Mining Requirements from Closed-Loop Control Models

Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V. Deshmukh, Sanjit A. Seshia

***Abstract*—Formal verification of a control system can be performed by checking if a model of its dynamical behavior conforms to temporal requirements. Unfortunately, adoption of formal verification in an industrial setting is a formidable challenge as design requirements are often vague, non-modular, evolving, or sometimes simply unknown. We propose a framework to mine requirements from a closed-loop model of an industrial-scale control system, such as one specified in Simulink. The input to our algorithm is a *requirement template* expressed in Parametric Signal Temporal Logic: a logical formula in which concrete signal or time values are replaced with parameters. Given a set of *simulation traces* of the model, our method infers values for the template parameters to obtain the *strongest candidate requirement* satisfied by the traces. It then tries to falsify the candidate requirement using a falsification tool. If a counterexample is found, it is added to the existing set of traces and these steps are repeated; otherwise, it terminates with the synthesized requirement. Requirement mining has several usage scenarios: mined requirements can be used to formally validate future modifications of the model, they can be used to gain better understanding of legacy models or code, and can also help enhancing the process of bug-finding through simulations. We demonstrate the scalability and utility of our technique on three complex case studies in the domain of automotive powertrain systems: a simple automatic transmission controller, an air-fuel controller with a mean-value model of the engine dynamics, and an industrial-size prototype airpath controller for a diesel engine. We include results on a bug found in the prototype controller by our method.**

***Index Terms*—Model-based design; Parametric Temporal Logics; Simulink; software engineering and verification**

I. Introduction

Industrial-scale controllers used in automobiles and avionics are now commonly developed using a model-based development (MBD) paradigm [?], [?]. The MBD process consists of a sequence of steps. In the first step, the designer captures the *plant model*, i.e., the dynamical characteristics of the physical parts of the system using differential, logic, and algebraic equations. Examples of plant models include the rotational dynamics model of the camshaft in an automobile engine, the thermodynamic model of an internal combustion engine, and atmospheric turbulence models. The next step is to design a *controller* that employs some specific control law to regulate the behavior of the physical system. The *closed-loop model* consists of the composition of the plant and the controller.

In the next step, the designer may perform extensive *simulations* of the closed-loop model. The objective is to analyze the controller design by observing the time-varying behavior of the signals of interest by exciting the exogenous, time-varying inputs of the closed-loop model. An important aspect of this step is *validation*, i.e. checking if the time-varying behavior of the closed-loop system matches a set of *requirements*. Unfortunately, in practice, these requirements are high-level and often vague. Examples of requirements the authors have encountered in the automotive industry include "better fuel-efficiency", "signal should eventually settle", and "resistance to turbulence". If the simulation behavior is deemed unsatisfactory, then the designer refines or tunes the controller design and repeats the validation step.

In the formal methods literature, a requirement (also called a *specification*) is a mathematical expression of the design goals or desirable design properties in a suitable logic. In an industrial setting, many companies have made a strenuous effort to document clear and concise requirements. However, for systems built on legacy models or legacy code, requirements are normally not available. Moreover, to date, formal validation tools have been unable to digest the format or scale of industrial-scale requirements and models. As a result, widespread adoption of formal tools has been restricted to testing syntactic coverage of the controller code, which is an open-loop system without the important behavior of the physical system, with the hope that higher coverage implies better chances of finding bugs.

In this paper, we propose a scalable technique to systematically mine requirements from the closed-loop model of an industrial-scale control system from observations of the system behavior. In addition to the closed-loop model, our technique takes as input a *template requirement*. The final output is a synthesized requirement matching the template. We assume that the model is specified in Simulink [?], an industry-wide standard that is able to: (1) express complex dynamics (differential and algebraic equations), (2) capture discrete state-machine behavior by allowing both Boolean and real-valued variables, (3) allow a layered design approach through modularity and hierarchical composition, and (4) perform high-fidelity simulations.

Formalisms such as Metric Temporal Logic (MTL) [?], [?], and later Parametric Signal Temporal Logic (PSTL) [?] have emerged as logics adept at capturing both the real-valued and time-varying behaviors of hybrid control systems. PSTL is particularly well-suited to expressing template requirements of a broad nature: It can be used both to express control-theoretic properties such as overshoot, undershoot, settling-

X. Jin and J. Deshmukh are with Toyota Technical Center e-mail: {xiaoqing.jin,jyotirmoy.deshmukh}@tema.toyota.com.

A. Donzé and S. A. Seshia are with the University of California, Berkeley e-mail: {donze,seshia}@eecs.berkeley.edu.

time, rise-time, RMS error, and dwell-time, as well as properties involving timing relations between events corresponding to discrete switching behavior. With the increasing acceptance of temporal logics in practical domains such as automotive systems [?], [?], it is reasonable to expect that libraries of commonly-used requirements will become available to control designers.

As a concrete example of an STL requirement, consider the following English specification: "eventually between time 0 and some unspecified time τ_1, the signal x is less than some value π_1, and from that point for some τ_2 seconds, it remains less than some value π_2". In PSTL the above property would be expressed as:

$$\Diamond_{[0,\tau_1]}(\mathtt{x} < \pi_1 \ \wedge \ \Box_{[0,\tau_2]}(\mathtt{x} < \pi_2)).$$

We refer to the unspecified values $\tau_1, \tau_2, \pi_1, \pi_2$ as *parameters*.

The proposed mining algorithm is an iterative procedure; in each iteration, it does the following steps:

1) In the first step, the algorithm synthesizes a *candidate requirement* from a given template requirement expressed in PSTL and a set of simulation traces of the model.
2) It then tries to falsify the candidate requirement using an optimization-based search algorithm.
3) If the falsification tool finds a counterexample, we add this trace to the existing set of simulation traces, and go to Step 1 of the next iteration. If no counterexample is found, the algorithm terminates.

