s]$	$ \text{BinVars} $	Opt[s]	Controller[s]	$ \mathcal{C}_{\text{ex}} $	Flow	$ C $
Maze 1	(16, 81)	(26, 80)	(196, 604)	1.6226	355	0.0007	68.7052	3	1.0	3
Patrolling	(6, 18)	(386, 1539)	(210, 831)	0.4573	621	6.0535	16.1191	0	1.0	13
Maze 2	(8, 27)	(21, 66)	(80, 252)	0.2195	176	0.0160	5.0072	5	2.0	8

$\bigwedge_{i=2}^n \Diamond p_i$, with $|AP| = |\{p_0, \dots, p_n, q_2, \dots, q_n\}| = 2n$. In the reachability and safety case (iii), only the length of the system objective changes: $\varphi_{\text{sys}} = \Diamond p_1 \wedge \bigwedge_{i=2}^n \Box \neg p_i$ and $\varphi_{\text{test}} = \Diamond p_0$, with $|AP| = |\{p_0, \dots, p_n\}| = n + 1$. Improving the runtimes for graph construction and controller synthesis subroutines is orthogonal to the focus of this paper. Since the test synthesis framework is carried out offline, we observe reasonable runtimes for medium-sized problems with hundreds to thousands of integer variables. An interesting direction for future research involves identifying good convex relaxations of the MILPs to further improve scalability.

IX. COMPARISON TO REACTIVE SYNTHESIS

We presented an approach to solve Problems 1 and 2 leveraging tools from automata theory and network flow optimization. In particular, for Problem 2, we rely on the optimization solution to construct a GR(1) specification to reactively synthesize a test agent strategy. One indication of the optimization step being necessary is the computational complexity of the problem. If the problem data are consistent, there exists a GR(1) specification for the test agent that would solve the problem, but directly expressing this specification is impractical. Essentially, the challenge is in finding the restrictions on system actions, which are then captured in the sub-formulae of the GR(1) specification. In this section, we argue that we cannot solve Problems 1 and 2 solely via synthesis from an LTL specification.

To the authors' knowledge, directly capturing the different perspectives of the system and the test agent in this neither fully adversarial nor fully cooperative setting is not possible with current state-of-the-art approaches in GR(1) synthesis. Particularly in the reactive setting, the test strategy must ensure that from the system's perspective, there always exists a path to the system goal. To capture this constraint, we reason over a second product graph that represents the system perspective. It is not obvious how this semi-cooperative setting can be directly encoded as a synthesis problem in common temporal logics.

In the static setting, the problem can be posed on a single graph. However, it is difficult to find the set of static obstacles

directly from GR(1) synthesis. Every state in the winning set describes an edge-cut combination, but qualitative GR(1) synthesis cannot maximize the flow or minimize the cuts. Furthermore, the winning set can include states that vacuously satisfy the formula, i.e., not allowing the system any path to the goal. Finally, the combinatorial complexity of the problem would manifest as follows. Although the time complexity of GR(1) synthesis is $O(N^3)$ in the number of states N , we require an exponential number of states to characterize the GR(1) formula. For example, in Figure 15, this is illustrated for the GR(1) formula:

$$\Box \varphi_{\text{sys}}^{\text{dyn}} \wedge \Box \Diamond T \rightarrow \Box \varphi_{\text{test}}^{\text{dyn}} \wedge \Box \varphi_{\text{test}}^{\text{aux_dyn}} \wedge \Box \Diamond I_{\text{aux}},$$

where $\varphi_{\text{sys}}^{\text{dyn}}$ captures the system transitions on the grid world, $\varphi_{\text{test}}^{\text{dyn}}$ are the dynamics of the test environment, and $\varphi_{\text{test}}^{\text{aux_dyn}}$ and I_{aux} capture the $\Diamond I$ condition in GR(1) form. In this example, each edge in the system transition system T_{sys} can take 0/1 values, and once an edge is cut, it remains cut and the system cannot take a transition that corresponds to a cut edge. Due to this, the number of states N to describe the GR(1) formula includes the $2^{|T_{\text{sys}} \cdot E|}$ states that characterize the edge cuts. As seen in Figure 15, the direct GR(1) synthesis approach returns a trivial solution corresponding to an impossible setting for the system. Finally, even when an acceptable solution is returned, the problem being at least NP-hard will result in the combinatorial complexity manifesting in the synthesis approach.

