Underminer: A Framework for Automatically Identifying Non-converging Behaviors in Black Box System Models

Ayca Balkan
University of California, Los Angeles
abalkan@ucla.edu

Paulo Tabuada
University of California, Los Angeles
tabuada@ucla.edu

Jyotirmoy V. Deshmukh
Toyota Technical Center
jyotirmoy.deshmukh@toyota.com

Xiaoqing Jin
Toyota Technical Center
xiaoqing.jin@toyota.com

James Kapinski
Toyota Technical Center
jim.kapinski@toyota.com

ABSTRACT

Evaluation of industrial embedded control system designs is a time-consuming and imperfect process. While an ideal process would apply a formal verification technique such as model checking or theorem proving, these techniques do not scale to industrial design problems, and it is often difficult to use these techniques to verify performance aspects of control system designs, such as stability or convergence. For industrial designs, engineers rely on testing processes to identify critical or unexpected behaviors. We propose a novel framework called *Underminer* to improve the testing process; this is an automated technique to identify non-converging behaviors in embedded control system designs. Underminer treats the system as a black box, and lets the designer indicate the model parameters, inputs and outputs that are of interest. It supports a multiplicity of convergence-like notions, such as those based on Lyapunov analysis and those based on temporal logic formulae. Underminer can be applied in the context of testing models created in the controller-design phase, and can also be applied in a scenario such as hardware-in-the-loop testing. We demonstrate the efficacy of Underminer by evaluating its performance on several examples.

1. INTRODUCTION

Embedded control software for complex real-world systems is often designed using the model-based development (MBD) paradigm. In this paradigm, designers develop a closed-loop model of the system, which consists of a model of the physical aspects of the system as well as a model of the embedded real-time software. To check the performance of a closed-loop model, control engineers excite the model with dynamic disturbances in its environment, and visually inspect its response. Control engineers typically desire behaviors that indicate stability or *convergence* of the system. A convergent behavior can be defined informally as the property that a signal converges to a desired value within reasonable time. To label an observed behavior as convergent or non-convergent, control designers often rely on ad-hoc and subjective notions of convergence. A key drawback of such a testing procedure is that it does not allow a high degree of automation, due to the reliance on engineering insight, both to identify system parameter values or test conditions that exercise critical behaviors and to perform the labeling task. To improve the current control-design process, we would need to surmount two key challenges: (1) formal, machine-checkable definitions of convergence are sparsely used, and (2) test generation techniques are ad-hoc or manual. In this paper, we propose a new framework to address these two challenges, and thus to automate the process of efficiently evaluating control-system designs.

In some settings, control engineers employ stability analysis and Lyapunov techniques to determine whether a system design will exhibit convergent behaviors, but formally checking whether a system is asymptotically stable requires rigorous mathematical reasoning. Detailed industrial control design models are not amenable to such formal checks for two main reasons. First, the models can have high complexity, in the sense that: (a) they contain many state variables, and (b) they contain nonlinearities and implementation details such as controller output and sensor input saturation, transport delays due to computation times and communication delays, and quantization due to fixed-point number representations. Second, models are often represented in modeling frameworks that have proprietary semantics, such as Simulink® [1]. In a sense, the high complexity and opacity of the models forces us to treat closed-loop models as effectively *black box*, i.e., *the only information we can glean about the model is by exciting its inputs and observing its outputs*. Through the techniques proposed in this paper, we seek to adapt the mathematically rich techniques of stability analysis to models that are effectively black box.

We propose a two-phase framework in Section 3 for discovering non-convergent test cases. We define a test case as a set of values for input signals and parameter values to stimulate the closed-loop model and the corresponding set of output signals.

In Section 4, we discuss techniques to learn a classification function based on a set of heuristics to automatically discriminate timely convergent output behaviors from those that do not converge (or do not converge in a timely fashion). We call a function belonging to this class a *Convergence Classifier Function* (CCF). One kind of heuristic CCF that we consider is based on a real-time temporal logic. Logics such as Metric Temporal Logic (MTL) [9] and Signal Temporal Logic (STL) [19], can express common notions re-

EMSOFT'16, October 01-07 2016, Pittsburgh, PA, USA

ISBN 978-1-4503-4485-2/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2968478.2968487

lated to convergence, such as settling to a specific *settling region* with a given *settling time*. In our approach we use the function that computes the robustness of the STL formula [7] specifying the settling time requirement as a CCF. Our framework also supports CCFs based on the notion of Lyapunov stability [15]. It can be shown that a system is convergent (in the sense that it is Lyapunov-stable), if there exists a function (known as the Lyapunov function) that satisfies certain mathematical conditions. We observe that Lyapunov analysis provides a set of elegant mathematical tools to specify stable or convergent behaviors over the system state space. We utilize these tools to define a *Lyapunov-like function* (LLF) over the output space of models. We adapt two different techniques proposed in the literature for learning Lyapunov functions for dynamical systems to learn LLFs from simulation traces for the model outputs [14, 4].

In Section 5, we present the second phase of our framework. In this phase, we use the CCFs obtained in Phase I to guide the test case generation. The intention of the test generation phase is to produce test cases that informally represent the worst-case behaviors of the system from the perspective of "convergence to a reference value". Inspired by optimization-guided falsification approaches for temporal logics, such as those implemented in the tools S-TaLiRo [2] and Breach [6], our framework supports the use of stochastic global optimization for test generation. Our framework also supports a new test generation procedure based on adaptively sampling values from an implicit grid imposed on the input/parameter space. This method attempts to use local sensitivity information in a model to prioritize search in a region most likely to contain non-convergent behaviors (as defined by the chosen CCF). This method thus attempts to combine property-driven testing while achieving coverage of the search space.

We benchmark the performance of our framework on various academic examples in Sec. 6, and then apply the framework to a set of case studies of practical closed-loop control systems from the automotive domain in Sec. 7.

Related Work. The proposed test generation framework builds on the vibrant research on simulation driven analysis of dynamical systems. In [25], Topcu et al. focus on identifying invariant subsets of regions of convergence for polynomial dynamical systems. By leveraging the information from the simulation traces, authors approximate the region of convergence via the solution of a tractable optimization problem. Kapinski et al. incorporate automated reasoning tools into this idea to perform data-driven stability analysis of a more general class of systems. This was later extended by Balkan et al. for identifying contraction metrics [3]. In [4] Bobiti and Lazar present a sampling based procedure to identify finite-step Lyapunov functions. Note that the methods used to construct CCFs in this work are closely related to those in the aforementioned papers. Kozarev et al. proposed a machine learning based methodology to learn regions of attraction of dynamical systems [17]. Following the work of Kong et al. on classification of desired and undesired behaviors of dynamical systems, Jones et al. introduced a procedure for anomaly detection using simulation traces [16, 13]. Medhat et al. present a technique to generate a hybrid automaton modeling the behavior of a black box system using its simulation traces [20]. It would be interesting to evaluate such an approach in our context, but for industrial-scale systems scalability would be a challenge due to the highly complex dynamics and features such as pure delays that are inexpressible using hybrid automata. In [5, 8], the authors propose a test generation method that utilizes a statistical coverage metric called the star-discrepancy metric. Combining such coverage metrics with stability testing could be an interesting future direction.

CCF	**C**onvergence **C**lassifier **F**unction	LLF	**L**yapunov-**L**ike **F**unction
SoS	**S**um-**o**f-**S**quares	STL	**S**ignal **T**emporal **L**ogic
TG	**T**est **G**eneration	S-NM	**S**eeded **N**elder-**M**ead

Table 1: The list of abbreviations that appear frequently in the rest of the paper.