For our implementation of the mining algorithm, we use the framework provided by the BREACH tool [?]. BREACH contains both key ingredients for the mining algorithm: a sophisticated parameter synthesizer [?] and an efficient STL falsifier [?]. At the heart of Step 1 is an efficient search over the space defined by the parameters in the PSTL property in order to generate a candidate requirement. A naïve way to use BREACH would (1) grid the parameter-space, (2) for each point in the grid, instantiate the PSTL property for each grid value (to get an STL property), and (3) pick the grid-point leading to an STL property with the *minimum* satisfaction value over all traces. (A lower satisfaction value[1] corresponds to a *stronger* STL property.) If the number of parameters is n, and the number of grid points for each parameter is m, then the number of times this naïve approach would invoke BREACH to compute the satisfaction values is $O(m^n)$, i.e., exponential in the number of parameters.

However, we observe that the satisfaction value of certain PSTL properties is monotonic in the parameter values. For example, for the property $\Diamond_{[0,\tau]}(x > \pi)$, the satisfaction value monotonically increases in the parameter τ and decreases in π. When monotonicity holds, we can get exponential savings when searching over the parameter-space by using methods like binary search. Though syntactic rules for *polarity* of a PSTL property identified in previous work [?] ensure satisfaction monotonicity, these rules are not complete. Hence, we provide a general way of reasoning about monotonicity of arbitrary PSTL properties using Satisfiability-Modulo-Theories (SMT) solving [?].

[1]We define satisfaction value of an STL property with respect to a given trace in Sec. IV-A

In this paper, we explore two applications for requirement mining. The first application is the obvious one: to generate requirements that serve as high-level specifications for the closed-loop model. The second application explores the use of mining as an enhanced bug-finding procedure.

In an industrial setting, formalized requirements that can be used for design validation are often unavailable. For example, consider the case of legacy controller code. Such code usually goes through several years of refinement, is developed in a non-formal setting, and is not very easy to understand for any engineers other than its original developers. In this context, mined requirements can enhance understanding of the code and help future code maintenance.

Consider another scenario. The model-based design of a controller usually involves different representations of the same controller at varying levels of abstraction. For example, a controller model could be in a visual, block-diagram-based language such as Simulink, or as just low-level code (e.g. in a language like C); it could be a research prototype, or the final mass-production-stage controller, and so on. A requirement mined for *one* model at any of these levels of abstraction could be used to validate the behavior of *all* other models, and thus ensure consistency across models.

To better explain the second application, we consider a motivating example. Suppose we wish to check if the model behavior ever has a signal that oscillates with an amplitude greater than a threshold. Considering the huge space of input signals, simply running tests on the closed-loop model requires executing a large number of simulations in order to detect such behavior. We instead attempt to mine the requirement, "the signal settles to a steady value π in time τ" (roughly corresponding to the negation of the original property). In each step, our algorithm pushes the trajectory-space exploration of the falsification tool in a region not already subsumed by existing traces. Hence, the search for a counterexample is guided by the intermediate candidate requirements. Note that state-of-the-art falsifiers such as S-TALIRO and BREACH would require a *concrete* STL property encoding the oscillation behavior, which would require tedious manual effort given many possible expressions of such behavior arising from unknowns such as the oscillation amplitude, frequency, and the time at which oscillations start.

To summarize, our contributions are as follows:

1) We propose a novel counterexample-guided iterative procedure for mining temporal requirements satisfied by signals of interest of an industrial-scale closed-loop control model (i.e., a highly nonlinear hybrid system of significant dynamical and mode complexity). Specifically, we target the mining of properties expressible in PSTL.
2) We extend BREACH to support Simulink models and the falsification of STL formulas. In addition we enhance the BREACH tool framework with efficient strategies for synthesizing parameters of monotonic PSTL properties. To extend the range of formula for which we can prove monotonicity, and hence apply these strategies, we formulate the query for monotonicity in a fragment of first order logic with quantifiers,

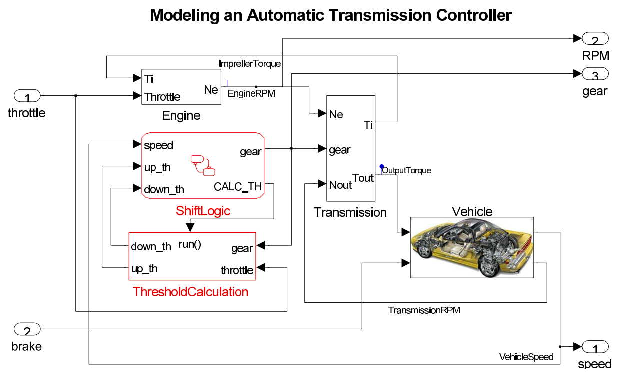


Fig. 1: The close-loop Simulink model of an automatic transmission controller. The input to the model is the throttle position and the brake torque.

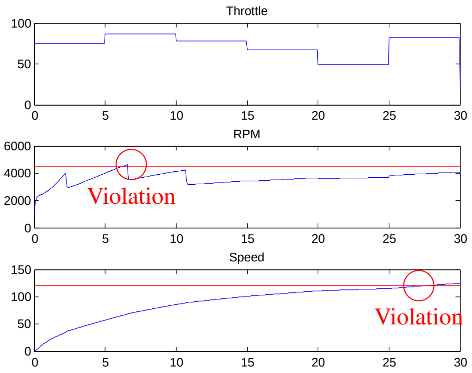


Fig. 2: Falsifying trace for the automatic transmission controller and the requirement that RPM never goes beyond 4500 or speed beyond 120 mph.

real arithmetic and uninterpreted functions, and use an SMT solver to answer the query.