One key advantage of the network flow optimization is reasoning over flows as opposed to paths, which allows for tractable implementations. These insights from network flow optimization in this work can help in driving further research along these directions.

X. CONCLUSION AND FUTURE WORK

We presented a framework to synthesize least-restrictive strategies for test environments according to specified system and test objectives. To do this, we formulate a network flow-based MILP corresponding to the types of agents available in the test environment. In the case of a dynamic test agent,

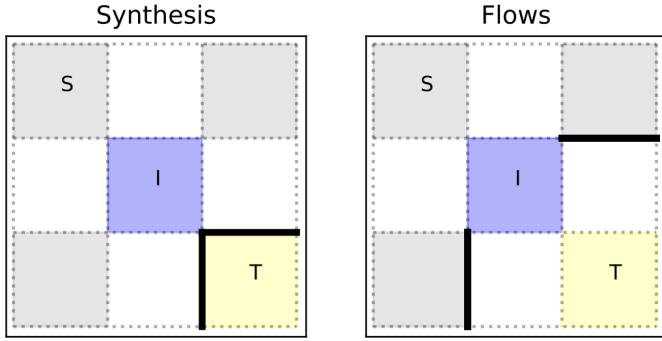


Fig. 15: Solution returned by GR(1) synthesis and the network flow optimization in the case of static constraints

we parse the solution of the MILP to synthesize a test agent strategy via reactive synthesis. Furthermore, we use a counterexample-guided approach to find a realizable test agent strategy. Our problem is shown to be NP-hard, yet the MILP can handle medium-sized problem instances. Our test strategies are such that the system is minimally restricted while routing the test execution through the test objective without creating a livelock. Therefore, a test execution in which the system fails to meet the system objective is solely the fault of the system, and not due to the test environment.

There are several exciting future directions. First, we aim to extend this framework to automatically select dynamic test agents from a library. This selection can be optimized to meet user-defined metrics such as testing effort or cost. Secondly, we wish to improve the runtime of our algorithm by using symbolic methods to speed up graph construction and exploring convex relaxations to the MILP. More broadly, we want to investigate how to incorporate test metrics such as coverage and difficulty into our framework.

ACKNOWLEDGMENT

The authors acknowledge Emily Fourney, Chris Umans, Scott Livingston, Joel Burdick, Ioannis Filippidis, Mani Chandy, and Lijun Chen for useful discussions.