2. PRELIMINARIES

In this section, we introduce the notation and terminology used in rest of the paper. We consider a dynamical system, $\mathcal{M}$, and we assume that the system under test is a black box, that is, that we can obtain examples of behaviors of $\mathcal{M}$ without explicit general description of the dynamic behaviors (ODEs). The black box assumption allows us to consider a broad class of systems and representations. For example, $\mathcal{M}$ can be modeled in an environment that uses proprietary semantics, such as Simulink®, which does not make available an analytic description of the dynamics, or alternatively, $\mathcal{M}$ could be a physical manifestation of the system, where the output behaviors are measured as part of a physical testing activity.

For a given system $\mathcal{M}$, we assume the existence of a transition function $\Phi_{\mathcal{M}} : P \times \mathbb{R}_{\geq 0} \to Y$, where $P \subseteq \mathbb{R}^p$ is a set of parameters, and $Y \subseteq \mathbb{R}^n$ is a set of outputs. This function defines the output $y(t) = \Phi_{\mathcal{M}}(p, t)$ at a time $t \in \mathbb{R}_{\geq 0}$ given a parameter value $p \in P$. Note that P can be used to represent internal system parameters, such as mechanical damping coefficients and thermal constants, but we can also use it to specify the initial conditions of the system or finitely parameterized exogenous input signals. The transition function can be used to generate output traces for $\mathcal{M}$. Table 1 lists the abbreviations that appear frequently throughout the paper.

Definition 1 (Output Trace). *An output trace of $\mathcal{M}$ under parameters p is a function $y_p : \mathcal{T} \to \mathbb{R}^n$, where $\mathcal{T} = (t_1, \ldots, t_T)$ is a finite set of strictly increasing, non-negative real values, and $y_p(t) = \Phi_{\mathcal{M}}(p, t)$.*

Output traces are defined over discrete-time and contain a finite number of samples T. We use the standard notation $\mathcal{T}^Y$ to denote the set of all output traces. In the rest of the paper, if it is clear from the context we drop the parameter p while referring to an output trace, and explicitly state it whenever required. When $\mathcal{M}$ is a computer model of a dynamical system, we typically assume that we can obtain output traces via numerical integration of the underlying ODEs, but the traces could also be obtained from test data from a physical system.

We use ψ to denote any property that defines correct system behavior. Assume that $\mathcal{M}$ can be evaluated to determine whether ψ holds for an output trace corresponding to a particular p. We use $y \models \psi$ to denote that y satisfies ψ, and $y \not\models \psi$ to denote that y does not satisfy ψ.

Broadly speaking, the goal of testing procedures is to identify a $p \in P$ such that $y \not\models \psi$, where y is an output trace of $\mathcal{M}$ under p.

Definition 2 (Testing). *The testing task is to* determine *if $\hat{y}_{\hat{p}} \models \psi$ is true for all $\hat{p} \in \hat{P}$, where $\hat{P}$ is a given finite subset of P.*

In general, a *test generation* procedure attempts to *generate* a $\hat{P}$ for the purpose of testing. A class of test generation procedures known as *falsification* techniques assume that there exist some output traces y_p, obtained from the model under parameter p such that $y_p \not\models \psi$, and then seeks to find p.

DEFINITION 3 (FALSIFICATION). *The falsification problem is to find a $p \in P$ such that $y_p \not\models \psi$, where y_p is an output trace of $\mathcal{M}$ under p.*

The difference between testing and falsification is subtle. Testing determines whether a property holds for a given (finite) set of parameters, whereas falsification actively searches for parameters from a (possibly infinite) set to demonstrate that the property does not hold. In this paper, by a test generation (TG) procedure, we mean a falsification-like procedure that attempts to find a finite set $\hat{P}$ such that some parameters in $\hat{P}$ falsify the system. In this work, we are also interested in parameters p for which $y_p \models \psi$ but there exists some y' close to y_p such that $y' \not\models \psi$. In that sense, we do not require $\hat{P}$ to contain only falsifying parameters, but can also permit parameters that are "close" to falsifying the system.

Our TG framework assumes that the property ψ relates to the convergence of the system. The convergence property can take any of a variety of forms, so long as there exists some efficient means of quantifying the degree of satisfaction of the property. For example, two types of properties that our framework supports are properties based on Lyapunov-like functions and signal temporal logic formulae.

DEFINITION 4 (LYAPUNOV-LIKE FUNCTION). *Given a finite set $\{y_1, \ldots, y_N\}$ of output traces of $\mathcal{M}$, a function $V : \mathbb{R}^n \to \mathbb{R}_{\geq 0}$ is called a Lyapunov-like Function (LLF) for system $\mathcal{M}$ if, for each y_i and each $j \in [1, T-1]$, where y_i is defined over T time steps:*

$$V(y_i(t_j)) > V(y_i(t_{j+1})). \quad (1)$$

In control theory, Lyapunov functions are the standard tool to assess the stability of dynamical systems. Lyapunov functions quantify the energy of the system at a state, are defined over the system state variables, and are required to decrease over any system trajectory. The decrease of this generalized energy function along the system trajectories implies the stability of the system. By contrast, Lyapunov-like Functions (LLF) are defined over system outputs, instead of system states and are only required to decrease over observed output traces. The intuition behind LLFs is that they demonstrate that the system output signals are, in some sense, converging. We define convergence properties based on LLFs in such a way that they provide a means to quantify the degree to which a given LLF decreases over a set of output traces.

A second class of convergence properties that we consider is based on signal temporal logic (STL) [19]. STL is used to measure the satisfaction value of logical formulae over real-valued signals. Syntactically, an STL formula is defined recursively; the basic unit is an atomic formula expressing constraints on signals, and larger formulas are built recursively using negations, Boolean combinations (conjunctions, disjunctions), or temporal operators applied to subformulas. Atomic formulas, without loss of generality, can be reduced to a form $f(\mathbf{x}) \bowtie 0$, where $\mathbf{x}$ is the name of signal (a function from $\mathbb{R}_{\geq 0}$ to $\mathbb{R}^n$), $\bowtie \in \{<, \leq, >, \geq, =\}$, and f is an arbitrary function from $\mathbb{R}^n$ to $\mathbb{R}$. A temporal formula is formed using operators "always" (denoted $\Box$), "eventually" (denoted $\Diamond$) and "until" (denoted $\mathbf{U}$). Each temporal operator is indexed by an interval I over $\mathbb{R}_{\geq 0} \cup \{\infty\}$. As an example, consider the following example of an STL formula: $\psi := \Box_{[2.0,5.0]} y_1(t) \leq 100.0$, which means that output signal y_1 should not exceed 100.0 between 2.0 and 5.0 seconds.

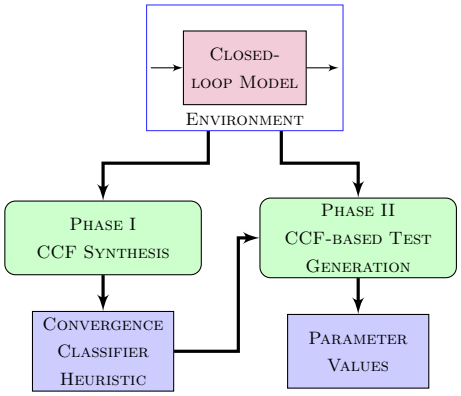

Figure 1: Underminer framework.

3. FRAMEWORK

In this section, we describe the Underminer framework, illustrated in Figure 1. The framework automatically produces test cases for a closed-loop model $\mathcal{M}$, based on a user-selected notion of convergence. The user selects the notion of convergence by choosing a class of *convergence classifier functions* (CCF), a function that measures the degree to which the system converges, and a test generation (TG) scheme that uses the chosen CCF.