3) We demonstrate the practical applicability of our technique in three case studies: (a) a simple automatic transmission controller, (b) a complex air-fuel control closed-loop model, and (c) an industrial closed-loop model of the airpath-control in an automobile engine model. We also demonstrate the use of the mining technique as a bug-finding tool, showing how it found a bug in the industrial model that was confirmed by a designer. All three case studies use closed-loop models specified in the Simulink language.

The rest of the paper is as follows: In Sec. II, we present a transmission controller as a running example. In Sec. III we present the background, the problem formulation, and an overview of our technique. We present our approach for finding the counterexample to a candidate requirement in Sec. IV, and the procedure for synthesis of parameter values for a template requirement from simulation traces in Sec. V. We collect a set of common requirements for automotive control systems and express them in temporal logic in Sec. VI. Finally, we present three case studies and experimental results for each in Sec. VII, and conclude with a discussion on related work in Sec. VIII.

II. A Running Example

As an illustrative example throughout the paper, we consider a closed-loop model designed for a four-speed automatic transmission controller of a vehicle (shown in Fig. 1). Although this model is not a real industrial model, it has all necessary mechanical components: models for the engine, the transmission, and the vehicle. The transmission block computes the torque converter impeller torque (Ti) and the transmission output torque (Tout) from engine speed (Ne), gear status (Gear), and transmission output speed (Nout). The logic of gear selection for the transmission is implemented using a Stateflow block [?] labeled ShiftLogic. Block ThreshholdCalculation computes the upshift and downshift speed thresholds for the gear shifting logic. The model takes as inputs the percentage of the `throttle` position and the `brake` torque.

The transmission controller has four gears, and the system switches from gear i up to gear $i+1$ or down to gear $i-1$ based on certain conditions on the current gear i, the current vehicle speed and the applied throttle. The threshold speed that causes a shift in the gear is specified using a look-up table that is indexed by the current gear and the applied throttle.

We are interested in the following signals: the vehicle `speed`, transmission `gear` position, and engine speed measured in `RPM` (rotations per minute). Suppose we want to use this controller to ensure the requirement that the engine speed never exceeds 4500 rpm, and that the vehicle never drives faster than 120 mph. After simulating the closed-loop system Fig. 2 shows that these requirements are not met.

However, this negative result does not provide further insight into the model. If a requirement does not hold, we would like to know what *does* hold for the controller, and how narrowly the controller misses the requirement. Such a characterization would shed more light on the working of the system, especially in the context of legacy systems and for reverse engineering the behavior of a very complex system. In the context of this example, it would help to know the maximum speed and RPM that the model can reach, or the minimum dwell time that the transmission enforces to avoid frequent gear shifts. Next, we present a technique to automatically obtain such requirements from the model.

Note that such specific, precise requirements automatically mined from the model can help understand and enforce a high-level requirement. For example, the frequency of gear shifting is correlated with the less precise requirement of “better driving experience” and “better fuel consumption”.

III. Preliminaries and Overview

Signals and Systems. The systems considered in this paper are hybrid dynamical systems, that is systems mixing discrete dynamics (such as the shifting logic of gears) and continuous dynamics (such as the rotational dynamics of the car engine). We define a *signal* as 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, e.g., $\mathbf{p} = (p_1, \cdots, p_n)$. Likewise, a multi-dimensional signal $\mathbf{x}$ is a function from $\mathbb{T}$ to $\mathbb{R}^n$ such that $\forall t \in \mathbb{T}$, $\mathbf{x}(t) = (x_1(t), \cdots, x_n(t))$. A *system* $\mathcal{S}$ (such as a Simulink model) is an input-output state machine: it takes as input a signal $\mathbf{u}(t)$ and computes an output signal $\mathbf{x}(t)$. A *trace* is a collection of output signals resulting from the simulation of a system, i.e., it can be viewed as a multi-dimensional signal. In the following, we use interchangeably the words trace and signal.

Signal Temporal Logic. Temporal logics were introduced in the late 70s by Amir Pnueli [**?**] to reason formally about the temporal behaviors of *reactive* systems – originally input-output systems with Boolean, discrete-time signals. Temporal logics to reason about real-time signals, such as Timed Propositional Temporal Logic [**?**], and Metric Temporal Logic (MTL) [**?**] were introduced later to deal with dense-time signals. More recently, Signal Temporal Logic [**?**] was proposed in the context of analog and mixed-signal circuits as a specification language for constraints on real-valued signals. These constraints, or *predicates* can be reduced to inequalities of the form $\mu = f(\mathbf{x}) \sim \pi$, 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 following 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 as follows: $\Box_I \varphi \triangleq \neg\Diamond_I \neg\varphi$, $\Diamond_I \varphi \triangleq \top \, \mathbf{U}_I \, \varphi$. 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. Formally, the semantics are given as follows:

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

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 $(\mathbf{x}, 0) \models \varphi$.

Hunter et al. [**?**] show that MTL with rational constants (of which STL is a generalization) is as expressive as first order logic with $<$ (a binary order operation), and a family of unary functions $+q$, $q \in \mathbb{Q}$. This indicates the rich expressive power of STL. The kind of properties that cannot be expressed in STL require quantifying over time or parameter values. For example, the following property (inexpressible in STL) defines the standard Lyapunov stability of a system: $\forall\epsilon\exists\delta : (\|\mathrm{x}\| < \delta \Rightarrow \Diamond(\|\mathrm{x}\| < \epsilon))$.