REFERENCES

- [1] I. S. Organization, "Road vehicles: Safety of the intended functionality (ISO Standard No. 21448:2022)," 2022. <https://www.iso.org/standard/77490.html>, Last accessed on 2024-04-11.
- [2] Zoox, "Putting Zoox to the test: preparing for the challenges of the road," 2021. <https://zoox.com/journal/structured-testing/>, Last accessed on 2024-04-11.
- [3] N. Webb, D. Smith, C. Ludwick, T. Victor, Q. Hommes, F. Favaro, G. Ivanov, and T. Daniel, "Waymo's safety methodologies and safety readiness determinations," 2020.
- [4] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Conference on Robot Learning*, pp. 1–16, PMLR, 2017.
- [5] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, "Scenic: a language for scenario specification and scene generation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 63–78, 2019.
- [6] A. Gambi, T. Huynh, and G. Fraser, "Generating effective test cases for self-driving cars from police reports," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 257–267, 2019.
- [7] C. Stark, C. Medrano-Berumen, and M. Akbaş, "Generation of autonomous vehicle validation scenarios using crash data," in *2020 South-eastCon*, pp. 1–6, 2020.
- [8] G. Lou, Y. Deng, X. Zheng, M. Zhang, and T. Zhang, "Testing of autonomous driving systems: where are we and where should we go?," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 31–43, 2022.
- [9] A. Corso, R. Moss, M. Koren, R. Lee, and M. Kochenderfer, "A survey of algorithms for black-box safety validation of cyber-physical systems," *Journal of Artificial Intelligence Research*, vol. 72, pp. 377–428, 2021.
- [10] H. Winner, K. Lemmer, T. Form, and J. Mazzega, "Pegasus—first steps for the safe introduction of automated driving," in *Road Vehicle Automation 5*, pp. 185–195, Springer, 2019.
- [11] L. Li, W.-L. Huang, Y. Liu, N.-N. Zheng, and F.-Y. Wang, "Intelligence testing for autonomous vehicles: A new approach," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 2, pp. 158–166, 2016.
- [12] G. E. Mullins, P. G. Stankiewicz, R. C. Hawthorne, and S. K. Gupta, "Adaptive generation of challenging scenarios for testing and evaluation of autonomous vehicles," *Journal of Systems and Software*, vol. 137, pp. 197–215, 2018.
- [13] A. Corso, P. Du, K. Driggs-Campbell, and M. J. Kochenderfer, "Adaptive stress testing with reward augmentation for autonomous vehicle validation," in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pp. 163–168, IEEE, 2019.
- [14] S. Feng, H. Sun, X. Yan, H. Zhu, Z. Zou, S. Shen, and H. X. Liu, "Dense reinforcement learning for safety validation of autonomous vehicles," *Nature*, vol. 615, no. 7953, pp. 620–627, 2023.
- [15] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan, "S-talro: A tool for temporal logic falsification for hybrid systems," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 254–257, Springer, 2011.
- [16] H. Abbas and G. Fainekos, "Linear hybrid system falsification through local search," in *International Symposium on Automated Technology for Verification and Analysis*, pp. 503–510, Springer, 2011.
- [17] G. E. Fainekos and G. J. Pappas, "Robustness of temporal logic specifications for continuous-time signals," *Theoretical Computer Science*, vol. 410, no. 42, pp. 4262–4291, 2009.
- [18] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *International Conference on Computer Aided Verification*, pp. 167–170, Springer, 2010.
- [19] D. J. Fremont, E. Kim, Y. V. Pant, S. A. Seshia, A. Acharya, X. Bruso, P. Wells, S. Lemke, Q. Lu, and S. Mehta, "Formal scenario-based testing of autonomous vehicles: From simulation to the real world," in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pp. 1–8, IEEE, 2020.
- [20] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski, "Simulation-based adversarial test generation for autonomous vehicles with machine learning components," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1555–1562, IEEE, 2018.
- [21] C. Innes and S. Ramamoorthy, "Automated testing with temporal logic specifications for robotic controllers using adaptive experiment design," in *2022 International Conference on Robotics and Automation (ICRA)*, pp. 6814–6821, 2022.
- [22] P. Akella, M. Ahmadi, R. M. Murray, and A. D. Ames, "Formal test synthesis for safety-critical autonomous systems based on control barrier functions," in *2020 59th IEEE Conference on Decision and Control (CDC)*, pp. 790–795, 2020.
- [23] T. Wongpiromsarn, M. Ghasemi, M. Cubuktepe, G. Bakirtzis, S. Carr, M. O. Karabag, C. Neary, P. Gohari, and U. Topcu, "Formal methods for autonomous systems," *arXiv preprint arXiv:2311.01258*, 2023.
- [24] G. Fainekos, H. Kress-Gazit, and G. Pappas, "Hybrid controllers for path planning: A temporal logic approach," in *Proceedings of the 44th IEEE Conference on Decision and Control*, pp. 4885–4890, 2005.
- [25] L. Tan, O. Sokolsky, and I. Lee, "Specification-based testing with linear temporal logic," in *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration*, 2004. IRI 2004., pp. 493–498, IEEE, 2004.
- [26] E. Plaku, L. E. Kavrakli, and M. Y. Vardi, "Falsification of LTL safety properties in hybrid systems," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 4, pp. 305–320, 2013.
- [27] G. Fraser and F. Wotawa, "Using LTL rewriting to improve the performance of model-checker based test-case generation," in *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, pp. 64–74, 2007.