The framework consists of two main phases. In Phase I, Underminer learns a CCF belonging to the class of CCFs specified by the user. The CCF class can be quite general; our current implementation supports LLF-based CCFs and STL-based CCFs. Any particular CCF $\mathcal{C} : Y^{\mathcal{T}} \to \mathbb{R}_{\geq 0}$ uses a specific notion of convergent behaviors and is able to distinguish convergent behaviors from non-convergent ones in the following way. Let $\psi_{\mathcal{C}}$ be the convergence property as defined by the CCF, and consider, y, the output trace of $\mathcal{M}$ under p. If the CCF evaluated on y (denoted $\mathcal{C}(y)$) is positive, then $y \models \psi_{\mathcal{C}}$, and if $\mathcal{C}(y) < 0$, then $y \not\models \psi_{\mathcal{C}}$. CCFs and corresponding learning heuristics are described in Sec. 4.

In Phase II, a TG procedure is used to seek output trajectories exhibiting non-convergent behaviors. The framework is flexible enough to allow different TG techniques to select the system parameters and inputs. Our current implementation supports stochastic global optimization based methods which use $\mathcal{C}(\cdot)$ as a cost function to find p such that $\mathcal{C}(y_p) < 0$. We also support a number of sampling-based TG procedures, including a method to pick points adaptively from a grid exploiting the sensitivity of $\mathcal{C}(y_p)$ within a particular region of P. We describe the TG techniques in detail in Sec. 5. The result of the framework shown in Fig. 1 is a collection of p_i values that correspond to output traces y_i, some of which may represent non-convergent behaviors as labeled by the chosen CCF.

4. CLASSIFIERS

Our framework requires a way to quantify the degree of acceptability of a given output trace. We call any function that serves this purpose a *classifier* function.

Definition 5 (Classifier). *A classifier $\mathcal{C}$ is any function from the space of finite-time signals over bounded domains to a finite or infinite interval over the real numbers that contains 0. A classifier is often associated with a property $\psi_{\mathcal{C}}$, and we say that an output trace y satisfies $\psi_{\mathcal{C}}$ (denoted $y \models \psi_{\mathcal{C}}$) if $\mathcal{C}(y) \geq 0$; otherwise we say that y does not satisfy $\psi_{\mathcal{C}}$ (denoted $y \not\models \psi_{\mathcal{C}}$).*

Classifiers can be used to define a wide variety of system behaviors, including simple properties such as bounds on overshoot values or settling times, but also complex properties such as any of those expressible in Signal Temporal Logic (STL). Our framework can be easily extended to support any of those properties, but our focus in this work is on classifiers to capture *convergent* behavior. Formalizing a canonical notion of convergence that fits every testing scenario is challenging, as selecting an appropriate definition can depend on the particular system domain and the testing context, and convergence in an engineering setting can be qualitative in nature. Control designers may deem a system output trace as well-behaved if the signal has some desirable shape, which is known intuitively to the designer (based on past experience) but is difficult to capture precisely in a mathematical sense.

Consider the examples illustrated in Fig. 2, to demonstrate this point. Fig. 2-(a) illustrates three output traces that all exhibit convergent behavior, in the sense that their respective envelopes approach the reference value, $y(t) = 0$. These are typical behaviors a designer expects to see for many types of feedback control systems. A naïve approach to capture this behavior with, for example, an STL formula might be to define a time (e.g., τ_s) in which the output signal must remain within a given boundary around the reference value (e.g., $|y_i(t)| \leq r_s$). But from the perspective of a control designer, the significant quality that the example output traces all possess is that they continue to approach the reference value, not that they remain within a fixed bound near the reference value within some specific amount of time. Thus, this naïve approach would not be sufficient to capture the intended behavior. One could argue that a more elaborate STL formula could be constructed to capture behaviors similar to those expected by the control designer, but the formulations would require significant user insight (for example to define appropriate problem-specific envelope profiles). By contrast, such behaviors can be naturally captured using a Lyapunov-like function (LLF), and using an LLF would require little user intervention, as we provide a means to learn LLFs based on example output traces.

Next, consider the signal shown in Fig. 2-(b). This is the type of behavior an engineer expects to see for a system that is implementing sampled-data sliding mode control [15]. This signal satisfies a convergence requirement that can be naturally captured in STL by selecting a settling time τ_s and a settling radius r_s. Note, however, that the signal would not satisfy any standard[1] Lyapunov conditions, as the signal does not continually converge to the $y(t) = 0$ value.

Thus, the architecture of our framework allows classifiers built on disparate notions of convergence. Intuitively, our classifiers can specify the following general qualities of the system behaviors:

– There is a certain set-point for the controlled signal, and as time progresses the signal value approaches this set-point as in Fig. 2-(a);

[1]One can define an alternative classifier to capture this notion of convergence based on Lyapunov characterization of practical stability.

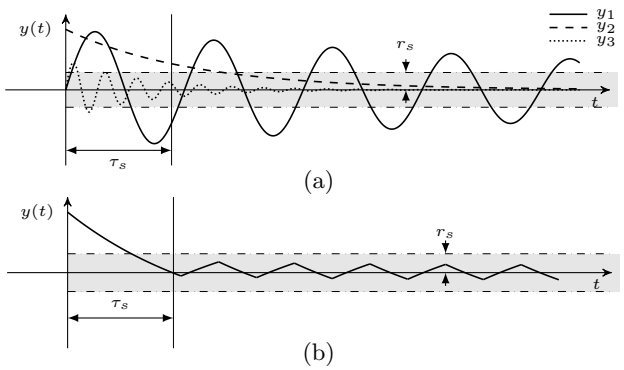

Figure 2: (a) Illustration of a class of desirable output traces that can be captured naturally with an LLF but are difficult to capture with temporal logic; (b) Illustration of desirable output traces that can be captured naturally with temporal logic but are difficult to capture with an LLF.

– It is desirable for the signal to approach a specified settling region quickly enough (in comparison to the time-scale at which the underlying system operates) as in Fig. 2-(b).

In what follows, we propose heuristic functions designed to capture these informal notions of convergence in a formal, machine-checkable fashion. Each such function is called a *convergence classifier function* (CCF).

STL CCF. We start with the simplest notion of convergence: after a certain time known as the *settling time* τ_s, the output trace should enter a prescribed *settling region*, $\{y : \|y - y_{\text{ref}}\| \leq r_s\}$, and stay there indefinitely. The settling region defines a neighborhood around the reference value y_{ref} to which the system should converge. Note that for the scenario given in Fig. (2), $y_{\text{ref}} = 0$. The following function corresponds to this notion of convergence[2]:

$$\mathcal{C}(\hat{y}(t)) = \inf_{t \in [\tau_s, t_T]} (r_s - |\hat{y}(t) - y_{\text{ref}}|), \tag{2}$$

where $\hat{y}(t)$ is the continuous time signal generated by piecewise linear interpolation of $y(t)$. The above function is negative for those output signals that either never enter the settling region centered at the reference value, or enter the region and then subsequently leave the region. In this way, it can capture some classes of non-convergent traces.

SoS LLF. LLFs provide a means to construct classifiers based on traditional notions of stability. An LLF gives an indication as to whether the energy in an output signal is decreasing over runs of the system. One way to construct a classifier based on an LLF is to consider the value of

$$\min_{i \in (1,\ldots,T-1)} V(y(t_i)) - V(y(t_{i+1})), \tag{3}$$

which provides the minimum decrease of the LLF over the trace y.