Parametric Signal Temporal Logic (PSTL) is an extension of STL introduced in [**?**] 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. These parameters are divided into two types:

- A *Scale* parameter π is a parameter appearing in predicates of the form $\mu = f(\mathbf{x}) \sim \pi$,
- A *Time* parameter τ is a parameter appearing in an interval of a temporal operator.

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 $v = \{\tau \mapsto 10, \; \pi \mapsto 1.2\}$.

Example III.1. *For the example from Sec. II, suppose we want to specify that the* `speed` *never exceeds* 120 *and* `RPM` *never exceeds* 4500. *The predicate specifying that the speed is above 120 is:* `speed`>120 *and the one for RPM is* `RPM`>4500. *The STL formula expressing these to be always false is:*

$$\psi = \Box(\mathtt{speed} \leq 120) \wedge \Box(\mathtt{RPM} \leq 4500). \qquad \text{(III.1)}$$

To turn this into a PSTL formula, we rewrite by introducing parameters π_{speed} *and* π_{rpm}*:*

$$\varphi(\pi_{speed}, \pi_{rpm}) = \Box(\mathtt{speed} \leq \pi_{speed}) \wedge \Box(\mathtt{RPM} \leq \pi_{rpm}). \qquad \text{(III.2)}$$

The STL formula ψ *expressed in* (III.1) *is then obtained by using the valuation* $v = (\pi_{\mathtt{speed}} \mapsto 120, \pi_{rpm} \mapsto 4500)$.

In this work, we evaluate the satisfaction of STL formulas over finite traces resulting from numerical simulation. In principle, this means that formulas such as ((III.2)) with unbounded horizon cannot be evaluated. In practice, however, we adopt the view of [**?**] where finite traces are completed toward ∞ with constant extrapolation, and assume that the simulation time is long enough to give meaningful results.

Problem III.1. *Given (a) a system* $\mathcal{S}$ *with a set* $\mathcal{U}$ *of inputs, and, (b) a PSTL formula with* n *symbolic parameters* $\varphi(p_1, \ldots, p_n)$ *where* p *could either be scale parameter* π *or time parameter* τ*, the objective is to find a* "tight" *valuation function* v *such that*

$$\forall \mathbf{u} \in \mathcal{U} : \mathcal{S}(\mathbf{u}) \models \varphi(v(p_1), \ldots, v(p_n)).$$

Note that by "*tight*", we mean to enforce mining for non-trivial or not overly conservative requirements. E.g., we are not interested in the requirement that "the car cannot go faster than 500 mph". We define this notion more precisely in Section V-A.

Algorithm 1: Requirement mining algorithm. Note that the initial trace $\mathbf{x}$ is obtained using some nominal input of the system $\mathcal{S}$. The algorithm can also be initiated with a set of traces or inputs. Using particular inputs (extremal or corner cases) can sometimes speed up the convergence.

```
Data: A Model $\mathcal{S}$, a trace $\mathbf{x}$, and a PSTL Formula $\varphi$
Result: An STL formula $\varphi(\mathbf{p})$
1 FTraces $\leftarrow \{\mathbf{x}\}$;
2 while True do
3     $\mathbf{p} \leftarrow$ FindParam(FTraces, $\varphi$);
4     FTraces$_{\text{new}}$ = FalsifyAlgo($\mathcal{S}, \varphi(\mathbf{p})$);
5     if FTraces$_{new}$ == $\emptyset$ then
6         return $\varphi(\mathbf{p})$
7     else
8         FTraces $\leftarrow$ FTraces $\cup$ FTraces$_{\text{new}}$;
```

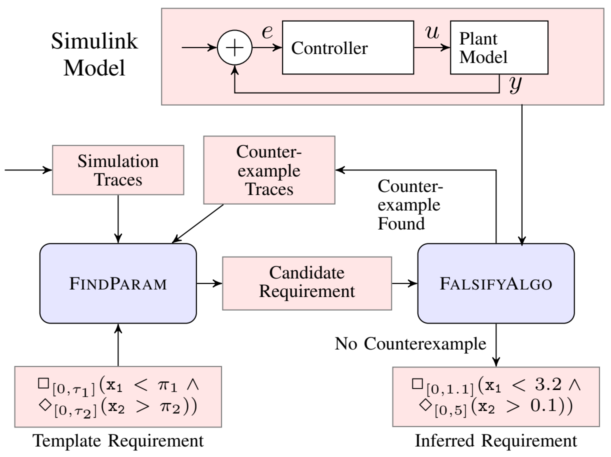


Fig. 3: Flowchart of the requirement mining.

Requirement Mining Algorithm: Overview. Our algorithm (Algorithm 1) for mining STL requirements from the closed-loop model in Simulink is an instance of a counterexample-guided inductive synthesis procedure [?], shown in Fig. 3. It consists of two key components:

1) A falsification engine, which, given a formula φ generates a set of traces $F = \{\mathbf{x}_1, \dots \mathbf{x}_l\}$ such that for all $\mathbf{x}$ in F, there is an input u such that $\mathbf{x}(t) = \mathcal{S}(u)(t) \not\models \varphi$. We denote this functionality by FalsifyAlgo.

2) A synthesis function denoted FindParam that given a set of traces $\mathbf{x}_1, \dots, \mathbf{x}_k$, finds parameters $\mathbf{p}$ such that $\forall i$, $\mathbf{x}_i \models \varphi(\mathbf{p})$. We denote this function by FindParam.

The algorithm terminates if the set F, i.e., the result of FalsifyAlgo$(\mathcal{S}, \varphi(p))$ is empty (the falsification algorithm failed to find a falsifying trace). In the next sections, we detail possible implementations of FalsifyAlgo and FindParam.

