Title

CPSGrader: Synthesizing Temporal Logic Testers for Auto-Grading an Embedded Systems Laboratory

Permalink
https://escholarship.org/uc/item/9tb3k46m

ISBN
9781450330527

Authors
Juniwal, Garvit
Donz, Alexandre
Jensen, Jeff C
et al.

Publication Date
2014-10-12

DOI
10.1145/2656045.2656053

Peer reviewed

CPSGrader: Synthesizing Temporal Logic Testers for Auto-Grading an Embedded Systems Laboratory

Garvit Juniwal
UC Berkeley
garvitjuniwal@eecs.berkeley.edu

Alexandre Donzé
UC Berkeley
donze@eecs.berkeley.edu

Jeff C. Jensen
National Instruments
jeff.c.jensen@ni.com

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

ABSTRACT

We consider the problem of designing an automatic grader for a laboratory in the area of cyber-physical systems. The goal of this laboratory is to program a robot for specified navigation tasks. Given a candidate student solution (control program for the robot), our grader first checks whether the robot performs the task correctly under a representative set of environment conditions. If it does not, the grader automatically generates feedback hinting at possible errors in the program. The auto-grader is based on a novel notion of constrained parameterized tests based on signal temporal logic (STL) that capture symptoms pointing to success or causes of failure in traces obtained from a realistic simulator. We define and solve the problem of synthesizing constraints on a parameterized test such that it is consistent with a set of reference solutions with and without the desired symptom. The usefulness of our grader is demonstrated using a large data set obtained from an on-campus laboratory-based course at UC Berkeley.

1. INTRODUCTION

Massive open online courses (MOOCs) [22] and related technological advances promise to bring world-class education to anyone with Internet access. Additionally, MOOCs present a range of problems to which the field of formal methods has much to contribute. These include *automatic grading*, *automated exercise generation*, and *virtual laboratory environments*. In automatic grading, a computer program verifies that a candidate solution provided by a student is "correct", i.e., that it meets certain instructor-specified criteria (the specification). In addition, and particularly when the solution is incorrect, the automatic grader (henceforth, *auto-grader*) should provide feedback to the student as to where he/she went wrong. Automatic exercise generation is the process of synthesizing problems (with associated solutions) that test students' understanding of course material, often starting from instructor-provided sample problems. Finally, for courses involving laboratory assignments, a virtual laboratory (henceforth, lab) seeks to provide the remote student with an experience similar to that provided in a real, on-campus lab.

Lab-based courses that are not software-only pose a particular technical challenge. An example of such a course is *Introduction to Embedded Systems* at UC Berkeley [13]. In this course, students not only learn theoretical content on modeling, design, and analysis [14], but also perform lab assignments on programming an embedded platform interfaced to a mobile robot [9]. What would an online lab assignment in embedded systems look like? In an ideal world, we would provide an infrastructure where students can log in remotely to a computer which has been preconfigured with all development tools and laboratory exercises; in fact, pilot projects exploring this approach have already been undertaken (e.g., see [20]). However, in the MOOC setting, the large numbers of students makes such a remotely-accessible physical lab expensive and impractical. A virtual lab environment, driven by simulation of real-world environments, appears to be the only solution at present. For example, the MIT circuits course (MITx 6.002x) uses rudimentary circuit simulation software [21].

In this paper, we formalize the auto-grading problem for a virtual lab environment in the field of embedded and cyber-physical systems (CPS). The virtual lab under consideration is the one designed for EECS149.1x [15], a MOOC on Cyber-Physical Systems offered on the edX platform, based on the afore-mentioned on-campus course, and described in more detail in Sec. 2. The main point we make here is that the dynamical model for the virtual lab is so complex that simulation is currently the only verification method that can be practically employed. Thus, the auto-grader is based on simulation-based verification. The high-level approach, previously hinted at in a position paper [10], is as follows. Correctness properties are formalized in *signal temporal logic* (STL) [19]. Simulation test benches are created by a combination of manual environment setup and simulation-based falsification implemented in tools such as Breach [5]. For each lab assignment, there is an *end-to-end correctness property*, hereafter referred to as the *goal* property. If the goal is satisfied, the student solution (hereafter referred to as a *controller*) is deemed correct. Otherwise, it is incorrect, and more analysis must be performed to identify the mistake (fault) and provide feedback. This latter analysis is based on monitoring simulation traces of the student controller on a library of known faults, also formalized in STL. If any of these "fault properties" hold for a student controller, they are provided to the student as feedback.

This approach, though straightforward on the surface, requires further technical advances to be effective. The first problem is that the STL properties that encode both goal and fault properties reference parameters that can vary over the set of environments and student controllers; in fact, such variation must be allowed. For example, in a real lab, students may program robots to move at different velocities while performing obstacle avoidance. If the goal of the lab is only to correctly avoid an obstacle, the speed at which

ESWEEK'14, October 12-17 2014, New Delhi, India
Copyright 2014 ACM 978-1-4503-3052-7/14/10 ...$15.00.
http://dx.doi.org/10.1145/2656045.2656053

it does so is irrelevant. However, given the variations in the controllers students design, setting a reasonable range for parameters such as time or velocity in STL properties can be tricky. Similarly, environments can also be parametric (for example, the location of obstacles) and tests should be synthesized in a manner that accounts for these variations. Thus, an effective approach to auto-grading CPS labs requires one to solve a certain *parameter synthesis* problem.

We formalize this parameter synthesis problem and give an algorithm to solve it. First, we define the notion of a *parametrized test* which is a combination of a parametrized environment and a parametrized STL (PSTL) property. A parametrized test is thus a collection of tests. However, as discussed above, one needs to impose a constraint on this collection to capture "legal" variations in student solutions. Such a constraint, termed a *sub-domain*, defines the allowed set of parameter valuations. However, manually computing this sub-domain is tedious and error-prone. We therefore give an algorithmic approach to synthesize the sub-domain from reference controllers that should/should not pass the test bench. In practice, it is easier for instructors to provide such reference controllers than it is to manually compute sub-domains. The resulting *constrained parameterized test bench* then becomes the "specification" that determines whether a student solution is correct, and, if not, which fault is present. Further, we identify a property, *monotonicity*, under which we can efficiently compute the sub-domain, and which holds for the lab of interest.

Any auto-grader must have at least two desirable properties: *accuracy* and *efficiency*. The former means that the auto-grader must correctly classify right and wrong student solutions, and for wrong solutions, correctly explain the mistake (fault). The latter means that it must run efficiently in practice. For efficiency, we show how monotonicity can be exploited again to avoid the need to run the entire constrained parametric test bench. Instead, we define the notion of an *adequate* test sample and show that it is much smaller in practice than the entire constrained test bench. We also provide an experimental evaluation on the on-campus lab demonstrating that our approach is both accurate and efficient in practice.

To summarize, the main novel contributions of this paper are:

- A formalization of the auto-grading problem for simulation-based virtual laboratories in cyber-physical systems;
- A formalization of the problem of synthesizing a constrained parametric test bench for the auto-grader along with an efficient solution approach based on monotonicity, and
- An empirical evaluation demonstrating the accuracy and efficiency of CPSGrader, the auto-grader for the on-campus embedded systems lab, on a database of actual student solutions.

The outline of the rest of the paper is as follows. We begin in Sec. 2 by describing the motivating application for this work, the lab assignments in the course. We introduce basic terminology and background results in Sec. 3. In Sec. 4, we describe the main theoretical contributions, including our formalization and solution approach. Experimental results are given in Sec. 5. We discuss related work in Sec. 6 and conclude with future directions Sec. 7.

2. MOTIVATING APPLICATION

The embedded systems laboratory course offered at University of California, Berkeley employs a custom mobile robotic platform called the Cal Climber [8, 7]. The Cal Climber is based on the commercially-available iRobot Create (derived from the iRobot-Roomba autonomous vacuum cleaner) (Fig. 1a), and the National Instruments myRIO embedded controller. This off-the-shelf platform is capable of driving, sensing bumps and cliffs, executing simple scripts, and communicating with an external controller. This configuration demonstrates the composition of cyber-physical systems, where a robotics platform is modeled as a sub-system and treated as a collection of sensors and actuators potentially located beyond a network boundary. The problem statement centers on model-based design and is given as follows (paraphrased from [9]):

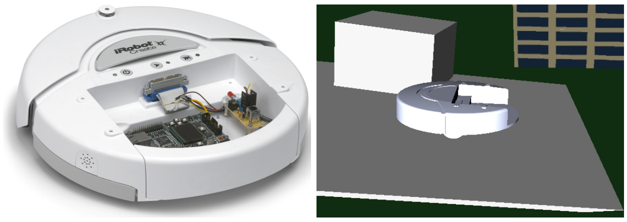

(a) (b)

Figure 1: (a) Cal Climber laboratory platform. (b) Cal Climber in the LabVIEW Robotics Environment Simulator.