- [28] G. Fraser and P. Ammann, "Reachability and propagation for LTL requirements testing," in *2008 The Eighth International Conference on Quality Software*, pp. 189–198, IEEE, 2008.
- [29] R. Bloem, G. Fey, F. Greif, R. Könighofer, I. Pill, H. Riemer, and F. Röck, "Synthesizing adaptive test strategies from temporal logic specifications," *Formal Methods in System Design*, vol. 55, no. 2, pp. 103–135, 2019.
- [30] J. Tretmans, "Conformance testing with labelled transition systems: Implementation relations and test generation," *Computer Networks and ISDN Systems*, vol. 29, no. 1, pp. 49–79, 1996.
- [31] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran, "Killing strategies for model-based mutation testing," *Software Testing, Verification and Reliability*, vol. 25, no. 8, pp. 716–748, 2015.
- [32] R. Hierons, "Applying adaptive test cases to nondeterministic implementations," *Information Processing Letters*, vol. 98, no. 2, pp. 56–60, 2006.
- [33] A. Petrenko and N. Yevtushenko, "Adaptive testing of nondeterministic systems with FSM," in *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pp. 224–228, IEEE, 2014.
- [34] 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*, pp. 179–190, 1989.
- [35] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive (1) designs," *Journal of Computer and System Sciences*, vol. 78, no. 3, pp. 911–938, 2012.
- [36] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.
- [37] M. Yannakakis, "Testing, optimization, and games," in *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pp. 78–88, IEEE, 2004.
- [38] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp, "Optimal strategies for testing nondeterministic systems," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 55–64, 2004.
- [39] A. David, K. G. Larsen, S. Li, and B. Nielsen, "Cooperative testing of timed systems," *Electronic Notes in Theoretical Computer Science*, vol. 220, no. 1, pp. 79–92, 2008.
- [40] E. Bartocci, R. Bloem, B. Maderbacher, N. Manjunath, and D. Ničković, "Adaptive testing for specification coverage in CPS models," *IFAC-PapersOnLine*, vol. 54, no. 5, pp. 229–234, 2021.
- [41] T. Marcucci, J. Umenberger, P. Parrilo, and R. Tedrake, "Shortest paths in graphs of convex sets," *SIAM Journal on Optimization*, vol. 34, no. 1, pp. 507–532, 2024.
- [42] T. Marcucci, M. Petersen, D. von Wrangel, and R. Tedrake, "Motion planning around obstacles with convex optimization," *Science Robotics*, vol. 8, no. 84, p. ead7843, 2023.
- [43] H. Zhang, M. Fontaine, A. Hoover, J. Togelius, B. Dilkina, and S. Nikolaidis, "Video game level repair via mixed integer linear programming," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, pp. 151–158, 2020.
- [44] M. Fontaine, Y.-C. Hsu, Y. Zhang, B. Tjanaka, and S. Nikolaidis, "On the Importance of Environments in Human-Robot Coordination," in *Proceedings of Robotics: Science and Systems*, (Virtual), July 2021.
- [45] A. Badithela, J. B. Graebener, W. Ubellacker, E. V. Mazumdar, A. D. Ames, and R. M. Murray, "Synthesizing reactive test environments for autonomous systems: testing reach-avoid specifications with multi-commodity flows," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 12430–12436, IEEE, 2023.
- [46] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger, "Specification patterns for robotic missions," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2208–2224, 2019.
- [47] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray, "TuLiP: a software toolbox for receding horizon temporal logic planning," in *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control, HSCC '11*, (New York, NY, USA), p. 313–314, Association for Computing Machinery, 2011.
- [48] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Receding horizon temporal logic planning," *IEEE Transactions on Automatic Control*, vol. 57, no. 11, pp. 2817–2830, 2012.
- [49] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Temporal-logic-based reactive mission and motion planning," *IEEE Transactions on Robotics*, vol. 25, no. 6, pp. 1370–1381, 2009.
- [50] C. Belta and S. Sadraadini, "Formal methods for control synthesis: An optimization perspective," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 2, pp. 115–140, 2019.
- [51] J. R. Büchi, *On a Decision Method in Restricted Second Order Arithmetic*, pp. 425–435. New York, NY: Springer New York, 1990.
- [52] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, pp. 1–64, 2011.
- [53] K. Havelund and G. Rosu, "Monitoring programs using rewriting," in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pp. 135–143, IEEE, 2001.
- [54] A. Morgenstern, M. Gesell, and K. Schneider, "An asymptotically correct finite path semantics for LTL," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pp. 304–319, Springer, 2012.
- [55] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.
- [56] V. V. Vazirani, *Approximation algorithms*, vol. 1. Springer, 2001.
- [57] I. Filippidis, S. Dathathri, S. C. Livingston, N. Ozay, and R. M. Murray, "Control design for hybrid systems with tulip: The temporal logic planning toolbox," in *2016 IEEE Conference on Control Applications (CCA)*, pp. 1030–1041, IEEE, 2016.
- [58] S. Maoz and J. O. Ringert, "GR(1) synthesis for LTL specification patterns," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pp. 96–106, 2015.
- [59] S. A. Cook, "The complexity of theorem-proving procedures," in *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, pp. 143–152, 2023.
- [60] C. H. Papadimitriou, *Computational complexity*, p. 260–265. GBR: John Wiley and Sons Ltd., 2003.
- [61] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2023.
- [62] W. Ubellacker and A. D. Ames, "Robust locomotion on legged robots through planning on motion primitive graphs," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, preprint, 2023.