IV. Falsification Problem

Recall that we need to implement a function F = FalsifyAlgo$(\mathcal{S}, \varphi)$ such that $\mathbf{x} \in F$ is a valid output signal of a system $\mathcal{S}$ and $\mathbf{x} \not\models \varphi$. Unfortunately, this is an undecidable problem for general hybrid systems. Indeed, if φ is a simple safety property, this problem can be reduced to the reachability problem which is undecidable except for specific subclasses, such as initialized rectangular hybrid automata [?]. For such classes the mining technique can be *complete*, i.e., absence of a counterexample means that we have identified the strongest requirement. Due to its incompleteness for general systems, the falsification tool may not be able to find a counterexample though one exists. We argue that a requirement mined in this fashion is still useful as it is one that FalsifyAlgo is unable to disprove even after extensive simulations, and is thus likely to be close to the actual requirement. An alternative is to use a sound verification tool [?], [?]. However, in our experience, they do not scale to the complex control systems that we consider here. In this paper, we follow the approach taken by the developers of the tool S-Taliro [?] and propose a falsification algorithm based on the minimization of the quantitative satisfaction of a temporal logic formula.

A. Quantitative Semantics of STL

The *quantitative semantics* of STL are defined using a real-valued function ρ of a trace $\mathbf{x}$, a formula φ, and time t satisfying the following property:

$$\rho(\varphi, \mathbf{x}, t) \geq 0 \text{ iff } (\mathbf{x}, t) \models \varphi. \tag{IV.1}$$

Quantitative semantics capture the notion of *robustness of satisfaction* of φ by a signal $\mathbf{x}$, i.e., whenever the absolute value of $\rho(\varphi, \mathbf{x}, t)$ is large, a change in $\mathbf{x}$ is less likely to affect the Boolean satisfaction (or violation) of φ by $\mathbf{x}$. In [?], different quantitative semantics for STL have been proposed. Without loss of generality, an STL predicate μ can be identified to an inequality of the form $f(\mathbf{x}) \geq 0$ (the use of strict or non strict inequalities is a matter of choice and other inequalities can be transformed into this form). The quantitative semantics of STL are then defined inductively using the following rules:

$$\rho(\mu, \mathbf{x}, t) = f(\mathbf{x}(t)) \tag{IV.2}$$
$$\rho(\neg\varphi, \mathbf{x}, t) = -\rho(\varphi, \mathbf{x}, t) \tag{IV.3}$$
$$\rho(\varphi_1 \wedge \varphi_2, \mathbf{x}, t) = \min(\rho(\varphi_1, \mathbf{x}, t), \rho(\varphi_2, \mathbf{x}, t)) \tag{IV.4}$$
$$\rho(\varphi_1 \mathbf{U}_I \varphi_2, \mathbf{x}, t) = \sup_{t' \in t \oplus I} \min \begin{pmatrix} \rho(\varphi_2, \mathbf{x}, t'), \\ \inf\limits_{t'' \in [t, t')} \rho(\varphi_1, \mathbf{x}, t'') \end{pmatrix} \tag{IV.5}$$

Then it can be shown [?] that ρ satisfies (IV.1) and thus defines a quantitative semantics for STL. Additionally, by combining (IV.5), and $\Box_I \varphi \triangleq \neg \Diamond_I \neg \varphi$, we get:

$$\rho(\Diamond_I \varphi, \mathbf{x}, t) = \sup_{t' \in t + I} \rho(\varphi, \mathbf{x}, t') \tag{IV.6}$$
$$\rho(\Box_I \varphi, \mathbf{x}, t) = \inf_{t' \in t \oplus I} \rho(\varphi, \mathbf{x}, t') \tag{IV.7}$$

Example IV.1. *Consider again the STL property:*

$$\varphi = \Box(\texttt{speed} \leq 120) \wedge \Box(\texttt{RPM} \leq 4500).$$

It has two predicates, say $\mu_1 : \texttt{speed} \leq 120$ *and* $\mu_2 : \texttt{RPM} \leq 4500$. *To put them into the standard form* $\mu_i : f_i(\mathbf{x}) \geq 0$, *we define* $\mathbf{x} = (\texttt{speed}, \texttt{RPM})$, $f_1(\mathbf{x}) = 120 - \texttt{speed}$ *and* $f_2(\mathbf{x}) = 4500 - \texttt{RPM}$. *From* (*IV*.2), *we get*

$$\rho(\texttt{speed} \leq 120, \mathbf{x}, t) = 120 - \texttt{speed}(t).$$

Algorithm 2: FALSIFYALGO algorithm.

```
Data: A Model $\mathcal{S}$ and an STL Formula $\varphi$
Result: A falsifying input $\mathbf{u}^*$ or $\mathcal{S} \models \varphi$.
1 Solve $\rho^* = \min_{\mathbf{u} \in \mathcal{U}} \rho(\varphi, \mathcal{S}(\mathbf{u}), 0)$
2 if $\rho^* < 0$ then return $\mathbf{u}^* = \arg\min_{\mathbf{u} \in \mathcal{U}} \rho(\varphi, \mathcal{S}(\mathbf{u}), 0)$
3 else return $\mathcal{S} \models \varphi$
```

Applying rule $(IV.7)$ for the semantics of $\Box$, we get:

$$\rho(\Box(\texttt{speed} \leq 120), \mathbf{x}, t) = \inf_{t \in \mathbb{T}} (120 - \texttt{speed}(t)).$$

Similarly for μ_2,

$$\rho(\Box(\texttt{RPM} \leq 4500), \mathbf{x}, t) = \inf_{t \in \mathbb{T}} (4500 - \texttt{RPM}(t)).$$

Finally, by applying rule $(IV.4)$:

$$\rho(\varphi, \mathbf{x}, t) = \min(\inf_{t \in \mathbb{T}} (120 - \texttt{speed}(t)), \inf_{t \in \mathbb{T}} (4500 - \texttt{RPM}(t)).$$

In other word, the resulting satisfaction function ρ looks for the maximum speed and RPMs over time and returns the minimum of the differences with the thresholds 120 *and* 4500.