> *Design a StateChart to drive the Cal Climber. On level ground, your robot should drive straight. When an obstacle is encountered, such as a cliff or an object, your robot should navigate around the object and continue in its original orientation. On an incline, your robot should navigate uphill, while still avoiding obstacles. Use the accelerometer to detect an incline and as input to a control algorithm that maintains uphill orientation.*

Source files distributed with the Cal Climber laboratory are structured such that students only need to implement a function that receives as arguments the most recent values of the accelerometer and robot sensors and returns desired wheel speeds. This function is called repeatedly at short regular intervals of time (60 ms in our case) with most recent sensor and accelerometer data. Students implement this function for controlling the Cal Climber. In the on-campus course, students first prototype their controller to work within a simulated environment (without any auto-grading) based on the LabVIEW Robotics Environment Simulator by National Instruments. The simulator is based on the Open Dynamics Engine [27] rigid body dynamics software that can simulate robots in a virtual environment(Fig. 1b). In EECS149.1x, the afore-mentioned online version of the course, the same simulator, extended with the auto-grader described in the present paper, has been used.

We refer to the functions implemented by students as *solutions*, or *controllers*. A solution is evaluated in a collection of environments against a collection of goal and fault properties, forming *test benches* (a notion formalized in the following sections). For this purpose, the simulator allows to define customized environments (with walls, objects, obstacles, ramps, etc) described in XML files and we further extended its API to facilitate the exportation of traces of simulation to external property monitoring tools. For the experiments reported in Section 5, traces were written in files and monitored offline against STL properties in the publicly-available Breach toolkit [5].

3. PRELIMINARIES

3.1 Signals, Controllers, and Environments

Definition 1. (Signal) A (uni-dimensional) *signal* is a function mapping the time domain $\mathbb{T} = \mathbb{R}_{\geq 0}$ to the reals $\mathbb{R}$.

Boolean signals, used to represent discrete dynamics, are signals whose values are restricted to *false* (denoted $\perp$) and *true* (denoted $\top$). Vectors in $\mathbb{R}^n$ with $n > 1$ are denoted in bold fonts and their components are indexed from 1 to n, for example, $\mathbf{p} = (p_1, \cdots, p_n)$. Likewise, a *multi-dimensional signal* $\mathbf{x}$ is a function from $\mathbb{T}$ to $\mathbb{R}^n$ such that $\forall t \in \mathbb{T}$, $\mathbf{x}(t) = (x_1(t), \cdots, x_n(t))$. We will use the term "signal" to also refer to multi-dimensional signals.

Definition 2. (Controller) A *controller* C is a (deterministic) dynamical system that takes as input a signal $\mathbf{y}(t)$ and computes an output signal $\mathbf{u}(t)$. It is common to drop time, and say $\mathbf{u} = C(\mathbf{y})$.

Note that we make no assumption about how a controller computes its output. A controller can have discrete or continuous dynamics or it can be a hybrid system. As an example, a program running on the Cal Climber is a controller that takes bumps and cliff sensors signals, and accelerometer data as input $\mathbf{y}(t) = (\mathsf{bump}(t), \mathsf{cliff}(t), \mathsf{accel}(t))$, and responds with the desired left and right wheel speeds as output $\mathbf{u}(t) = (\mathsf{lws}(t), \mathsf{rws}(t))$.

Definition 3. (Environment) An *environment* E for a controller C is a dynamical system generating all inputs to C.

As before, we make no assumptions about the form of the environment. All we assume is the existence of a simulator that can take representations of E and C, compose them, and produce execution traces of the composite system. In other words, the simulator is an oracle that gives semantics to the composite system $E \| C$.

We only consider deterministic environments, i.e., the composition of a controller and an environment has deterministic behavior. For example, an arena composed of obstacles and hills on level ground is an environment for the Cal Climber controller. Formally, a *trace* $\mathsf{sim}(C, E)$ is a multi-dimensional signal $(\mathbf{x}(t), \mathbf{y}(t), \mathbf{u}(t))$ consisting of the inputs $\mathbf{y}$ and outputs $\mathbf{u}$ of the controller and optionally other signals $\mathbf{x}$ regarding the state of the environment. For example, the position and orientation (in the plane of the ground) of the robot in the arena $\mathbf{x}(t) = (\mathsf{pos}(t), \mathsf{angle}(t))$ are a part of the observable environment state. By varying the environment, or the property being verified on the composition (see Sec. 3.2), the instructor can test different features of the controller.

3.2 Signal Temporal Logic

Since propositional (linear) temporal logic was introduced by Amir Pnueli [23], variants have also been proposed. Temporal logics to reason about real-time signals, such as Timed Propositional Temporal Logic [2], and Metric Temporal Logic (MTL) [12] were introduced later to deal with dense-time signals. More recently, Signal Temporal Logic [19] was proposed in the context of analog and mixed-signal circuits to deal with dense-time signals taking values over both discrete and continuous domains. We use STL as the specification language for the Embedded Systems lab assignment. Goals that the robotic controller must achieve are expressed as STL properties.

The primitive constraints, or *predicates*, in STL take the form $\mu \doteq f(\mathbf{x}) \sim \pi$, where f is a scalar-valued function over the signal $\mathbf{x}$, $\sim \in \{<, \leq, \geq, >, =, \neq\}$, and π is a real number. Temporal formulas are formed using temporal operators, "always" (denoted as $\Box$), "eventually" (denoted as $\Diamond$) and "until" (denoted as $\mathbf{U}$). Each temporal operator is indexed by intervals of the form (a, b), $(a, b]$, $[a, b)$, $[a, b]$, (a, ∞) or $[a, \infty)$ where each of a, b is a non-negative real-valued constant. If I is an interval, then an STL formula is written using the grammar:

$$\varphi := \top \mid \mu \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \, \mathbf{U}_I \, \varphi_2$$

The always and eventually operators are defined as special cases of the until operator in the standard way: $\Box_I \varphi \triangleq \neg\Diamond_I \neg\varphi$, $\Diamond_I \varphi \triangleq \top \mathbf{U}_I \, \varphi$. When the interval I is omitted, we use the default interval of $[0, +\infty)$. The semantics of STL formulas are defined informally as follows. The signal $\mathbf{x}$ satisfies $f(\mathbf{x}) > 10$ at time t (where $t \geq 0$) if $f(\mathbf{x}(t)) > 10$. It satisfies $\varphi = \Box_{[0,2)} \, (x > -1)$ if for all time $0 \leq t < 2$, $x(t) > -1$. The signal x_1 satisfies $\varphi = \Diamond_{[1,2)} \, x_1 > 0.4$ iff there exists time t such that $1 \leq t < 2$ and $x_1(t) > 0.4$. The two-dimensional signal $\mathbf{x} = (x_1, x_2)$ satisfies the formula $\varphi = (x_1 > 10) \, \mathbf{U}_{[2.3,4.5]} \, (x_2 < 1)$ iff there is some time u where $2.3 \leq u \leq 4.5$ and $x_2(u) < 1$, and for all time v in $[2.3, u)$, $x_1(u)$ is greater than 10. The formal semantics of STL can be found in [19] and is given in Appendix A.

Parametric Signal Temporal Logic (PSTL) is an extension of STL introduced in [3] to define *template formulas* containing unknown parameters. Syntactically speaking, a PSTL formula is an STL formula where numeric constants, either in the constraints given by the predicates μ or in the time intervals of the temporal operators, can be replaced by symbolic parameters.

An STL formula is obtained by pairing a PSTL formula with a valuation function that assigns a value to each symbolic parameter. For example, consider the PSTL formula $\varphi(\pi, \tau) = \Box_{[0,\tau]} x > \pi$, with symbolic parameters π (scale) and τ (time). The STL formula $\Box_{[0,10]} x > 1.2$ is an instance of φ obtained with the valuation $v = \{\tau \mapsto 10, \ \pi \mapsto 1.2\}$.

3.3 Defects and Faults

A controller is usually designed to meet certain *goals*. For example, the Cal Climber controller should be able to navigate around obstacles and climb hills. To talk about grading and feedback generation, we introduce some relevant terminology from the fault testing and diagnosis literature.

Definition 4. (Defect, symptom and fault) Given a controller and an environment with some desired goals,

- A *defect* is a bug in the controller implementation that leads to failure in meeting goals;
- A *symptom* is an interesting pattern in a simulation trace of the controller-environment composition that can be characterized, for example, using STL, and
- A *fault* is a symptom that is present in a trace as a result of some defect in the controller.

A general symptom, such as the inability to meet an end-to-end correctness goal (for example, obstacle avoidance), is a fault that could be the result of multiple defects in the controller. On the other hand, certain specific faults could be mapped to specific kinds of defects. As an example, consider an obstacle avoidance strategy for the Cal Climber controller, implemented in a language like C. The strategy states that every time the bump sensor signal indicates a bump, the robot backs up, moves some distance to either right or left and then re-orients by turning in-place until the heading direction is same as the original direction angle_0. A controller will check the guard $|\mathsf{angle(t)} - \mathsf{angle}_0| \leq \epsilon$ for some small $\epsilon > 0$ to determine when to stop turning in the re-orientation mode. A defect can be introduced by replacing this guard by the exact equality check $\mathsf{angle(t)} == \mathsf{angle}_0$. This modification usually leads to failure in practice, because the controller implementation polls its sensors at certain intervals, and therefore, it is highly unlikely that the sensor value at some polled time t, $\mathsf{angle}(t)$, will be exactly angle_0. The fault resulting from this defect is that in the re-orientation mode, the robot keeps turning in-place while making full circles multiple times. We call this the *circle* fault and will revisit it again in the paper.