One challenge in applying the above classifier is to learn an appropriate LLF. We automate the process of learning an LLF using a technique reported in [25] and later elaborated on in [14]. The key step in this technique is to restrict the class of LLFs to a fixed template form corresponding to a sum-of-squares (SoS) polynomial function, as

[2]Note that this function closely corresponds to the robust satisfaction value of the STL formula $\Box_{[\tau_s, t_T]} (|\hat{y} - y_{\text{ref}}| < r_s)$ under the quantitative semantics as defined in [7].

in $V(y) = z^\top Pz$, where z is a $j \times 1$ vector of monomials in y, $\top$ is the transpose operator, and $P \in \mathbb{S}^{j \times j}$, where $\mathbb{S}$ denotes the set of positive $j \times j$ semidefinite matrices. The learning technique computes a P matrix from output traces as follows: for each output trace y and each $i \in (1, \ldots, T-1)$, the following two constraints are added to a collection of constraints:

$$V(y(t_i)) > 0, \quad (4)$$
$$V(y(t_i)) > V(y(t_{i+1})). \quad (5)$$

Both of the above constraints are linear in the decision variable P, and so the set of constraints defines a linear programming (LP) problem, for which robust and efficient numerical solvers are available [18]. Applying an LP solver to the set of linear constraints results in a P matrix defining an LLF, based on the set of output traces used to construct the constraints. An alternative approach is to replace the constraint (5) with a constraint requiring the P matrix to be a positive semidefinite matrix. This approach can provide stronger results by learning a $V(y)$ function that is positive everywhere, but requires the use of a semidefinite programming (SDP) solver such as `SDPT3` [24]. These solvers are less efficient than LP solvers, and can require greater conditioning of the constraints to obtain numerically robust results.

M-step LLF. We also consider LLFs as defined in the work of Geiselhart et.al. on Lyapunov functions for discrete-time dynamical systems [10]. In [10], Geiselhart et.al. introduced a template for Lyapunov functions such that for a large class of discrete-time dynamical systems whose energy is decreasing over time, a Lyapunov function in this template is guaranteed to exist.

For any output trace y of $\mathcal{M}$ defined over T time steps and any $M \leq T$, we define the M step output of $\mathcal{M}$ as

$$z(t_i) = \begin{bmatrix} y(t_i)^\top & y(t_{i+1})^\top & \ldots & y(t_{i+M-1})^\top \end{bmatrix}^\top, \quad (6)$$

where $1 \leq i \leq T - M + 1$. In the definition of $z(t_i)$, $y(t_i)$ is an $n \times 1$ vector representing the n outputs of $\mathcal{M}$ at time t_i. We consider LLFs of the form:

$$V(z) = z^\top Pz, \quad (7)$$

where z is the $nM \times 1$, and $P \in \mathbb{S}^{nM \times nM}$.

Note that contrary to an SoS LLF, such LLFs use not only the value of an output trace at time t_i but also its values for $t_{i+1}, \ldots, t_{i+M-1}$, hence the name M-step. We compute the matrix P using the same technique used for learning SoS LLF-type classifiers. Specifically, we solve an LP program satisfying the following constraints for each output trace y and each $i \in \{1, \ldots, T - M + 1\}$:

$$V(z(t_i)) > 0,\ V(z(t_i)) > V(z(t_{i+1})). \quad (8)$$

The strength of M-step LLFs compared to SoS LLFs is that M-step LLFs can be much easier to learn for certain systems. For example, consider a single dimensional output for a simple two dimensional linear system. A characteristic output of some linear systems is an exponentially decaying sinusoid that converges to zero. An SoS LLF over such an output signal requires an arbitrarily high degree to certify convergence. Furthermore, even for systems with multidimensional stable outputs, we empirically observe that M-step LLFs are easier to find than SoS LLFs (discussed in the experimental evaluation presented in Sections 6 and 7).

We can show that for a large class of discrete-time dynamical systems an LLF of the form (7), which satisfies (8), is guaranteed to exist. The following remarks formalize this discussion.

REMARK 1. *Let $\mathcal{M}$ be a discrete-time dynamical system*

$$x^+ = f(x) \qquad x \in X \subset \mathbb{R}^d, y \in Y \subset \mathbb{R}^n, \quad (9)$$

where X and Y are compact sets. Then if (9) is exponentially stable, i.e., if the state trajectories of (9) satisfy:

$$\|f^k(x)\| \leq c\mu^k \|x\|,$$

for some $c \in \mathbb{R}$ and $\mu \in [0, 1)$, and if for some $\lambda > 0$ and $\alpha_1 \leq \alpha_2$ the output map satisfies: $\alpha_1 \|x\|^\lambda \leq \|g(x)\| \leq \alpha_2 \|x\|^\lambda$, then an LLF $V(z)$ of the form (7) satisfying (8) is guaranteed to exist for some $M \in \mathbb{N}^+$.

REMARK 2. *If the $\mathcal{M}$ is a discrete-time dynamical system whose output trajectories satisfy: $\|y(t_k)\| \leq c\mu^k \|y(t_0)\|$, for all $k \in \mathbb{N}$, for some $c \in \mathbb{R}$, and $\mu \in [0, 1)$, then an LLF $V(z)$ of the form (7) satisfying (8) is guaranteed to exist for some $M \in \mathbb{N}^+$.*

The proofs for Remark 1 and Remark 2 follow from the results in [10], and are omitted here due to lack of space.

Finally, we note that learning the CCFs occurs in Phase I of the Underminer framework. The quality of learned CCFs usually depends on the number of randomly sampled parameter values to generate traces to learn the CCFs, as well as the options provided to the LP or SDP solvers. The various options used for learning CCFs (including the template polynomials used in SoS LLFs) constitute the *inductive bias* of our learning procedure.

5. TEST GENERATORS

We now explain how CCFs constructed in Phase I are used in Phase II to generate tests. A key thesis in this paper is that using a CCF $\mathcal{C}$ to guide the TG procedure can generate qualitatively better test results. A good TG procedure attempts to find parameter values p such $\mathcal{C}(y_p)$ is minimal; however, as $\mathcal{M}$ is a black box system, we do not have access to a symbolic representation of transition function $\Phi_\mathcal{M}$ and thus cannot apply white-box optimization approaches such as gradient-descent. Thus, we focus on black box optimization techniques. We begin by describing simple TG schemes inspired by prevalent engineering practices.

Random Sampling-based TG. The simplest TG scheme (denoted `URand` TG) has the following steps: (1) use a uniformly random sampling method to obtain a set $\hat{P}$, such that $|\hat{P}| =$ `NumSimulations`, (2) for each $p \in \hat{P}$, obtain y_p, (3) sort the set $\{\mathcal{C}(y_p) \mid p \in \hat{P}\}$ in ascending order, and (4) report the top `NumTests` values back to the user (*i.e.*, corresponding to the worst `NumTests` values of the CCF).

Grid-based TG. The second TG scheme (denoted `Grid` TG) constructs an equally spaced grid over the parameter space P such that the grid contains at least `NumSimulations` number of grid points $\hat{P}$. Similar to `URand` TG, it obtains the set $\{\mathcal{C}(y_p) \mid p \in \hat{P}\}$, sorts it, and returns the top `NumTests` parameters in $\hat{P}$ to the user.

Adaptive Grid-based Sampling. A disadvantage of the simple schemes outlined above is that they are oblivious to the underlying dynamics of the black box system; however, for any CCF $\mathcal{C}$, and for most models $\mathcal{M}$, there are parameter values p where the value of $\mathcal{C}(y_p)$ is low (possibly negative or even a local minimum), and there also are regions where $\frac{\partial \mathcal{C}(y_p)}{\partial p}$ is high. In other words, either the CCF achieves a value of interest, or the region shows rapid change in the value of the CCF. The `AGrid` TG method aims to intelligently explore an implicit grid on the parameter space P by

Algorithm 1: Adaptive Sampling Algorithm.