Note that, in the previous example, `speed` and `RPM` are measured in different units. However, the standard quantitative semantics for STL does not capture this difference. BREACH supports weighted STL (WSTL) semantics which associate a weight with each predicate to normalize the numerical difference and improve the expressiveness [**?**].

B. Solving the Falsification Problem

The objective of the falsification problem can be reduced to: given an STL formula φ, find a signal $\mathbf{u}$ such that $\mathcal{S}(\mathbf{u}) \not\models \varphi$. Following the above definitions, this is equivalent to finding a trace $\mathbf{x}$ of $\mathcal{S}$ such that $\rho(\varphi, \mathbf{x}, 0) < 0$. A common approach to solve this problem, described in the Algorithm 2, is to frame it as an *optimization* problem, where the objective function (to minimize) is $\rho(\varphi, \mathbf{x}, 0)$ and the decision variable is $\mathbf{u}$.

The undecidability of the falsification problem is reflected here in the fact that the minimization problem (Line 1 in Algorithm 2) is a general non-linear optimization problem for which no solver can guarantee convergence, uniqueness or even existence of a solution. On the other hand, many heuristics can be used to find an approximate solution. In a series of recent papers, the authors of S-TALIRO proposed and implemented different strategies, namely Monte-Carlo [**?**], ant-colony optimization [**?**] and the cross entropy method [**?**]. Going into the details of these methods and their comparison is beyond the scope of the paper. In previous work, we document the use of S-TALIRO as a falsification tool [**?**]. In this paper, we focus on using the falsification engine in BREACH to attack (Line 1 in Algorithm 2) as follows:

1) Define the space of permissible input signals with the help of m *input parameters* $\mathbf{k} = (k_1, \ldots, k_m)$ that take values from a set $\mathcal{P}_u$, and a generator function g such that $\mathbf{u}(t) = g(v(\mathbf{k}))(t)$ is a permissible input signal for $\mathcal{S}$ for any valuation $v(\mathbf{k}) \in \mathcal{P}_u$.
2) Sample the space of the signal-parameters uniformly at random to obtain N_{init} distinct valuations $v_i(\mathbf{k}) \in \mathcal{P}_u$.
3) For $i \leq N_{\text{init}}$, solve $\min_{v(\mathbf{k}) \in \mathcal{P}_u} \rho(\varphi, \mathcal{S}(g(v(k))), 0)$ using Nelder-Mead non-linear optimization algorithm and $v_i(\mathbf{k})$ as an initial guess.
4) Return the minimum ρ thus found.

One motivation for implementing a falsification module in BREACH has been to get more flexibility in the definition of input parameters than available in existing implementations of falsifiers such as S-TALIRO. For example, if permissible input signals are step functions, then the input parameters would characterize the amplitude of the step, and the time at which the step input is applied. Note that g does not necessarily generate all possible inputs to the system. However, it is useful in a very generic way to restrict the search space of possible input signals. It is worth mentioning again that many different strategies exist to solve the falsification problem using optimization algorithms. The particular strategy described above was chosen to allow some trade-off between global randomized exploration (by the number N_{init} of random initial valuations) and local optimization (using Nelder-Mead) exploiting the gradient of the satisfaction function. Experiments in Section VII-A illustrates the importance of this trade-off.

V. PARAMETER SYNTHESIS

A. Parameter Synthesis Algorithm

We now discuss the function FINDPARAM. Recall that given a trace[2] $\mathbf{x}$, we need to find a valuation v for the parameters $p_1, \ldots, p_n$, of φ such that $\mathbf{x}$ satisfies $\varphi(v(p_1), \ldots, v(p_n))$ (which we sometimes abbreviate in $\varphi(v)$ in the following). In the following, we call such a valuation a *valid valuation* for $\mathbf{x}$ and φ (or simply a valid valuation if $\mathbf{x}$ and φ are clear from the context). This problem can be treated as a dual of the falsification problem: instead of minimizing the satisfaction function in an attempt to make it negative, we can try to maximize it in an attempt to make it positive, i.e., to make the formula φ true. However, this approach is not directly applicable in the context of this work, due to the additional *tightness* requirement on the mined parameters. The rationale is that for a specification to be useful it should not be too conservative: it is of not much use to know that a vehicle speed will never exceed 200 miles per hour. Now maximizing the satisfaction function will precisely tend toward the most conservative parameters: not exceeding 200 miles per hour is a very robustly true property, i.e., with a high satisfaction function value. A more useful piece of information is to know that a car can go up to 100 miles per hour, but not 101. More generally, for each parameter mined in a formula, when it is possible, we require that a change of some amplitude $\delta > 0$ in a given direction makes the formula false. We formalize this with the following notion of δ-satisfaction:

Definition V.1. *Given $1 \leq i \leq n$ and $\delta_i > 0$, the signal $\mathbf{x}$ δ_i-satisfies $\varphi(v)$, denoted as $\mathbf{x} \models_{\delta_i} \varphi(v)$, iff $\mathbf{x} \models \varphi(v)$ and*

[2]We restrict our presentation to one trace even though in Algorithm 1, FINDPARAM is applied to a set of traces. The generalization to multiple traces is straightforward.