The ability to classify traces that present a fault from those that don't is important for auto-grading. Using this classification, we can not only separate correct solutions from incorrect ones but also generate diagnostic feedback for failed traces by monitoring for relevant faults that will likely correspond to known defects.

4. FORMALISM AND APPROACH

We now formally define the auto-grading problem, the technical challenge in synthesizing a constrained parametrized set of tests, and our approach to solve this problem.

For the purpose of examples in this section, we always assume the controller is a Cal Climber program and the environment is an arena with one robot, multiple obstacles and fixed inclines (flat rectangular planks) placed on level ground. Positions in the arena are given using x, y, and z coordinates (in meters). Orientation in the $x - y$ plane is given by the yaw angle varying from -180 to 180 degrees, increasing in counter-clockwise direction with 0 aligned with y-axis. The initial position and orientation of the robot is also a part of the environment.

4.1 Constrained Parametrized Tests

One of the fundamental notions for auto-grading is that of a test.

Definition 5. (Test) A pair (E, φ) of an environment E and an STL formula φ is called a *test*. A test *passes* for a controller C if and only if $\mathsf{sim}(C, E) \models \varphi$.

For the end-to-end correctness property (goal), we will employ the convention that the STL formula φ in a test for this goal is the *negation* of the property that we want to hold. In other words, if a test "passes," it actually means that the correctness property did not hold for that test case. The reason for this convention is that it allows us to treat STL formulas encoding correctness goals and fault symptoms in a symmetric fashion, something that is required for the main technical results of this paper. Hereafter we will treat the STL property as specifying a fault unless explicitly stated otherwise.

Example 1. Consider an environment E_0 with a square obstacle occupying the region $[4.5, 5.5] \times [5.0, 5.5]$. The initial position of the robot is $\langle 5.0, 4.9 \rangle$ and the initial orientation is 0. Consider the STL property $\varphi = \Box(\mathsf{pos}.y \leq 5.5)$ which states that the robot is never able to reach a point with y coordinate more than 5.5. If the test (E_0, φ) passes, we can assert that the robot did not meet the goal of being able to avoid the obstacle.

Consider a vector of symbolic parameters $\mathbf{p} = (p_1, p_2, \cdots, p_n)$. A valuation function v maps each symbolic parameter to a concrete value (for example, in $\mathbb{R}^n$) and $v(p_i)$ denotes the value of parameter p_i in v. The set of all possible valuations of $\mathbf{p}$, its domain, is $\mathfrak{U}$.

Definition 6. (Parametrized Test) A *parametrized* environment is an environment with unknown parameters, denoted $E(\mathbf{p})$. A *parametrized test* $\Gamma(\mathbf{p}) = (E(\mathbf{p}), \varphi(\mathbf{p}))$ is a pair of a parametrized environment $E(\mathbf{p})$ and a PSTL formula $\varphi(\mathbf{p})$. Given any valuation $v \in \mathfrak{U}$, $\Gamma(v(\mathbf{p})) = (E(v(\mathbf{p})), \varphi(v(\mathbf{p})))$ is a *concrete test.*

Example 2. Consider the same environment E_0 from Example 1 except that the initial orientation of the robot is an unknown parameter θ_{init} that can take one of two possible values $\{-45, 45\}$. (See Figure 2a.) Consider the PSTL property $\varphi_0(\pi) = \Box(\mathsf{pos}.y > 5.5 \Rightarrow \pi_l < \mathsf{pos}.x < \pi_u)$, where $\pi = (\pi_l, \pi_u)$, with unknown parameters π_l and π_u that can take one of three possible values $\{-\infty, 5.0, \infty\}$ each. The property states that if the robot is able to get around the obstacle and reach a point $\mathsf{pos}.y > 5.5$, then $\mathsf{pos}.x$ is always in the interval (π_l, π_u). The pair $\Gamma_0(\theta_{init}, \pi) = (E_0(\theta_{init}), \varphi_0(\pi))$ is an example of a parameterized test.

Definition 7. (Satisfaction Region) The satisfaction region $\Omega(C, \Gamma(\mathbf{p}))$ of a controller C on a parametrized test $\Gamma(\mathbf{p})$ is the set of all valuations v of $\mathbf{p}$ such that $\Gamma(v(\mathbf{p}))$ passes for C, i.e., $\Omega(C, \Gamma(\mathbf{p})) = \{v \in \mathfrak{U} | \Gamma(v(\mathbf{p})) \text{ passes for } C\}$.

Definition 8. (Test Bench) Given a parameterized test $\Gamma(\mathbf{p})$ and a set of valuations $\rho \subseteq \mathfrak{U}$, the pair $(\Gamma(\mathbf{p}), \rho)$ is called a *constrained parametrized test*, simply referred to as *test bench*. The set of valuations ρ is called the *sub-domain* of the test bench. We say that test bench $(\Gamma(\mathbf{p}), \rho)$ *succeeds* for a controller C iff there exists a $v \in \rho$ such that $\Gamma(v(\mathbf{p}))$ passes for C or equivalently, $\Omega(C, \Gamma(\mathbf{p})) \cap \rho$ is non-empty.

Since a test bench typically includes both the goal properties (determining whether a student controller is correct or not) and the fault properties (determining the mistakes the student made), the crux of the auto-grading problem is to *synthesize a test bench* that can *accurately* classify an "unlabeled" controller as correct/incorrect and with the fault(s), if any. Treating goal and fault properties uniformly, we seek to synthesize a test bench to classify whether an unlabeled controller exhibits faulty behaviors.

To auto-grade, for every known fault, we create a test bench. If the test bench succeeds for an unlabeled controller, we can conclusively label it as one exhibiting faulty behavior. The sub-domain of a test bench essentially identifies the set of tests that indicate the presence of the fault. As mentioned earlier, a test bench can also be used in a similar way to check if a given controller meets goal requirements by formulating the failure to meet the goal as a fault.

Example 3. Consider the parameterized test Γ_0 from Example 2. Consider the sub-domain $\rho_0 = \{[\theta_{init} \mapsto 45, \pi \mapsto (5.0, \infty)], [\theta_{init} \mapsto -45, \pi \mapsto (-\infty, 5.0)]\}$. For a controller, if either of valuations in ρ_0 leads to a test that passes, it provides good evidence that the robot is either unable to avoid the obstacle or it is not able to proceed in the initial direction. (See Figure 2a.) So the test bench $(\Gamma_0(\theta_{init}, \pi), \rho_0)$ can be used to capture this failure to meet desired goals.

Example 4. Consider an environment E_1 with a fixed incline s.t. the uphill direction is along the orientation 0. The initial location of the robot is fixed at the center of the bottom boundary of the incline. The initial orientation of the robot is a parameter $\theta_{init} \in [-180, 180]$. We wish to determine whether a given controller (in an initial orientation pointing towards the incline) fails to climb within reasonable time. This can be expressed via the STL property $\varphi_1(h, \tau) = \Box_{[0,\tau]}(\mathsf{pos}.z \leq h)$, that states that the robot is not able to reach the height h, within time τ. The parametrized test bench $\Gamma_1(\theta_{init}, h, \tau) = (E_1(\theta_{init}), \varphi_1(h, \tau))$, combined with the sub-domain $\rho_1 = \{[\theta_{init} \mapsto v_{\theta_{init}}, h \mapsto v_h, \tau \mapsto v_\tau]\text{s.t. } |v_{\theta_{init}}| < 90 \wedge v_\tau > 60 \wedge v_h \leq 0.4\}$ can reliably capture the failure to climb to a height above 0.4 m within 60 secs for some initial orientation pointing towards the hill.

4.2 Synthesis of Test Bench Constraints

Designing a test bench for a fault involves (i) creating a parametrized test bench, and (ii) finding a sub-domain of the parameters such that it reliably captures the fault. While creating a parametrized test bench by hand is easy, in our experience manually coming up with the sub-domain is tedious. It not only requires the instructor to be a relative expert in STL and run-time verification, but also requires careful observation of traces where the fault is known to be present and not present, and a number of iterations of trial and error. On the other hand, instructors can easily come up with a set of *reference controllers*: a set $\mathcal{C}^+$ of positive-labeled controllers that are all known to exhibit the faulty behavior, and a set $\mathcal{C}^-$ of negative-labeled controllers that are all known to *not* exhibit the faulty behavior.

We define below the problem of synthesizing a sub-domain from a set $\mathcal{C}^+$ of positive-labeled controllers and a set $\mathcal{C}^-$ of negative-labeled controllers.

Problem 1. Given the following: (1) a parameterized test $\Gamma(\mathbf{p})$ with a domain $\mathfrak{U}$ for parameters $\mathbf{p}$, and (2) two sets $\mathcal{C}^+$ and $\mathcal{C}^-$ of controllers. Synthesize a sub-domain $\rho \subseteq \mathfrak{U}$ s.t. test bench $(\Gamma(\mathbf{p}), \rho)$ does not succeed for any $C \in \mathcal{C}^-$ and succeeds for all $C \in \mathcal{C}^+$.