Input: $\{\delta_0, P, \texttt{NumSimulations}, \texttt{NumTests}, \text{CCF } \mathcal{C}, \mathcal{M}\}$
Output: TESTS

1 $\delta \leftarrow \delta_0$, parameters $\leftarrow$ GRID(δ, P)
2 TESTS $\leftarrow$ SORT(parameters, `NumTests`)
3 num_p $\leftarrow 0$
4 **while** (num_p $<$ `NumSimulations`) **do**
5 **foreach** $q \in$ parameters **do**
6 **foreach** $p \in$ GRID$(\delta/2, \mathbb{B}^{\inf}_{\delta}(q))$ **do**
7 num_p $\leftarrow$ num_p $+ 1$
8 $L \leftarrow$ EstimateSensitivity$(\mathbb{B}^{\inf}_{\delta}(p), \mathcal{C}, \mathcal{M})$
9 **if** $\left(\frac{\delta}{2} > \frac{\mathcal{C}(y_p)}{L}\right)$ **then**
10 TESTS $\leftarrow$ SORT(TESTS $\cup \{p\}$, `NumTests`)
11 parameters $\leftarrow$ TESTS
12 $\delta \leftarrow \frac{\delta}{2}$

avoiding regions with high values of $\mathcal{C}(y_p)$, or regions where $\mathcal{C}(y_p)$ is positive and changing slowly. This has the effect of prioritizing search in the interesting regions, by avoiding uninteresting regions.

Algorithm 1 implements the `AGrid` TG scheme used in Underminer. It takes as input the CCF $\mathcal{C}$, an initial resolution for discretization δ_0, the parameter space P, a budget for the maximum number of simulations (`NumSimulations`), and the number of tests to generate (`NumTests`). The algorithm is a breadth-first procedure; in each iteration, it maintains a list of the most promising parameter values in a sorted list TESTS, and iterates over each value in the list, discarding the value if it is discovered to lie in an uninteresting region, and keeping it otherwise. The algorithm begins by creating a coarse grid over the parameter space with the subroutine GRID(δ_0, P). This subroutine outputs a finite gridding of P with discrete steps of magnitude δ (Line 1). The subroutine SORT(parameters, `NumTests`) in Line 2 computes $\mathcal{C}(y_p)$ for each $p \in$ parameters, sorts the resulting values in ascending order, and picks the first min(|parameters|, `NumTests`) values. These values are our initial set of test cases. Up to this point, Algorithm 1 has the same effect as using `Grid` TG with a grid size δ_0.

We then start refining the grid (Line 6). We use $\mathbb{B}^{\inf}_{\delta}(q)$ to denote $\{q' \mid \|q - q'\|_\infty < \delta\}$. The idea here is that regions in the parameter space are only refined in regions where we expect that the CCF value $\mathcal{C}(y_p)$ can become negative. To this end, given a parameter value p, the function EstimateSensitivity estimates $\frac{\partial \mathcal{C}(y_p)}{\partial p}$ by sampling S number of points $\Delta(p) = \{p_1, \ldots, p_S\}$ in the neighborhood of p and computing: $\max_{p_i, p_j \in \Delta(p)} \frac{\|\mathcal{C}(y_{p_i}) - \mathcal{C}(y_{p_j})\|}{\|p_i - p_j\|}$.

Using this estimate, in Line 9, we check whether we need to refine the region containing p. If yes, we add p to the list TESTS if $\mathcal{C}(y_p)$ is worse than the values of $\mathcal{C}(y_{p'})$ for all p' in TESTS in Line 10. We then repeat this process by refining the grid around the first `NumTests` number of parameter values. The algorithm terminates when the number of explored parameter values exceeds the user defined budget. `NumSimulations`.

Optimization-guided TG. Underminer supports TG schemes that utilize black box optimization techniques for test generation. The application of these techniques is common in falsification methods such as those used in tools like S-TaLiRo [2] and Breach [6]. Using input parameters as decision variables, these tools use a global optimizer to find the minimum (or at least low) values of a cost function that encodes property satisfaction. Any valuation of the cost function that is negative corresponds to system behaviors violating the given logical property. The CCFs that we learn share similar characteristics; negative values indicate non-convergent behaviors. We use an extension of the Nelder-Mead nonlinear optimization algorithm [22]. The key property of the Nelder-Mead algorithm is that it is a derivative-free heuristic search method that can be used in a black box setting where there is no explicit description of the objective function. Note that, the Nelder-Mead algorithm can converge to a suboptimal solution depending on the initial point we start our search from. To avoid local refinements in a suboptimal region, we introduce a stochastic extension of this algorithm, where we start the heuristic search from a number of randomly chosen initial points in the search space. We denote this TG scheme as `S-NM` TG.

6. EXPERIMENTAL EVALUATION

In this section, we benchmark Underminer on textbook examples of dynamical systems. For each of the examples to follow, we have an explicit representation of the system dynamics; however, we do not make use of this knowledge during the experiments, and effectively treat each system as a black box. We consider two sets of examples. The systems in the first set are known to be **not** globally asymptotically stable. For these systems, we wish to use Underminer to identify parameter values for which the output traces do not converge to a given equilibrium point. In the second set, we consider dynamical systems whose output traces seem to have convergent behavior, yet could have undesirable transient behavior such as high overshoots or slow settling time. Here, we wish to demonstrate the utility of having a classifier function to guide the search for undesirable behaviors. Our thesis is that as long as Underminer learns a good CCF in Phase I, the traces that it labels as severe (as measured by the CCF) also fare poorly in terms of traditional metrics such as settling time or overshoot.

In all experiments below, if necessary, we first perform a change of coordinates so that the desired reference value of the system is at the origin, i.e., $y_{\text{ref}} = 0$. For each example, we benchmark all three types of CCFs: SoS LLF, M-step LLF, and STL CCF, and consider four different test generation (TG) methods. Phase I and Phase II of Underminer require the user to select various options for the classifier learning and test generation algorithms. When learning the SoS LLF and M-step LLF in Phase I, we allow the user to specify the number of uniformly randomly or quasi-randomly sampled parameter values (denoted `NumCCFSeeds`) to generate traces to learn the CCFs. In Phase II, we assume that the user specifies: (1) a budget for the maximum number of simulations to be run (denoted `NumSimulations`), and (2) the number of tests that they would like to obtain using Underminer (denoted `NumTests`, such that `NumTests` $\leq$ `NumSimulations`).

Finally, for test cases generated by Underminer we consider two performance metrics: the ratio of non-convergent test cases to the number of total generated test cases, which we denote by r_{nc}, and the average distance between the parameter values output by each TG method, denoted by d_{avg}. The quantity d_{avg} is an estimate of the dispersion of the points in the parameter space.

Time-Reversed Van der Pol Example. The first ex-

	CCF	TG	r_{nc}	T_{TG} (s)	d_{avg}
Van der Pol osc.	SOS LLF	URand	0.44	21.73	0.096
		S-NM	1	16.22	0.004
		Grid	0.68	23.56	0.113
		AGrid	0.72	25.06	0.076
	M-Step LLF	URand	0.44	2	0.094
		S-NM	1	2.33	0.008
		Grid	0.68	2.12	0.1
		AGrid	0.88	3.9	0.065
	STL CCF	URand	0.44	41.79	0.067
		S-NM	1	7.37	0.015
		Grid	0.68	8.57	0.1
		AGrid	1	115.9	0.059
Lorenz attr.	SOS LLF	URand	0.85	27.33	0.077
		S-NM	1	29.56	0.008
		Grid	0.95	27.16	0.021
		AGrid	0.9	38.91	0.022
	M-Step LLF	URand	0.775	3.34	0.07
		S-NM	1	2.84	0.009
		Grid	0.8	6.97	0.021
		AGrid	1	3.35	0.023
	STL CCF	URand	0.9	100.56	0.076
		S-NM	1	7.19	0.01
		Grid	0.9	90.8	0.021
		AGrid	1	437	0.029

Table 2: Comparison of different CCF and TG methods on dynamical systems known to be not globally stable. T_{TG} denotes the computation time for the corresponding TG in seconds.

ample we consider is the time-reversed version of the van der Pol oscillator. The dynamics can be described by the following set of ODEs:

$$\dot{x}_1 = -x_2,\ \dot{x}_2 = -(1 - x_1^2)x_2 + x_1,\ y = x.$$

The unique equilibrium of the system is at $x = 0$. The origin is a locally asymptotically stable equilibrium point but is not globally asymptotically stable. Thus, some traces asymptotically converge to the equilibrium, while others diverge. In this example, we use Underminer to detect the initial conditions that lead to these non-convergent traces. To this end, we choose $P = [-2, 2]^2$, i.e., the parameter space is a subset of the state space. For the STL CCF, the parameters in (2) are chosen as $r_s = 0.1$ and $\tau_s = 5$. For both the SoS LLF and M-step LLF we use `NumCCFSeeds` $= 75$, `NumSimulations` $= 105$, and `NumTests` $= 25$. Table 2 summarizes our results.