Josefine B. Graebener (Student Member, IEEE) received a B.Eng. in Aerospace Engineering in 2017 from the Aachen University of Applied Sciences (FH Aachen) in Aachen, Germany, and a M.S. in Space Engineering from California Institute of Technology in 2019. Currently, she is a Ph.D. candidate in Space Engineering with a minor in Computer Science at the California Institute of Technology. Her research interest lies in using formal methods for test and evaluation of autonomous systems, and system diagnostics.



Apurva Badithela (Student Member, IEEE) received a Bachelors degree in Aerospace Engineering and Mechanics in 2018 from the University of Minnesota, Twin-Cities. Currently, she is a Ph.D. candidate in Control and Dynamical Systems at the California Institute of Technology. Her dissertation work focuses on Formal Test Synthesis and System-level Evaluation for Safety-Critical Autonomous Systems.



Denizalp Goktas (Student Member, IEEE), Denizalp Goktas is a Ph.D. Candidate in Computer Science at Brown University. His research focuses on artificial intelligence, particularly how it intersects with economics and computer science. His research seeks to create algorithms for games and markets, aiming to use these to tackle problems practical problem such as in economics and robotics. He is supported by a JP Morgan AI fellowship. Previously, Denizalp earned his BA in Computer Science and Statistics from Columbia

University and another BA in Political Science and Economics from Sciences Po. His past research experience includes internships at Google DeepMind and JP Morgan, and a visiting scholar position at UC Berkeley's Simons Institute.