We can see that any sub-domain that does not intersect with $\Omega(C, \Gamma(\mathbf{p}))$ for any $C \in \mathcal{C}^-$ and has a non-empty intersection with $\Omega(C, \Gamma(\mathbf{p}))$ for every $C \in \mathcal{C}^+$ satisfies the requirements in

Problem 1. From amongst all these possibilities, we choose the following (also illustrated in Figure 2b)

$$\rho = \bigcup_{C \in \mathcal{C}^+} \Omega(C, \Gamma(\mathbf{p})) \setminus \bigcup_{C \in \mathcal{C}^-} \Omega(C, \Gamma(\mathbf{p})) \qquad (1)$$

For convenience, we use $\Omega(\mathcal{C}^+, \Gamma(\mathbf{p}))$ (and $\Omega(\mathcal{C}^-, \Gamma(\mathbf{p}))$) to refer to $\bigcup_{C \in \mathcal{C}^+} \Omega(C, \Gamma(\mathbf{p}))$ (and $\bigcup_{C \in \mathcal{C}^-} \Omega(C, \Gamma(\mathbf{p}))$). The rationale behind this choice of ρ is two-fold:

1. To increase *coverage* of fault detection for unlabeled controllers, we wish to include as much of $\Omega(\mathcal{C}^+, \Gamma(\mathbf{p}))$ in ρ as possible because every parameter valuation in that set corresponds to a test that passed on some positively-labeled controller, i.e. a controller that exhibits the faulty behavior.
2. For the tests corresponding to valuations that are not in either one of $\Omega(\mathcal{C}^+, \Gamma(\mathbf{p}))$ or $\Omega(\mathcal{C}^-, \Gamma(\mathbf{p}))$, we choose a *lenient* grading route and do not include them in ρ. This means that if an unlabeled controller does not pass on any test that lies in $\Omega(\mathcal{C}^+, \Gamma(\mathbf{p}))$, it will not be labeled as one exhibiting the fault. This is how instructors often grade labs in practice, i.e., if tests conclude that a solution may or may not be faulty, it is considered to be non-faulty, pending a more detailed manual review. Here we are also assuming that we have a good range of positive and negative labeled controllers that cover a wide variety of ways in which the fault may or may not be exhibited.

To generate ρ as in Eqn. 1, we compute Ω, as discussed next.

4.3 Computing the Function Ω

Given a controller C and a parametrized test $\Gamma(\mathbf{p})$ with $\mathbf{p} = (p_1, p_2, \cdots, p_k)$, we wish to compute $\Omega(C, \Gamma(\mathbf{p}))$. We assume that all parameters are numerical. Every parameter that is not finite valued is discretized by sampling uniformly at some granularity within reasonable lower and upper bounds. By this construction, the domain $\mathfrak{U}$ is now a finite k-dimensional array and can be written as a Cartesian product of finite sets $\mathfrak{U}_1 \times \mathfrak{U}_2 \times \cdots \times \mathfrak{U}_k$, where p_i takes values in the set $\mathfrak{U}_i$. We assume some indexing on each $\mathfrak{U}_i$ such that $\mathfrak{U}[j_1, j_2, \cdots, j_k]$ refers to the element of $\mathfrak{U}$ formed by picking the j_i-th element from each $\mathfrak{U}_i$. Moreover, we assume that this indexing is consistent with the natural order defined over each $\mathfrak{U}_i$ (i.e., a lower index implies a smaller value). Let $N = \max_i(|\mathfrak{U}_i|)$. The size of $\mathfrak{U}$ is $\mathcal{O}(N^k)$. Given this representation of $\mathfrak{U}$, $\Omega(C, \Gamma(\mathbf{p}))$ can be represented by a k-dimensional bit-array, such that, $\Omega(C, \Gamma(\mathbf{p}))[j_1, j_2, \cdots, j_k] = 1$ iff the test $\Gamma(\mathfrak{U}[j_1, j_2, \cdots, j_k](\mathbf{p}))$ passes on the test $\Gamma(\mathfrak{U}[j_1, j_2, \cdots, j_k](\mathbf{p}))$ passes on C. The most naïve way to compute $\Omega(C, \Gamma(\mathbf{p}))$ is to perform the test $\Gamma(v(\mathbf{p}))$ for every valuation $v(\mathbf{p}) \in \mathfrak{U}$. We describe a more efficient approach to do this in cases where the test bench is monotonic in one or more parameters.

Definition 9. (Monotonicity) Given an order $\preccurlyeq$ on a parameter p_i in the parameter vector $\mathbf{p} = (p_1, p_2, \cdots, p_k)$, a parameterized test $\Gamma(\mathbf{p})$ is *monotonic* in p_i if for every controller C

$$\begin{aligned} \forall v, v' \quad & v(p_i) \preccurlyeq v'(p_i), \forall j \neq i \cdot v'(p_j) = v(p_j) \\ & \Gamma(v(\mathbf{p})) \text{ passes for } C \Rightarrow \Gamma(v'(\mathbf{p})) \text{ passes for } C \end{aligned} \qquad (2)$$

Example 5. Consider the parameterized test $\Gamma_1(\theta_{init}, h, \tau)$ from Example 4. Consider the order $\leq$ over h and two values $v_h \leq v'_h$. For any controller C, if $\Gamma_1(v_{\theta_{init}}, v_h, v_\tau)$ passes, it means that the $\mathsf{pos}.z$ always stays below v_h for the time interval $[0, v_\tau]$, which implies that it stays below v'_h as well and hence $\Gamma_1(v_{\theta_{init}}, v'_h, v_\tau)$ will pass. Thus $\Gamma_1(\theta_{init}, h, \tau)$ is monotonic in h.

Similarly, for the order $\geq$ on the parameter τ and two values $v_\tau \geq v'_\tau$, if a test $\Gamma_1(v_{\theta_{init}}, v_h, v_\tau)$ passes for any controller, it means that the $\mathsf{pos}.z$ always stays below v_h for the time interval $[0, v_\tau]$, which implies that the same is true for the time interval $[0, v'_\tau]$ and hence the test $\Gamma_1(v_{\theta_{init}}, v_h, v'_\tau)$ will also pass.

We can extend the definition of monotonicity to sets of parameters by defining required orders on tuples of parameter values. For example, $\Gamma_1(\theta_{init}, h, \tau)$ is monotonic in (h, τ) if we consider $\preccurlyeq$ as the order, where $(v_h, v_\tau) \preccurlyeq (v'_h, v'_\tau)$ iff $v_h \leq v'_h$ and $v_\tau \geq v'_\tau$. Note that we do not need separate monotonically increasing and decreasing parameterized tests since we can always invert the order on the parameter and keep the definition consistent.

Note that the definition of monotonicity allows a parameterized test to be monotonic in environment parameters but, so far in practice we have never encountered cases when this happens. Checking that a parameterized test is monotonic in certain parameters that only occur in the PSTL part of the test can be done by reduction to satisfiability modulo theories (SMT) as described in more detail by Jin et al. [11]. This is an offline step carried out at the time of design of a parameterized test.

Definition 10. (Monotone Bit-Array) For two indices $\mathbf{j} = [j_1, j_2, \cdots, j_k]$ and $\mathbf{j}' = [j'_1, j'_2, \cdots, j'_k]$ of a k-dimensional bit-array A, we say $\mathbf{j} \leq \mathbf{j}'$ iff $j_1 \leq j'_1, j_2 \leq j'_2, \cdots, j_k \leq j'_k$. The array A is said to be *monotone* if for any indices $\mathbf{j}$ and $\mathbf{j}'$ s.t. $\mathbf{j} \leq \mathbf{j}'$, $A[\mathbf{j}] = 1$ implies that $A[\mathbf{j}'] = 1$.

We now describe how monotonicity proves to be a useful property to efficiently compute $\Omega(C, \Gamma(\mathbf{p}))$. First consider the case when $\Gamma(\mathbf{p})$ is monotonic in all k parameters $p_1, p_2, \cdots, p_k$. Owing to monotonicity, we can index the valuations using their respective orders s.t. for any controller C, the k-dimensional bit-array representation of $\Omega(C, \Gamma(\mathbf{p}))$ is monotone. We describe an algorithm to compute $\Omega(C, \Gamma(\mathbf{p}))$ in three separate cases.

4.3.1 Case: k=1

For the single parameter p_1 we can perform a binary search within its domain to determine the index b such that $\Gamma(\mathfrak{U}[j_1 = b](\mathbf{p}))$ does not pass on C while $\Gamma(\mathfrak{U}[j_1 = b+1])$ passes. We would have to perform $\mathcal{O}(\log N)$ tests.