All TG methods for each CCF are able to identify non-convergent output traces; this is expected as the output traces starting outside the region of attraction are inherently divergent. For this example, `S-NM` consistently outputs solely non-converging traces ($r_{\text{nc}} = 1$); however, based on the low value of d_{avg}, we conclude that the parameter values that correspond to the generated test cases are concentrated in a small region, *i.e.*, not dispersed. Since the initial conditions that lead to non-converging traces are confined in a small region of the parameter space in this example, `URand`, which uses a uniform distribution to randomly sample the parameter space P, produces the smallest ratio of non-converging tests ($r_{\text{nc}} = 0.44$).

Lorenz System. The next example we consider is the Lorenz system, whose dynamics are given as:

$$\dot{x}_1 = \sigma(x_2 - x_1),\ \dot{x}_2 = x_1(\rho - x_3) - x_2,\ \dot{x}_3 = x_1x_2 - \beta x_3,$$

and $y = x$, where $\sigma = 10$, $\rho = 14$, $\beta = \frac{8}{3}$. For small values of ρ, the Lorenz system has two locally stable equilibria. In this example, we choose a small neighborhood ($r_s = 0.1$) around one of the equilibria as the desired settling region and use Underminer to identify the initial conditions leading to output traces that do not converge to the region $|x| < 0.1$ in $\tau_s = 10$ seconds. We assume that $P = [-12, 12]^2 \times [-1, 1]$, `NumSimulations` $= 105$, `NumCCFSeeds` $= 80$, and `NumTests` $= 40$. We summarize our findings in Table 2. The performance of STL CCF is the best among CCFs since all the TG methods output at least 90% non-converging traces using STL CCF. Table 2 also illustrates that in this example the `S-NM` and `AGrid` TG methods consistently lead to good performance for all the CCFs ($r_{\text{nc}} \geq 0.9$).

Not Stable System. The next example we consider is from [23] with dynamics given as:

$$\dot{x}_1 = x_1^2 - x_2^2,\ \dot{x}_2 = 2x_1x_2,\ y = x. \tag{10}$$

The traces of this system are peculiar because all traces converge to the origin except the trace that follows the positive x_1 axis and goes to $+\infty$. Moreover, the closer the initial condition is to the positive x_1 axis, the larger the overshoot and the higher the settling time. Therefore, in this example, we expect a good TG method to provide several test cases close to the positive x_1 axis. We demonstrate in Fig. 3 that this is indeed the case for all TG methods. We use $P = [-1, 1]^2$, `NumCCFSeeds` $= 35$, `NumSimulations` $= 280$, `NumTests` $= 80$, and $r_s = 0.1$, $\tau_s = 4$ for this example. In Table 3, as a performance measure, we provide the average overshoot and average settling time of the traces generated using different TG methods and CCFs. Higher values of average overshoot and settling time indicate that the corresponding TG method more successfully focuses on the non-converging regions of the parameter space. As can be seen, the `Grid` and the `S-NM` have the best performance among the TG methods. While `S-NM` TG is able to identify traces that tend to $+\infty$ using only the STL CCF, the `Grid` TG consistently outputs these test cases for all CCFs. This is expected since `Grid` samples parameter values exactly on the x_1 axis independently of the chosen CCF. Fig. 3 demonstrates that `AGrid` does explore the regions in parameter space where the output traces exhibit high overshoot, however still `S-NM` has better performance in terms of identifying traces with higher average settling times and average overshoots. This is because the `S-NM` TG method discovers parameter values close to each other, as opposed to `AGrid` TG, whose outputs lie on an implicit grid. This can also be seen in Table 3, where d_{avg} values for parameters discovered by the `S-NM` TG is much smaller ($d_{\text{avg}} \approx 0.1$) than the other TG methods.

Rössler Attractor. The Rössler attractor is a well known example of a chaotic continuous time dynamical system. It first arose from modeling oscillations in chemical reactions but is now used as a benchmark chaotic system. The dynamics of the Rössler attractor are given by the following ODEs: $\dot{x}_1 = -x_2 - x_3$, $\dot{x}_2 = x_1 + ax_2$, $\dot{x}_3 = b + x_3(x_1 - c)$, and $y = x$, where $a, b, c \in \mathbb{R}$. For this example we fix $b = 0$ and $c = 5.7$. For these values of b and c, it is known that the attractor converges to the equilibrium at the origin whenever $a \leq 0$; however, the convergence properties depend on the value of a, as well as the initial condition of the system and this dependence is highly nonlinear. For this example, we seek to find values of a and initial conditions for the states x_1, x_2, and x_3 that correspond to traces that do not converge to the equilibrium (i.e., $y_{\text{ref}} = 0$) at all or for which

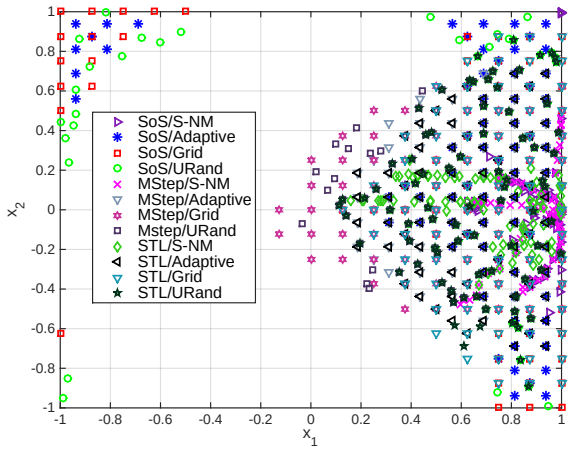

Figure 3: Tests generated using different CCFs for the "not stable" dynamical system. Each test case is an initial condition for the system (10)**.**

the convergence is slow. Each test case (x_1, x_2, x_3, a) is generated from the parameter set $[-2, 2]^3 \times [-1, -0.1]$. Table 3 summarizes our results. Underminer did not find any behaviors that do not settle around the equilibrium (which is expected for $a \leq 0$), but it did successfully discover traces with high settling time. As can be seen, the average settling time of the traces computed by the `S-NM` TG using STL CCF is approximately 9× that of the one obtained using `Grid` TG.

The experiments suggest that, in general, the `S-NM` TG method outputs the most interesting traces using a moderate computation budget. The generated test cases, however, are concentrated in a small region of the parameter space. Hence, `S-NM` TG does not give a good idea of the distribution of the interesting behaviors, which might prevent the user from gaining a better understanding of the source of the non-convergent behaviors. Gridding based methods, on the other hand, provide more coverage of the parameter space and therefore lead to more distinct test cases; however, this coverage comes at a cost. In `Grid` TG, this cost manifests in the performance of the test generation algorithm; the convergence properties of the output tests are not as interesting as the ones of `S-NM` TG within the given simulation budget. In `AGrid` TG, even though there is less performance degradation, the increase in the computation time is significant due to the increased number of simulations arising from the procedure to estimate the local sensitivity of the CCF to the parameter values.