Wyatt Ubellacker (Student Member, IEEE), earned his B.S. and M.S. degrees in Mechanical Engineering from the Massachusetts Institute of Technology in 2013 and 2016, respectively. Prior to joining Caltech in 2019, he was a Robotics Technologist at the Jet Propulsion Laboratory, where he wrote autonomy and control algorithms for the Mars Perseverance Rover.

Currently, he is a Ph.D. candidate in Control and Dynamical Systems at Caltech. His research interests focus on control and autonomy for dynamic

platforms, with a special emphasis on robotic morphologies that are capable of exhibiting a wide variety of behaviors.



Eric V. Mazumdar (Member, IEEE) received a B.S. degree in Computer Science from the Massachusetts Institute of Technology (MIT) in 2015, and a Ph.D in Electrical Engineering and Computer Science from UC Berkeley in 2021.

Currently he is an Assistant Professor in Computing and Mathematical Sciences and Economics at Caltech. His research interests lie at the intersection of machine learning and economics, focusing on developing theoretical foundations and tools to confidently deploy machine learning algorithms into

societal systems, particularly in settings with uncertain, dynamic environments in which learning algorithms interact with strategic agents.

Dr. Mazumdar received the NSF CAREER Award in 2023 as well as a Research Fellowship for Learning in Games from the Simons Institute for Theoretical Computer Science.



Aaron D. Ames (Fellow, IEEE) received a B.S. degree in Mechanical Engineering and a B.A. degree in Mathematics from the University of St. Thomas in 2001, and a M.A. degree in Mathematics and a Ph.D. in Electrical Engineering and Computer Sciences from UC Berkeley in 2006.

Currently he is the Bren Professor of Mechanical and Civil Engineering and Control and Dynamical Systems at the California Institute of Technology. Prior to joining Caltech, he was an Associate Professor in Mechanical Engineering and Electrical &

Computer Engineering at the Georgia Institute of Technology. He was as a Postdoctoral Scholar in Control and Dynamical Systems at Caltech from 2006 to 2008, and began is faculty career at Texas A&M University in 2008. His research interests span the areas of robotics, nonlinear control and hybrid systems, with a special focus on applications to bipedal robotic walking—both formally and through experimental validation.

Dr. Ames was the recipient of the 2005 Leon O. Chua Award for achievement in nonlinear science at UC Berkeley, and the 2006 Bernard Friedman Memorial Prize in Applied Mathematics. Dr. Ames received the NSF CAREER award in 2010, and the Donald P. Eckman Award in 2015, and the 2019 Antonio Ruberti Young Researcher Prize.



Richard M. Murray (Fellow, IEEE) received the B.S. degree in Electrical Engineering from California Institute of Technology in 1985 and the M.S. and Ph.D. degrees in Electrical Engineering and Computer Sciences from the University of California, Berkeley, in 1988 and 1991, respectively.

He is currently the Thomas E. and Doris Everhart Professor of Control & Dynamical Systems and Bioengineering at Caltech. Murray's research is in the application of feedback and control to networked systems, with applications in synthetic biology and

autonomy. Current projects include design and implementation of synthetic cells and design, verification, and test synthesis for discrete decision-making protocols for safety-critical, reactive control systems.

Dr. Murray's professional awards include the Richard P. Feynman-Hughes Faculty Fellowship in 1993, awarded annually to an outstanding young faculty member in Engineering and Applied Science at Caltech, the National Science Foundation Early Faculty Career Development (CAREER) Award in 1995, the Office of Naval Research Young Investigator Award in 1995 and the Donald P. Eckman Award in 1997. He is a Fellow of the Institute for Electrical and Electronics Engineers (IEEE) and holds an honorary doctorate from Lund University in Sweden. He is an elected member of the National Academy of Engineering (2013). Dr. Murray received the IEEE Bode Lecture Prize in 2016, the IEEE Control Systems Award in 2017, and the AACC John R. Ragazzini Education Award in 2019.