4.3.2 Case: k=2

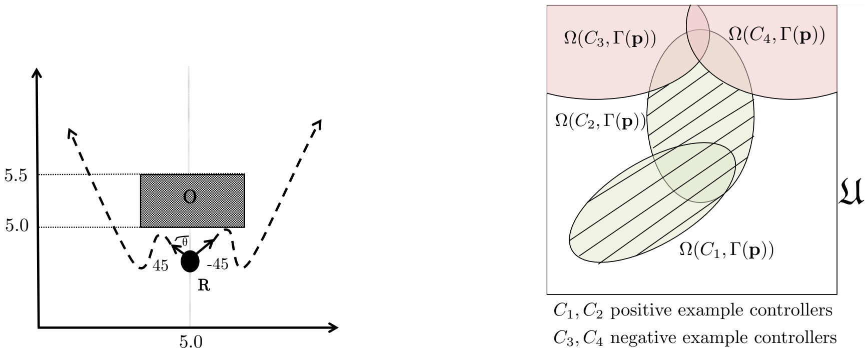
For two parameters p_1 and p_2, say we have the 2-d array of indices $[1 \cdots U] \times [1 \cdots V]$. We start at the index $\langle row = 1, col = V \rangle$. At each step we perform the test $\Gamma(\mathfrak{U}[j_1 = row, j_2 = col](\mathbf{p}))$ on C. If the test passes, we mark the complete column $\Omega(C, \Gamma(\mathbf{p}))[j_1 \geq row, j_2 = col]$ with 1s (we can do this because of monotonicity) and decrement col by 1. If the test does not pass, we mark the complete row $\Omega(C, \Gamma(\mathbf{p}))[j_1 = row, j_1 \leq col]$ with 0s and increment row by 1. We do this until we have covered the whole array. We would have to perform $\mathcal{O}(\max(U, V)) = \mathcal{O}(N)$ tests since we mark out a complete row or column after every test. Figure 3a shows an intermediate step in a run of this algorithm.

4.3.3 Case: k≥3

For more than 2 parameters, we enumerate over all possible valuations of first $k-2$ parameters and use the case for $k = 2$ for the 2-d sub-array obtained by fixing $p_1, p_2, \cdots, p_{k-2}$. We would have to perform $\mathcal{O}(N^{k-1})$ tests. We cannot hope to do (asymptotically) better than this as it is shown in [18] that searching in a monotone d-dimensional array where each dimension is of size at most n is lower bounded by $c_2(d)n^{d-1}$, where $c_2(d) = \mathcal{O}(d^{\frac{-1}{2}})$ for $d \geq 2$.

For the general case, let $\Gamma(\mathbf{p})$ be non-monotonic in the first $k-d$ parameters and monotonic in the d others. We enumerate over all possibilities of the first $k-d$ parameters and apply the algorithm for monotonic parameters to the d dimensional sub-array obtained by fixing $p_1, p_2, \cdots, p_{k-d}$.

Using the above approach, we can compute $\Omega(\mathcal{C}^+, \Gamma(\mathbf{p}))$, $\Omega(\mathcal{C}^-, \Gamma(\mathbf{p}))$ and $\rho = \Omega(\mathcal{C}^+, \Gamma(\mathbf{p})) \setminus \Omega(\mathcal{C}^-, \Gamma(\mathbf{p}))$, all represented in the form of k-dimensional bit-arrays.


5.5
5.0
O
θ
45
-45
R
5.0
$\Omega(C_3, \Gamma(\mathbf{p}))$
$\Omega(C_4, \Gamma(\mathbf{p}))$
$\Omega(C_2, \Gamma(\mathbf{p}))$
$\mathfrak{U}$
$\Omega(C_1, \Gamma(\mathbf{p}))$
C_1, C_2 positive example controllers
C_3, C_4 negative example controllers

(a) (b)

Figure 2: (a) Environment E_0 from Examples 1, 2, and 3 with robot R and obstacle O. The two trajectories shown by dotted lines meet the goals for the cases $\theta = 45$ and $\theta = -45$. (b) The hatched region is the sub-domain ρ obtained from satisfaction regions of positive and negative controller examples.

4.4 Adequate Test Samples for Grading

Checking whether a new controller C succeeds on a test bench $(\Gamma(\mathbf{p}), \rho)$ amounts to searching for a valuation in ρ such that $\Gamma(v(\mathbf{p}))$ passes for C. The naive approach to solve the search problem is to enumerate all valuations in ρ. We describe a more efficient search strategy when $\Gamma(\mathbf{p})$ is monotonic in one or more parameters.

Definition 11. (Adequate Test Sample) An *adequate test sample* $\alpha \subseteq \rho$ is a set of valuations s.t. for any controller C, $(\Gamma(\mathbf{p}), \rho)$ succeeds on C iff there is at least one $v \in \alpha$ for which $\Gamma(v(\mathbf{p}))$ passes for C.

Definition 12. (Corner) A *corner* in a monotone k-dimensional bit-array A is an index $\mathbf{j} = [j_1, j_2, \cdots, j_k]$ s.t. $A[\mathbf{j}] = 0$ and $\forall 1 \leq l \leq k$, if the index $[j_1, j_2, \cdots, j_l + 1, \cdots, j_k]$ lies within the bounds of A, then $A[j_1, j_2, \cdots, j_l + 1, \cdots, j_k] = 1$.

First consider the case when a parameterized test $\Gamma(\mathbf{p})$ is monotonic in all parameters $\mathbf{p} = (p_1, p_2, \cdots, p_k)$. Say we have computed $\Omega(\mathcal{C}^+, \Gamma(\mathbf{p}))$, $\Omega(\mathcal{C}^-, \Gamma(\mathbf{p}))$ and $\rho = \Omega(\mathcal{C}^+, \Gamma(\mathbf{p})) \setminus \Omega(\mathcal{C}^-, \Gamma(\mathbf{p}))$ in k-dimensional bit-array form.

PROPOSITION 1. *The set α comprising of all valuations $\mathfrak{U}[\mathbf{j}]$ s.t. $\mathbf{j}$ is a corner of $\Omega(\mathcal{C}^-, \Gamma(\mathbf{p}))$ and $\Omega(\mathcal{C}^+, \Gamma(\mathbf{p}))[\mathbf{j}] = 1$, is a minimal adequate test sample for $(\Gamma(\mathbf{p}), \rho)$.*

PROOF. We first show that α is adequate then we show α is also minimal. For this proof, we refer to $\Omega(\mathcal{C}^+, \Gamma(\mathbf{p}))$ by Ω^+ and $\Omega(\mathcal{C}^+, \Gamma(\mathbf{p}))$ by Ω^-.

Assume $\Gamma(v(\mathbf{p}))$ passes for C for some $v \in \alpha$. Let the index of this valuation be $\mathbf{j}_v$. By definition of α, $\Omega^+[\mathbf{j}_v] = 1$ and $\mathbf{j}_v$ is a corner of Ω^- implying $\Omega^-[\mathbf{j}_v] = 0$. From the way we have defined ρ, we can say that $\rho[\mathbf{j}_v] = 1$ or $v \in \rho$ which means $(\Gamma(\mathbf{p}), \rho)$ succeeds for C. For reverse implication, assume $(\Gamma(\mathbf{p}), \rho)$ succeeds for C, it means that it is possible to find an index $\mathbf{j} = [j_1, j_2, \cdots, j_k]$ s.t. $\mathfrak{U}[\mathbf{j}] \in \rho$ (equivalently, $\rho[\mathbf{j}] = 1$) and $\Gamma(v(\mathbf{p}))$ passes for C (equivalently, $\Omega(C, \Gamma(\mathbf{p}))[\mathbf{j}] = 1$). Since $\mathbf{j} \in \rho$, we have $\Omega^+[\mathbf{j}] = 1$ and $\Omega^-[\mathbf{j}] = 0$. If $\mathbf{j}$ is a corner of Ω^-, then we have $\mathfrak{U}[\mathbf{j}] \in \alpha$ and we are done. If not, then there exists $1 \leq l \leq k, \mathbf{j}' = [j_1, j_2, \cdots, j_l + 1, \cdots, j_k]$ s.t. $\Omega^-[\mathbf{j}'] = 0$. By monotonicity, we also have $\Omega^+[\mathbf{j}'] = 1$ and $\Omega(C, \Gamma(\mathbf{p}))[\mathbf{j}'] = 1$. If $\mathbf{j}'$ is a corner of Ω^-, then $\mathfrak{U}[\mathbf{j}] \in \alpha$ and we are done. Else we set $\mathbf{j}$ to $\mathbf{j}'$ and proceed again in the same way. Since $\mathfrak{U}$ is finite, this procedure is guaranteed to terminate at a corner of Ω^-.