7. CASE STUDIES

Suspension Control in a Quarter Car Model. The quarter car model is used to analyze automotive suspension systems. It considers only one of the wheels, and simplifies the system dynamics to a 1D model containing multiple spring-damper systems. A properly designed suspension system has good road holding ability, while providing the driver comfort when riding over bumps and holes in the road. A key performance indicator is the transient response of the suspension system; it is important that system not exhibit large overshoots or sustained oscillations. We use a Simulink® model of the quarter car system and the controller designed in [21]. The controlled quantity is the distance between the suspension and the body (denoted y), and the authors provided an informal specification for the controller in terms of a maximum permitted overshoot (5%) and a desired settling time of approximately 2 seconds. In

	CCF	TG	avg. settling time (s)	avg. overshoot	T_{TG} (s)	d_{avg}
Not Stable Sys.	SoS LLF	`URand`	11.07	6.69	102.58	0.04
		`S-NM`	11.49	31.44	105.66	0.016
		`Grid`	∞	∞	94.88	0.07
		`AGrid`	11.26	2.85	131.35	0.049
	M-Step LLF	`URand`	11.72	6.41	6.66	0.045
		`S-NM`	11.44	25.6	2.72	0.013
		`Grid`	∞	∞	5.66	0.083
		`AGrid`	11.57	2.61	11.33	0.06
	STL CCF	`URand`	11.84	6.61	348	0.038
		`S-NM`	∞	∞	11.45	0.02
		`Grid`	∞	∞	269	0.067
		`AGrid`	11.66	2.72	501	0.055
Rössler Attractor	SoS LLF	`URand`	14.51	2.51	98	0.165
		`S-NM`	30.09	2.76	86	0.015
		`Grid`	9.06	3.07	97	0.095
		`AGrid`	12.77	2.55	141	0.069
	M-Step LLF	`URand`	17.15	1.14	8.41	0.146
		`S-NM`	41	1.36	5.03	0.028
		`Grid`	8.07	1.63	8.41	0.08
		`AGrid`	27.49	1.35	17.06	0.1
	STL CCF	`URand`	44.53	2.08	370	0.175
		`S-NM`	70.47	2.27	30	0.009
		`Grid`	13.72	2.64	83	0.25
		`AGrid`	27.24	1.94	405	0.237

Table 3: Comparison of different CCF and TG methods for systems with converging traces.

TG	Overshoot		Settling Time (s)		r_{nc}
	Avg.	Max.	Avg.	Max.	
`URand`	0.43	0.91	1.62	2.8	0.16
`Grid`	0.12	0.27	1.03	1.4	0
`S-NM`	0.11	0.26	1.03	1.3	0
`AGrid`	0.63	5.02	∞	∞	0.14

Table 4: Comparison of different test generation methods for the passive suspension control model.

our experimental setup, we assume that the settling region is defined as $|y| < 0.01$.

The authors provided a nominal PID controller, and comment that the PID gains are obtained through trial-and-error. It is well-known that varying the gains K_p, K_i and K_d for the proportional, integral and derivative parts of a controller can drive a system to instability. In this experiment, we wish to investigate how the closed-loop system behaves if we vary the gains of the nominal controller. We pick a sizable region that is expected to contain a variety of behaviors, including large overshoots, unstable behavior or behavior that does not settle within the specified settling time. We then pick the M-step LLF-based CCF as the metric to determine convergence of trajectories. In Phase I of Underminer, we learn a suitable M-step LLF CCF from 100 convergent trajectories (with $M = 10$). In Phase II, we use the CCF to benchmark the various test generation options. The table below summarizes the key findings.

Table 4 shows that `AGrid` TG is very effective in this case study to find both high overshoots as well as non-convergent behaviors. In the course of execution, `AGrid` TG is able to mark several points in the parameter space as not interesting (using the local sensitivity and the value of the CCF

at the point), and is thus able to focus on parameter values that lead to non-convergent behaviors. We remark that `AGrid` TG also finds behaviors that do not settle to the given region before the simulation time horizon is reached; we assign a settling time of infinity to such behaviors. While the `S-NM` method does not seem to find any non-convergent behaviors, visual inspection of the results indicates that the output behaviors are more "oscillatory" than the ones found by other methods; however, these do not appear in the results as the magnitude of the oscillations is well within the settling region. Finally, this case study also highlights the possibility of design-space exploration using a tool such as Underminer, as we are able to effectively identify regions in the K_d, K_p, K_i space for which the controller shows acceptable behavior.

Air-Fuel Ratio Control Benchmark. In this experiment, we demonstrate the flexibility of Underminer. In [12], the authors present different closed-loop models of the air-fuel ratio control problem in gasoline engines. In this problem, we are interested in regulating the ratio of air to fuel (denoted λ) in the mixture that undergoes combustion in the engine to a stoichiometric value of 14.7 (denoted λ_{ref}). As the peak efficiency of a catalytic converter to reduce noxious emissions in the exhaust is reached when the air-fuel ratio is 14.7, this is an important control problem.

Among the various models provided, we focus on the first two models, which are both hybrid (i.e., contain continuous-valued states and also controller modes). The first is a complex model with features such as transport delays, look-up tables, highly nonlinear dynamics, and a discrete-time controller. The second model is a simplified version of the first model, with features such as transport delays removed and look-up tables replaced with polynomial approximations. Numerical simulations are marginally slower to run on the first model. In our setup, we used the second (easier) model to learn an SoS LLF and an M-step LLF (with $M = 5$), and then used these CCFs to generate tests with the first model. Our intention here is to mimic the process of learning a CCF on a simpler model, and then use it for test generation on a more complex model, or on a physical system such as a hardware-in-the-loop-simulation (HILS) setup. The exogenous inputs to both models are the pedal angle for a throttle plate (i.e. driver input) and the engine speed. The outputs of the models are the normalized air-fuel ratio μ, ($\mu = \frac{\lambda - \lambda_{\text{ref}}}{\lambda_{\text{ref}}}$), and the state of the integrator in the discrete-time PI controller.

In Table 5 we present the results of using different test generation schemes on the first model using a CCF learned on the second model. All TG methods perform in an almost similar fashion with respect to the overshoot behavior with either CCF. One outlier is an output trace found by the `S-NM` TG method corresponding to an overshoot of more than 3%. We next consider the settling behavior of the traces. We define the settling region as $|\mu| < 2 \times 10^{-3}$, and define a settling time of $\tau_s = 2.5$ seconds. As indicated in [12], the model used for test generation generally exhibits good transient response, and as expected, our experiments were not able to find many non-convergent traces. Hence, we instead report the average and maximum settling times found by each method. From Table 5, we can observe that both M-step LLF CCF and STL CCF were able to provide better guidance to the `AGrid` TG method when looking for non-convergent behaviors. Using M-step LLF CCF, we were able to find 1 and 7 non-convergent trajectories (out of `NumTests` = 50) using the `S-NM` and `AGrid` TG respectively.

CCF	TG	Overshoot% Avg.	Overshoot% Max.	Settling Time (s) Avg.	Settling Time (s) Max.
SoS LLF	`URand`	1.027	1.043	2.17	2.2
	`Grid`	1.022	1.037	2.18	2.3
	`S-NM`	1.028	1.045	2.17	2.2
	`AGrid`	1.024	1.037	2.2	2.2
M-step LLF	`URand`	1.019	1.039	1.73	3.3
	`Grid`	1.022	1.036	1.69	1.7
	`S-NM`	1.017	1.050	∞	∞
	`AGrid`	1.018	1.037	∞	∞
STL CCF	`URand`	0.996	1.051	∞	∞
	`Grid`	0.972	1.049	2.22	2.32
	`S-NM`	1.025	3.316	2.23	2.25
	`AGrid`	0.988	1.047	∞	∞

Table 5: Results of using different TG schemes with CCFs learned from a simplified version of the model.