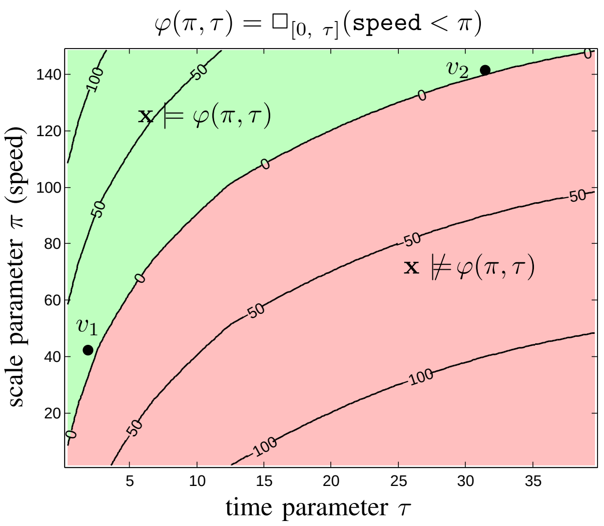


Fig. 4: Validity domain of a simple formula for a trace $\mathbf{x}$ obtained from the automatic transmission model. The FINDPARAM algorithm will return valuation v_1 (resp. v_2) depending if time (resp. scale) parameter is optimized first. The contour lines are isolines for the satisfaction function ρ.

there exists a valuation v' such that $|v(p_i) - v'(p_i)| \leq \delta_i$ and $\mathbf{x} \not\models \varphi(v')$. In that case, v is called a δ_i-tight valuation for $\mathbf{x}$ and φ.

When a valuation is tight for all p_i, we can omit the index i and call it a δ-tight valuation for $\mathbf{x}$ and φ, where δ is the n-dimensional vector $\delta = (\delta_1, \ldots, \delta_n)$.
In general, there is no guarantee of existence or uniqueness of δ-tight valuations. We note

$$D(\varphi, \mathbf{x}) \triangleq \{v(p) \text{ s.t. } \mathbf{x} \models \varphi(v(p))\}$$

the *validity domain* of φ and $\mathbf{x}$, i.e., the set of valid valuations for $\mathbf{x}$ and φ. The existence of a valid valuation is given by $D(\varphi, \mathbf{x}) \cap \mathcal{P} \neq \emptyset$, where $\mathcal{P}$ is feasible parameter range. A δ-tight valuation v is such that there exists another valuation v' such that $v'(p)$ is at distance at most δ from $v(p)$ and is not in $D(\varphi, \mathbf{x})$. Intuitively, this means that the valuation is closed to the *boundary* of the validity domain. Under certain regularity conditions, one can show that on such boundary, the satisfaction function ρ is equal to 0. On Fig. 4, we represent the validity domain for a simple property and a signal. In this example, the boundary of the validity domain is exactly given by the isoline $\rho(\varphi(v), \mathbf{x}, 0) = 0$, and any valuation above this line at a distance less than δ is a δ-tight valuation. This suggests a generic optimization strategy to solve the parameter synthesis problem. If we note $\mathcal{B}_\delta(v) = \{v' \text{ s.t. } \max_i |v(p_i) - v'(p_i)| < \delta\}$, then a δ-tight valuation, if it exists, is given by

$$v^* = \arg\min_v |\rho(\varphi(v), \mathbf{x}, 0)| \quad \text{(V.1)}$$
$$\text{s.t. } v(p) \in D(\varphi, \mathbf{x}) \cap \mathcal{P} \quad \text{(V.2)}$$
$$\mathcal{B}_\delta(v) \not\subset D(\varphi, \mathbf{x}) \quad \text{(V.3)}$$

A simple practical solution for (V.1-V.3) is to solve (V.1) using the same strategy as for the optimization-based falsification approach, then check (V.2-V.3) on the solution found. Clearly, other approaches are possible, but the problem is difficult in general without additional assumptions. In this work we focus on the specific situation where the PSTL formula is *monotonic* in its parameters, as described in the following sections.

Algorithm 3: FINDPARAM algorithm.

```
Data: A trace $\mathbf{x}$, a PSTL Formula $\varphi$, and parameter set
      $\mathcal{P}$, $\delta > 0$
Result: A valuation $v$ s.t. $\mathbf{x} \models_\delta \varphi(v)$
1 Find $v_\top$ s.t. $\mathbf{x} \models \varphi(v_\top)$ or return $\varphi$ unsat.;
2 Find $v_\bot$ s.t. $\mathbf{x} \not\models \varphi(v_\bot)$ or return $v$ maybe not tight;
3 Let $v = v_\top$;
4 for $i = 1$ to $n$ do
5     Find $v_i$ and set $v(p_i) = v_i$ s.t. $\mathbf{x} \models_{\delta_i} \varphi(v)$
```

B. Computing δ-tight Valuations for Monotonic Formulas

Intuitively, a formula is monotonic if when it is satisfied with a valuation v, then it is satisfied by any valuation v' greater than v. For example, if the car cannot go faster than 100 mph, it cannot go faster than 101, 150, 200 or any speed above 100 mph. Formally:

Definition V.2. *A PSTL formula $\varphi(p_1, \cdots, p_n)$ is monotonically increasing with respect to p_i if for every signal $\mathbf{x}$,*

$$\forall v, v', \mathbf{x} \models \varphi(\ldots, v(p_i), \ldots), \quad v'(p_i) \geq v(p_i) \Rightarrow \mathbf{x} \models \varphi(\ldots, v'(p_i), \ldots) \quad \text{(V.4)}$$

It is monotonically decreasing if this holds when replacing $v'(p_i) \geq v(p_i)$ with $v'(p_i) \leq v(p_i)$.