To show minimality, we remove some arbitrary valuation v from α and show that it becomes inadequate. Say $\mathbf{j}_v$ is the index corresponding to v. Consider a controller C s.t. $\Omega(C, \Gamma(\mathbf{p}))[\mathbf{j}] = 1$ iff $\mathbf{j} \geq \mathbf{j}_v$. Since $\mathbf{j}_v$ is a corner of Ω^-, for every index $\mathbf{j} \neq \mathbf{j}_v$ and $\mathbf{j} \geq \mathbf{j}_v$, we have that $\Omega^-[\mathbf{j}] = 1$. This means there is no corner of Ω^- in $\Omega(C, \Gamma(\mathbf{p}))$ apart from $\mathbf{j}_v$. Hence, we will not be able to find another $v' \in \alpha, v' \neq v$ s.t. $\Gamma(v'(\mathbf{p}))$ passes on C, even though $(\Gamma(\mathbf{p}), \rho))$ succeeds on C. This means α becomes inadequate if we remove any of its elements, thus making it minimal. □

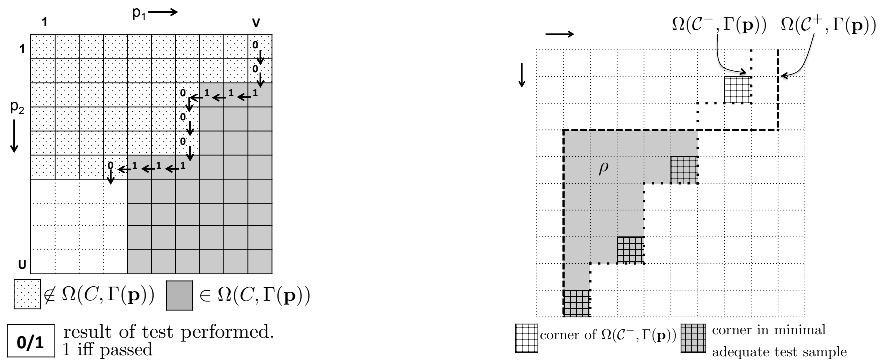
Figure 3b shows an example of a minimal adequate test sample for the 2-d case. To compute α, similar to Sec. 4.3; in case $k = 1$, we can do a binary search to find the corner; in case $k = 2$, we can we can find corners by starting at the boundary of the 2-d array and eliminating rows and columns; and in case $k \geq 3$, we can enumerate over first $k - 2$ parameters and apply the case for $k = 2$ on the rest. For the general case of $k - d$ non-monotonic and d monotonic parameters, we enumerate over all possibilities of first $k - d$ parameters, and keep accumulating the adequate test sample calculated for the d-dimensional monotone sub-array obtained by fixing the first $k - d$ parameters.

We conclude this section with a remark about an alternative mathematical formulation. If we treat a monotone bit-array as a partially-ordered set (poset) $\mathfrak{O}$, then, the satisfaction region $\Omega(C, \Gamma(\mathbf{p}))$ of some controller C is an upward closed subset of $\mathfrak{O}$. The sub-domain ρ is now the intersection of an upward-closed (Ω^+) and another downward-closed set ($\mathfrak{U} \setminus \Omega^-$). With some effort, we can show that the minimal adequate test sample α corresponds to the maximal elements of ρ. However, we find the monotone bit-array formulation more useful for our purposes because it is a special case of a poset that allows for efficient algorithms (as given in Sec. 4.3.1 and 4.3.2) for computation of α, which is not obvious with the general poset formulation.

5. EXPERIMENTAL RESULTS

We designed and experimentally evaluated our auto-grader using a collection of solutions implemented by 50 groups of students as part of the laboratory component of the Fall 2013 instance of the EECS 149 class at UC Berkeley.

The code was anonymized and collected automatically using post-build commands so that each group provided a variable number of versions, most of which being intermediate non-final solutions. The lab was organized in two sessions, one focusing on the obstacle avoidance problem, and another focusing on the hill climbing. In this section, we describe the set of test benches that we used to establish diagnostics with respect to each goal. For each test bench, we first manually label a set T of 100 randomly selected student solutions. We select 30 solutions out of the 100 while maintain-



(a) (b)

Figure 3: (a) An intermediate step in a run of the algorithm used to compute $\Omega(C, \Gamma(\mathbf{p}))$ for two monotonic parameters $\mathbf{p} = (p_1, p_2)$. The arrows indicate the tests that are performed. Monotonicity allows us to compute whole of $\Omega(C, \Gamma(\mathbf{p}))$ by performing $\mathcal{O}(\max(U, V))$ tests. (b) For the case of two monotonic parameters (increasing in the directions shown by arrows), the dashed (and dotted) lines represent the boundary between cells containing 0s and 1s for $\Omega(\mathcal{C}^+, \Gamma(\mathbf{p}))$ (and $\Omega(\mathcal{C}^-, \Gamma(\mathbf{p}))$). The shaded part is ρ. The hatched cells are corners of $\Omega(\mathcal{C}^-, \Gamma(\mathbf{p}))$ and the shaded hatched cells comprise the minimal adequate test sample.

ing balance between the number of positive and negative examples which are input to the synthesis algorithm. To elaborate, if we have more than 15 each of positive and negative examples (say 45 positive and 55 negative) then we select some 15 examples of each type arbitrarily. If either one of positive or negative examples is less than 15 (say 5 positive and 95 negative), then we select all instances of the type of example that is scarce and select the remainder of the 30 from the other type (in the example, we will take 5 positive and 25 negative). This is a standard technique in machine learning done to improve coverage and reduce bias in case a fault is rare [4]. In Sec. 5.1 and 5.2, for each test bench, we describe (1) the fault symptom and the corresponding PSTL formula, (2) environment and STL parameters, and their monotonic nature, (3) synthesized sub-domain and adequate test sample, and (4) synthesis time per training example. In Sec 5.3, we measure *accuracy* of the grader by comparing labels generated by the auto-grader against another set of manually graded solutions (disjoint from T). We also demonstrate *efficiency* in terms of the average grading time per solution.

Experiments are performed using a single core of a 2.3 GHz processor with 8 GB of memory. Since more than one tests share the same environment configuration, we run simulations for all solutions in all the environment configurations as needed for our evaluation in a pre-processing step and store traces to files. Each simulation is run for 60 secs of virtual time with a step size of 5 ms which takes about 10 secs of system time. For each test bench, in Sec. 5.1 and 5.2, we report running times of the synthesis algorithm that computes the sub-domain and the adequate test sample, and in Sec. 5.3, we report running times of the auto-grader which checks for existence of a passing test in the adequate test sample. These running times do not include time required for simulation since we are reading traces from files. When using the auto-grader in loop with the simulator, we need one simulation for every environment in each test bench per solution (the aggregate is lower in practice because more than one test benches share the same environment). All simulations are run using NI Robotics Simulator. STL monitoring is performed using Breach [5]. The synthesis modules and grading software with an extended library of faults is made available at `http://www.eecs.berkeley.edu/~garvitjuniwal/MOOC149.html`.

5.1 Obstacle Avoidance

In assessing faults in obstacle avoidance, we use an environment $E_3(\theta_{init})$ which contains an obstacle occupying the region

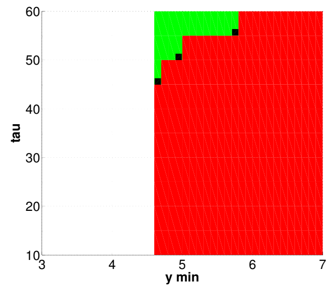


Figure 4: Test bench avoid_front. Green (lightly shaded) region is the computed sub-domain. Red (dark shaded) region is the set of tests excluded from the sub-domain because they are triggered on at least one negative example. White (unshaded) region is the set of tests that are not triggered on any negative or positive example. Little black squares are the points in the adequate test sample.

$[4.5, 5.5] \times [5.0, 5.5]$. Initial position of the robot is $(5.0, 4.9)$. The parameter θ_{init} encodes the initial orientation of the robot.

5.1.1 *Failing simple obstacle avoidance (*avoid_front*)*

This test bench checks whether the robot can get past the obstacle when started with the initial orientation $\theta_{init} = 0$, facing the obstacle directly.

- Parameterized Test: $(E_3(0), \varphi_{\text{orient}})$ with $\varphi_{\text{orient}} = \Box_{[0,\tau]}(\mathsf{pos}.y < y_{min})$. If φ_{orient} is satisfied for suitable values of τ and y_{min}, it indicates failure to avoid the obstacle.
- Parameters: (τ, y_{min})
- Domain:[1] $(\tau, y_{min}) \in \{60 : -5 : 10\} \times \{3.0 : 0.1 : 7.0\}$
- Monotonicity: τ monotonic for $\geq$ and y_{min} monotonic for $\leq$.
- Synthesized sub-domain: See Figure 4
- Adequate Test Sample: $\{(60, 5.7), (55, 4.9), (50, 4.6)\}$
- Average synthesis time per training example: 1.9 sec

[1] The notation $\{a : d : b\}$ denotes the set $\{a, a+d, a+2d, \cdots, a+kd\}$, where k is the greatest integer s.t. if $d \geq 0$ then $a + kd \leq b$ else if $d < 0$ then $a + kd \geq b$

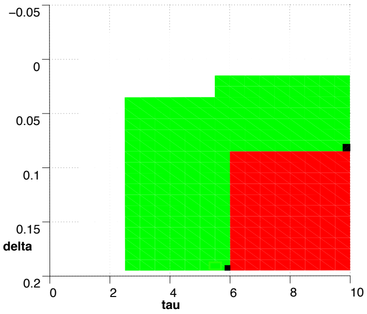


Figure 5: Test bench circle

5.1.2 *Failing re-orienting after obstacle avoidance (avoid_left/avoid_right)*

This test bench checks whether the robot can get past the obstacle and keep heading in the initial heading direction. We perform the test in two possible initial orientations; facing left ($\theta_{init} = 45$) or right ($\theta_{init} = -45$). We show details for the case $\theta_{init} = 45$.