CCF	TG	Norm. Overshoot Avg.	Norm. Overshoot Max.	r_{nc}	T_{TG} (mins)
M-step LLF	`URand`	1	1	0.66	29.43
	`Grid`	0.51	-16.28	0.56	29.55
	`S-NM`	1.15	56.16	0.78	32.43
	`AGrid`	1.24	-38.73	0.30	45.91
STL CCF	`URand`	1	1	0.74	28.56
	`Grid`	0.52	-6.31	0.76	28.64
	`S-NM`	0.19	7.68	0.94	31.06
	`AGrid`	5.00	-35.56	0.06	45.21

Table 6: Experimental results for the diesel air path model. The `AGrid` TG method using M-step LLF CCF based was able to finish 43 simulations, while using the STL CCF was able to finish 32 simulations.

Using STL CCF, we were able to find 1 and 2 non-convergent traces using `AGrid` TG and `URand` TG respectively. Overall, the `AGrid` TG method is able to focus on regions of parameter space that correspond to non-convergent behaviors.

Diesel Engine Control. Next, we consider an industrial-scale closed-loop model of a prototype air path controller for a diesel engine implemented in Simulink®. The model contains a high fidelity plant model of the air path dynamics and a model predictive controller (MPC) [11] to regulate the intake manifold pressure and the exhaust gas recirculation (EGR) flow estimation. The model is hybrid and has more than 3,000 blocks and 20 multi-dimensional lookup tables. Due to high model complexity, simulation is computationally expensive, and hence the design process would be assisted by a tool that can efficiently explore the input space. We consider a scenario with two model inputs: (1) the fuel injection rate (excited by a step of magnitude in a given allowable range), and the engine speed (picked from a given range, but held constant during any single simulation). The output signal of interest is the difference between intake manifold pressure and a given reference value.

In this study, we attempt to learn an SoS LLF and an M-step LLF in Phase I of Underminer. We fail to identify an SoS LLF CCF because the selected parameterizations of the SoS LLF are inadequate. As the model has a single output, learning an SoS LLF that decreases over all output traces requires the SoS LLF parameterization to be of an arbitrarily high degree. On the other hand, using an M-step LLF brings more flexibility as it contains M-long output traces. For this model, we find an M-step LLF with $M = 3$.

We report the experimental results in Table 6. As the actual numbers for overshoot are proprietary, we normalize the overshoots that we compute with respect to the results obtained using `URand` TG. For example, if the average overshoot reported by `URand` TG is b, and the average overshoot reported by `S-NM` TG is c, we report the quantity $100\frac{b-c}{b}$. For both CCFs we use a time limit of 45 minutes to run the `AGrid` TG. As can be seen from Table 6, all TG methods create test suites with both convergent and non-convergent outputs. Using STL CCF leads to discovering more non-convergent tests than using M-step LLF CCF. Among the three TG methods, the `S-NM` TG method has the best results as it identifies outputs with 50% greater average overshoot when using the M-step LLF CCF. As the `AGrid` TG uses single-step simulations to estimate local sensitivity of the CCFs, it takes more time than the other TG methods. In spite of running short simulations, the initialization time for running simulations in Simulink® becomes a bottleneck, and hence, the `AGrid` TG is unable to generate the desired number of tests in the given time limit. Nevertheless, the `AGrid` TG successfully finds an overshoot behavior that is 5% worse than the one found with `URand` TG. This example demonstrates that while the `AGrid` TG is promising, time required in estimating sensitivity and the curse of dimensionality inherent in any gridding approach can together be an impediment in a test-setup with limited trace or time budgets.

8. CONCLUSIONS

We propose the "Underminer" framework for finding parameter values and inputs to black box system models that lead to undesirable behaviors. Underminer operates in two phases. In Phase I, the tool learns a convergence classifier function (CCF) to discriminate convergent behaviors from those that are not convergent in a timely fashion. In Phase II, it uses the CCF to guide the search for parameter values such that the corresponding outputs are deemed non-convergent by the CCF. Underminer is flexible in the choice of CCFs including those based on temporal logic and those based on Lyapunov functions. It can use an array of test generation (TG) techniques such as optimizer-driven testing and sampling-based techniques. We benchmark the various CCF and TG combinations on a number of academic examples and compare the results with established metrics for transient behavior such as settling time and overshoot. Further, we demonstrate Underminer on case studies from automotive control systems.

Acknowledgements. This research was partially supported by the NSF project ExCAPE: Expeditions in Computer Augmented Program Engineering (CCF-1138996).

9. REFERENCES

[1] *Using Simulink*. The MathWorks, 2007.

[2] Y. S. R. Annapureddy, C. Liu, G. E. Fainekos, and S. Sankaranarayanan. S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems. In *TACAS*, pages 254–257, 2011.

[3] A. Balkan, J. Deshmukh, J. Kapinski, and P. Tabuada. Simulation based contraction analaysis. In *Proc. of the 1st Indian Control Conference*, 2015.

[4] R. Bobiti and M. Lazar. A delta-sampling verification theorem for discrete-time, possibly discontinuous systems. In *HSCC*, pages 140–148, 2015.

[5] T. Dang and T. Nahhal. Coverage-guided test generation for continuous and hybrid systems. *Formal Methods in System Design*, 34(2):183–213, 2009.

[6] A. Donzé. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In *CAV*, pages 167–170, 2010.

[7] A. Donzé and O. Maler. Robust satisfaction of temporal logic over real-valued signals. In *FORMATS*, pages 92–106, 2010.

[8] T. Dreossi, T. Dang, A. Donzé, J. Kapinski, X. Jin, and J. V. Deshmukh. Efficient guiding strategies for testing of temporal properties of hybrid systems. In *NASA Formal Methods*, pages 127–142, 2015.

[9] G. E. Fainekos and G. J. Pappas. Robustness of Temporal Logic Specifications for Continuous-Time Signals. *Theor. Comp. Sci.*, 410(42):4262–4291, 2009.

[10] R. Geiselhart, R. H. Gielen, M. Lazar, and F. R. Wirth. An alternative converse lyapunov theorem for discrete-time systems. *Systems & Control Letters*, 70:49 – 59, 2014.

[11] M. Huang, K. Zaseck, K. Butts, and I. Kolmanovsky. Rate-based model predictive controller for diesel engine air path: Design and experimental evaluation. *IEEE Trans. on Control Systems Technology*, PP(99):1–14, 2016.

[12] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Powertrain control verification benchmark. In *HSCC*, pages 253–262, 2014.

[13] A. Jones, Z. Kong, and C. Belta. Anomaly detection in cyber-physical systems: A formal methods approach. In *CDC*, pages 848–853, 2014.

[14] J. Kapinski, J. V. Deshmukh, S. Sankaranarayanan, and N. Aréchiga. Simulation-guided lyapunov analysis for hybrid dynamical systems. In *HSCC*, 2014.

[15] H. Khalil. *Nonlinear Systems*. Prentice Hall, 2002.

[16] Z. Kong, A. Jones, A. Medina Ayala, E. Aydin Gol, and C. Belta. Temporal logic inference for classification and prediction from data. In *HSCC*, pages 273–282, 2014.

[17] A. Kozarev, J. Quindlen, J. How, and U. Topcu. Case studies in data-driven verification of dynamical systems. In *HSCC*, 2016.

[18] J. Löfberg. YALMIP : A toolbox for modeling and optimization in MATLAB. In *Proc. of the CACSD Conference*, 2004.

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

[20] R. Medhat, S. Ramesh, B. Bonakdarpour, and S. Fischmeister. A framework for mining hybrid automata from input/output traces. In *International Conference on Embedded Software (EMSOFT)*, 2015.

[21] B. Messner and D. Tilbury. Control tutorials for matlab and simulink.

[22] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[23] S. S. Sastry. *Nonlinear systems : analysis, stability, and control*. Springer, New York, 1999.

[24] K. C. Toh, M. Todd, and R. H. Tutuncu. SDPT3 - a MATLAB software package for semidefinite programming. *Optimization Methods and Software*, 11:545–581, 1998.

[25] U. Topcu, P. Seiler, and A. Packard. Local stability analysis using simulations and sum-of-squares programming. *Automatica*, 44:2669–2675, 2008.