- Parameterized Test: $(E_3(45), \varphi_{\text{reorient}})$ with $\varphi_{\text{reorient}} = \Box_{[0,\tau]} (\text{pos}.y < y_{min} \vee \text{pos}.x > x_{max})$. If $\varphi_{\text{reorient}}$ is satisfied for suitable values of τ, x_{max} and y_{min}, it indicates either failure to avoid the obstacle or failure to re-orient in the correct heading direction.
- Parameters: (τ, y_{min}, x_{max})
- Domain: $(\tau, y_{min}, x_{max}) \in \{60 : -5 : 10\} \times \{3.0 : 0.1 : 7.0\} \times \{6.0 : -0.1 : 3.0\}$
- Monotonicity: τ monotonic for $\geq$; y_{min} monotonic for $\leq$ and x_{max} monotonic for $\geq$.
- Synthesized sub-domain: Due to more than 2 parameters, it is not possible to show it in a figure.
- Adequate Test Sample: {(60, 5.4, 4.2), (55, 5.4, 5.0), (50, 4.8, 5.8), (10, 4.4, 5.8)}
- Average synthesis time per training example: 26.2 sec

5.1.3 *Strict equality check (circle)*

This test bench investigates the circle fault mentioned in Section 3.3. The purpose of the test is to detect that at some time instant t_0, the robot bumps into the obstacle, then turns about itself with a maximum period of τ, while remaining close to its position at t_0 with a margin of δ.

- Parameterized Test: $(E_3(0), \varphi_{\text{circle}})$ $\varphi_{\text{circle}}(t_0, \delta, \tau) = \Diamond(\varphi_{\text{bump}}(t_0) \wedge \Diamond_{[0,2\tau]}(\varphi_{\text{fullturn}}(t_0, \delta)))$ where $\varphi_{\text{bump}}(t_0) = \text{bump}(t_0) \equiv TRUE$ and $\varphi_{\text{fullturn}}$ is given by $\varphi_{\text{fullturn}}(t_0, \delta, \tau) = (\varphi_{\theta\sim 0} \wedge \varphi_{\text{close}}(t_0, \delta) \mathbf{U}_{[0,\tau]} (\varphi_{\theta\sim 180} \wedge \varphi_{\text{close}}(t_0, \delta) \mathbf{U}_{[0,\tau]} \varphi_{\theta\sim 0}))$ where $\varphi_{\text{close}}(t_0, \delta) = \text{dist}(\text{pos}(t_0), \text{pos}) < \delta$ for some distance function dist and $\varphi_{\theta\sim 0}$ and $\varphi_{\theta\sim 180}$ assess that angle is close to 0 degrees and 180 degrees, respectively. The suitable value for the parameter t_0 can be determined by the first collision instant with the obstacle, which is common to all solutions since they all start moving forward in the same direction (say this common value is t_0). We fix t_0 to t_0.
- Parameters: (τ, δ)
- Domain: $(\tau, \delta) \in \{1 : 1 : 10\} \times \{-0.025 : 0.01 : 0.2\}$
- Monotonicity: τ monotonic for $\leq$ and δ monotonic for $\leq$
- Synthesized sub-domain: See Figure 5
- Adequate Test Sample: {(5.5, 0.195), (10.0, 0.075)}
- Average synthesis time per training example: 2.7 sec

5.2 Hill Climbing

To assess faults in the hill climbing part of the assignment, we use an environment $E_4(\beta)$ which contains a hill. The parameter β encodes the initial configuration of the robot. It can take two values B and M. In B the robot starts at the bottom of the hill facing 45 degrees rightwards of uphill and in M the robot starts on the hill (midway between bottom and top) facing downhill.

5.2.1 *Failing simple hill climb (hill_climb)*

This test bench checks whether the robot fails to reach near the top of the hill. We perform this test for both possible values of β.

- Parameterized Test: $(E_4, \varphi_{\text{hill}})$ with $\varphi_{\text{hill}} = \Box_{[0,\tau]}(\text{pos}.z \leq h)$. If φ_{hill} is satisfied for suitable values of τ and h, it indicates failure to reach near top of the hill.
- Parameters: (β, τ, h)
- Domain: $(\beta, \tau, h) \in \{B, M\} \times \{60 : -5 : 10\} \times \{-0.1 : 0.01 : 0.7\}$
- Monotonicity: τ monotonic for $\geq$ and h monotonic for $\leq$
- Synthesized sub-domain: See Figure 6
- Adequate Test Sample: {(M, 55, 0.41), (M, 50, 0.37), (M, 35, 0.35), (M, 15, 0.33), (M, 10, 0.31), (B, 55, 0.45), (B, 50, 0.34), (B, 45, 0.18), (B, 40, 0.07)}
- Average synthesis time per training example: 6.2 sec

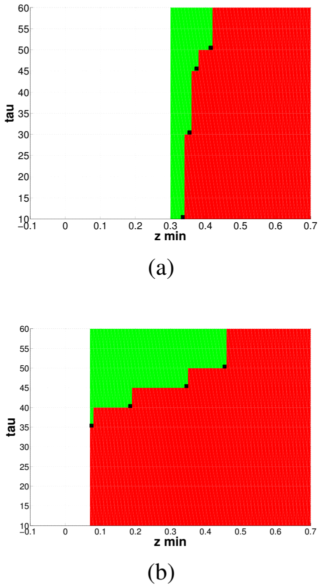


Figure 6: (a) Test bench hill_climb ($\beta = M$) (b) Test bench hill_climb ($\beta = B$)

5.2.2 *Failure to detect hill (what_hill)*

This test bench checks the failure of robot to detect when it is oh a hill. This is a specific bug which leads to failure in hill climbing. We use the environment E_4 with $\beta = B$.

- Parameterized Test: $(E_4(B), \varphi_{\text{hilldet}})$ with $\varphi_{\text{hilldet}} = \Diamond_{[0,\tau_1]}(\varphi_{\text{fwd}} \mathbf{U}_{[\tau_2,+\infty]} \varphi_{\text{cliff}})$, where φ_{fwd} assesses that the robot is moving forward and φ_{cliff} assess firing of cliff sensor. If this property is satisfied for suitable values of τ_1 and τ_2, it means that the robot keeps driving straight until it hits a cliff even if it is on a hill instead of re-orienting towards uphill direction.
- Parameters: (τ_1, τ_2)
- Domain: $(\tau_1, \tau_2) \in \{0 : 1 : 60\} \times \{60 : -1 : 0\}$
- Monotonicity: τ_1 monotonic for $\geq$ and τ_2 monotonic for $\leq$
- Synthesized sub-domain: See Figure 7
- Adequate Test Sample: {(1, 0), (41, 12), (60, 13)}

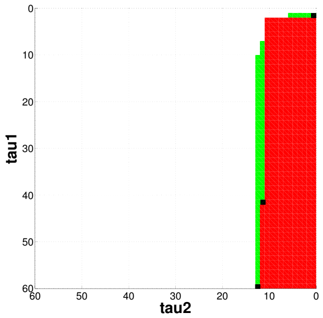


Figure 7: Test bench what_hill

- Average synthesis time per training example: 8.3 sec

5.2.3 *No filtering* (filter)

This test bench checks whether the reason for a failure to climb a hill is the absence of a low-pass filter applied to the accelerometer data to smoothen it. We check this by performing the test hill_climb with E_4 but applying a low-pass filter to the accelerometer data externally (before it is fed into the controller). If the robot is able to climb the hill with an external filter but fails to do so without it, we can conclude that absence of the filter is the bug.

5.3 Accuracy of Classification

To measure accuracy we use the synthesized test benches to label a set of student solutions (disjoint from the training set) and compare the labels assigned by the auto-grader to manually assigned labels. Table 1 shows obtained accuracy results and average running times for 8 test benches. The running times do not include time needed for simulation. For each solution, simulation in a total of 6 environment configurations is collectively needed for the 8 test benches (2 environments are shared). Note that we find a majority of solutions that are not able to meet goals but that is expected because our solution set has preliminary and intermediate versions of the solutions as well. We also find that accuracy is poorer in the hill climbing cases, which shows that variation in student solutions is higher in that part of the lab.

Test Bench	N^+	N^{++}	N^-	N^{--}	T_{avg}
avoid_front	74	74	27	27	0.119
avoid_left	78	78	23	23	0.158
avoid_right	82	82	19	19	0.148
circle	2	2	99	99	0.382
hill_climb ($\beta = B$)	49	36	345	345	0.111
hill_climb ($\beta = M$)	35	32	359	359	0.120
what_hill	220	216	174	156	0.288
filter	8	7	354	339	0.412

Table 1: N^+ is the number of solutions with fault (manually labeled). N^{++} is the number of solutions that the auto-grader correctly labeled as faulty. N^- and N^{--} are defined similarly for solutions without fault. T_{avg} is the average labeling time per solution in seconds.

5.4 Discussion

The experimental evaluation indicates that the auto-grader is both accurate and efficient. The test benches used in our evaluation capture common mistakes made by students, as observed in an on-campus offering, and even simply identifying these mistakes can be valuable feedback.

The parameter synthesis requires a set of “good” and “bad” solutions. We show that a small number of labeled examples (30) is enough to get reasonable accuracy. The overhead of providing these solutions is small: an instructor manually labels a small number of solutions by viewing the robotics simulator video and avoids the tedious process of parameter tuning.

6. RELATED WORK

Related work falls into two main categories: the use of formal methods and programming languages techniques for (online) education, and parameter synthesis for STL.

There is a growing number of efforts to incorporate formal methods into technologies for education. Singh et al. [25] present an approach to automatically generate problems in high-school algebra. Sadigh et al. [24] show how the problem of generating variants of exercises in an Embedded Systems textbook [14] can be mapped to standard problems in formal methods and apply some of these methods to classes of exercises. Singh et al. [26] present an auto-grader for a Python programming course, where, similar to the present paper, feedback is generated based on a library of common mistakes, but, differently, the technical approach uses an encoding to SAT-based program synthesis. Alur et al. [1] consider auto-grading DFA construction problems, providing a novel blend of three techniques for assigning partial grades for incorrect answers. The present work proposes different formalisms and algorithms, and represents the first auto-grader for lab assignments in the area of embedded, cyber-physical systems.

Parameter synthesis for PSTL formulas has been studied before [3, 11]. Unlike our work, these efforts seek to find specific parameter values rather than sub-domains, and are not directly usable in the auto-grading context of this paper. A symbolic approach to PSTL parameter synthesis has been discussed in [3], which reports that an enumerative approach outperforms the symbolic one.

We also note related work in the area of fault localization only using execution traces (black-box localization) [16, 17]. However, these tehcniques apply to digital systems and are not directly usable in our context of hybrid systems with continuous variables.

7. CONCLUSIONS & FUTURE WORK

In this paper, we have formalized the auto-grading problem for laboratory assignments in cyber-physical systems, and presented a formal, algorithmic approach to solve it based on parameter synthesis. The approach is general and can apply beyond the particular motivating lab setting considered here. The theoretical treatment makes no assumptions about the form of the controller, environment, and simulation model. Note also that our approach can be used with any black-box simulator.

There are several interesting directions for future work. One direction is to introduce cost or reward metrics into the model to quantify the quality of a student solution. Monitoring these metrics over a set of tests can help assign partial credit or extra credit to student solutions. For example, in a problem involving robot navigation to a goal location, a controller that gets closer, or takes less time, should intuitively receive more credit than one that does not. Additionally, for student controllers that do not satisfy the goal property, but which also do not exhibit any known fault, we need an approach to explain the student mistake, potentially by synthesizing an (P)STL formula that serves as a defining symptom.

As mentioned, the auto-grader has already been successfully deployed in an actual MOOC, EECS149.1x [15], and we plan to run user studies on its effectiveness in the future and use it in other classes and labs. One interesting topic is analog and mixed signal circuits, for which *Time Frequency Logic* (TFL [6]) could be used instead of STL.

Finally, beyond the application to education, we note that our technique can be applied to debugging problems for embedded controllers where we can assume a plausible fault model and where monotonicity holds; e.g., for industrial control systems where monotonicity of PSTL has already been found widespread [11].

8. ACKNOWLEDGEMENTS

This work was supported in part by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and by the NSF ExCAPE project (CCF-1139138).

9. REFERENCES

[1] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of DFA constructions. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, August 2013.

[2] R. Alur and T. A. Henzinger. A really temporal logic. In *Symposium on Foundations of Computer Science*, pages 164–169, 1989.

[3] E. Asarin, A. Donzé, O. Maler, and D. Nickovice. Parametric identification of temporal properties. In *Runtime Verification*, pages 147–160, 2011.

[4] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard. A study of the behavior of several methods for balancing machine learning training data. *SIGKDD Explorations*, 6(1):20–29, 2004.

[5] A. Donzé. Breach: A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In *Computer-Aided Verification*, pages 167–170, 2010.

[6] A. Donzé, O. Maler, E. Bartocci, D. Nickovic, R. Grosu, and S. A. Smolka. On temporal logic and signal processing. In S. Chakraborty and M. Mukund, editors, *ATVA*, volume 7561 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2012.

[7] J. C. Jensen. Elements of model-based design. Master's thesis, University of California, Berkeley, February 2010.

[8] J. C. Jensen, D. H. Chang, and E. A. Lee. A model-based design methodology for cyber-physical systems. In *First IEEE Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems (CyPhy)*, Istanbul, Turkey, 2011.

[9] J. C. Jensen, E. A. Lee, and S. A. Seshia. *An Introductory Lab in Embedded and Cyber-Physical Systems*. LeeSeshia.org, Berkeley, CA, 2012.

[10] J. C. Jensen, E. A. Lee, and S. A. Seshia. Virtualizing cyber-physical systems: Bringing CPS to online education. In *Proc. First Workshop on CPS Education (CPS-Ed)*, April 2013.

[11] X. Jin, A. Donzé, J. Deshmukh, and S. A. Seshia. Mining requirements from closed-loop control models. In *Hybrid Systems: Computation and Control (HSCC)*, 2013.

[12] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, 1990.

[13] E. A. Lee and S. A. Seshia. EECS 149 course website. `http://chess.eecs.berkeley.edu/eecs149`.

[14] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. LeeSeshia.org, Berkeley, CA, 2011.

[15] E. A. Lee, S. A. Seshia, and J. C. Jensen. EECS149.1x Course Website on edX. `https://www.edx.org/course/uc-berkeleyx/uc-berkeleyx-eecs149-1x-cyber-physical-1629`.

[16] W. Li, A. Forin, and S. A. Seshia. Scalable specification mining for verification and diagnosis. In *Design Automation Conference*, pages 755–760, 2010.

[17] W. Li and S. A. Seshia. Sparse coding for specification mining and error localization. In *Proceedings of the International Conference on Runtime Verification (RV)*, September 2012.

[18] N. Linial and M. E. Saks. Searching ordered structures. *J. Algorithms*, 6(1):86–103, 1985.

[19] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *FORMATS/FTRTFT*, pages 152–166, 2004.

[20] Massachusetts Institute of Technology (MIT). The iLab Project. `https://wikis.mit.edu/confluence/display/ILAB2/Home`, Last accessed: February 2014.

[21] P. Mitros, K. Afridi, G. Sussman, C. Terman, J. White, L. Fischer, and A. Agarwal. Teaching Electronic Circuits Online: Lessons from MITx's 6.002x on edX. In *International Symposium on Circuits and Systems (ISCAS)*, Beijing, China, May 2013. IEEE.

[22] L. Pappano. The Year of the MOOC. `http://www.nytimes.com/2012/11/04/education/edlife/massive-open-online-courses-are-multiplying-at-a-rapid-pace.html`, November 2012.

[23] A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[24] D. Sadigh, S. A. Seshia, and M. Gupta. Automating exercise generation: A step towards meeting the MOOC challenge for embedded systems. In *Workshop on Embedded Systems Education (in conjunction with ESWeek)*, Tampere, Finland, October 2012.

[25] R. Singh, S. Gulwani, and S. Rajamani. Automatically generating algebra problems. In *Intl. Conf. of the Association for the Advancement of Artificial Intelligence (AAAI)*, 2012.

[26] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Programming Languages Design and Implementation (PLDI)*, 2013.

[27] R. Smith. Open dynamics engine. `http://ode.org`.

APPENDIX

A. STL SEMANTICS

The formal semantics of signal temporal logic (STL) are given as follows:

Definition 13. The satisfaction of an STL formula relative to a signal $\mathbf{x}$ at time t is defined inductively as

$$\begin{array}{lll}
(\mathbf{x}, t) \models \mu & \text{iff} & \mathbf{x} \text{ satisfies } \mu \text{ at time } t \\
(\mathbf{x}, t) \models \neg\varphi & \text{iff} & (\mathbf{x}, t) \not\models \varphi \\
(\mathbf{x}, t) \models \varphi_1 \wedge \varphi_2 & \text{iff} & (\mathbf{x}, t) \models \varphi_1 \text{ and } (\mathbf{x}, t) \models \varphi_2 \\
(\mathbf{x}, t) \models \varphi_1 \, \mathbf{U}_{[a,b]} \, \varphi_2 & \text{iff} & \exists t' \in [t+a, t+b] \text{ s.t.} \\
 & & \quad (\mathbf{x}, t') \models \varphi_2 \text{ and} \\
 & & \quad \forall t'' \in [t+a, t'), (\mathbf{x}, t'') \models \varphi_1
\end{array}$$

Extension of the above semantics to other kinds of intervals (open, open-closed, and closed-open) is straightforward. We write $\mathbf{x} \models \varphi$ as a shorthand of $(\mathbf{x}, 0) \models \varphi$.