Asarin et al. [?] noted that, if the formula is monotonic, the boundary of the validity domain has the properties of a Pareto surface for which there are efficient computational methods, basically equivalent to multi-dimensional binary search. Here we propose an algorithm for monotonic formulas that takes advantage of this property (Algorithm 3) to find a valuation satisfying (V.2-V.3), i.e., a δ-tight valuation. It starts by trying to find a valuation $v_\top$ that satisfies the property and a valuation $v_\bot$ that violates it in a parameter range $\mathcal{P}$ provided by the user. By property of monotonicity, it is sufficient to check the corners of $\mathcal{P}$ for the existence of $v_\top$ and $v_\bot$. Then, each parameter i is adjusted using a binary search initialized with $v_\top(p_i)$ and $v_\bot(p_i)$. The user can choose which parameter to optimize in priority by specifying a different order for the input parameters.

Example V.1. *Consider $\varphi(\pi, \tau) = \Box_{[0,\ \tau]}(\texttt{speed} < \pi)$ and the scenario that the vehicle constantly accelerates at $\texttt{throttle} = 100$. The validity domain of φ is plotted on Fig. 4. The algorithm will return different values depending on the tightness parameter δ and if we order the parameters as (π,τ) or (τ,π). Here, the order represents the preference in optimizing a parameter over the other when mining for a tight specification.*

Formula	Monot.	Time (sec)
$\square_{(0,\infty)}(x < \pi)$	+	0.09
$\square_{[s,s+1]}(x \geq 3 \Rightarrow \Diamond_{(0,\infty)} x < 3)$	–	0.10
$\square_{(0,100)}((x < \pi) \Rightarrow \Diamond_{(0,5)}(x > \pi))$	–	0.09
$\mathtt{gear}_i \mathbf{U}_{(s,s+5)} \mathtt{gear}_{i+1}$	*	0.13

TABLE I: Proving monotonicity with an SMT solver.

C. Satisfaction monotonicity

In this section, we provide additional results on monotonicity of PSTL formulas. We first show that checking if an arbitrary PSTL formula is monotonic in a given parameter is undecidable.

Theorem V.1. *The problem of checking if a PSTL formula $\varphi(\mathbf{p})$ is monotonic in a given parameter p_i is undecidable.*

Proof. First, we observe that STL is a superset of MTL. We know from [**?**] that the satisfiability problem for MTL is undecidable. Thus, it follows that the satisfiability problem for STL is also undecidable. This, in turn, implies undecidability of the satisfiability problem of PSTL with at most one parameter (denoted as PSTL-1-SAT). We now show that PSTL-1-SAT can be reduced to a special case of the problem of checking monotonicity of a PSTL formula.

Let $\varphi(\mathbf{p})$ be an arbitrary PSTL formula where the set of parameters $\mathbf{p}$ is the singleton set with one time parameter τ (thus, $\tau \geq 0$). Construct the formula $\psi(\mathbf{p}) \doteq (\tau = 0) \vee \varphi(\mathbf{p})$.

Consider the monotonicity query for $\psi(\mathbf{p})$ in parameter τ:

$$\forall v, v', \mathbf{x} : \ [\mathbf{x} \models \psi(v(\tau)) \ \wedge \ v(\tau) \leq v'(\tau)] \ \Rightarrow \ \mathbf{x} \models \psi(v'(\tau)).$$

Consider the specialization of this formula for the case $v(\tau) = 0$. Note that, in this case, $\psi(0) = \top$, and that $v'(\tau) \geq 0$ for all v'. Thus, the query simplifies to $\forall v', \mathbf{x} : \ \mathbf{x} \models \psi(v'(\tau))$, which is checking the validity of the PSTL formula $\psi(\tau)$.

Thus, if one needs to check monotonicity of PSTL formula φ in one parameter τ, one needs to check that the negation of $\psi(\tau)$ is unsatisfiable. Thus the above specialization of the problem of checking the monotonicity of PSTL formulas is also undecidable, implying undecidability of the general case. □

Monotonicity is closely related to the notion of polarity introduced in [**?**], in which syntactic deductive rules are given to decide whether a formula is monotonic based on the monotonicity of its subformulae. Thus, one way to tackle undecidability is to first query if the given PSTL formula belongs to the syntactic class described in [**?**]. Unfortunately, the syntactic rules described therein are not complete; there are monotonic PSTL formulas that do not belong to this syntactic class, for instance, formulas with intervals in which both endpoints are parameterized, such as the following:

$$\square_{[\tau,\tau+1]}((x \geq 3) \Rightarrow \Diamond_{(0,\infty)}(x < 3)) \qquad \text{(V.5)}$$

Next, we show how we can use SMT solving to query monotonicity of a formula. If the SMT solver succeeds, it tells us that the formula is monotonic and allows us to use a more efficient search in the parameter space. For instance, we were able to show that the PSTL formula represented in (V.5) is monotonically decreasing in the parameter τ.

Encoding PSTL as constraints. Given a PSTL formula φ, we define the SMT encoding of φ in a fragment of first-order logic with real arithmetic and uninterpreted functions. Let $\mathcal{E}(\varphi)$ denote the encoding of φ, which we define inductively as follows:

– Consider a constraint $\mu \triangleq g(\mathbf{x}) > \tau$, where $\mathbf{x} = (x_1, \ldots, x_n)$. We model each signal x_i as an uninterpreted function χ_i from $\mathbb{R}$ to $\mathbb{R}$. We create a new free variable t of the type `Real` and replace each instance of the signal x_i in $g(\mathbf{x})$ by $\chi_i(t)$. We assume that the function g itself has a standard SMT encoding. For example, consider the formula $g(\mathbf{x}) > \tau$, where $\mathbf{x} = \{x_1, x_2\}$, and $g(\mathbf{x}) = 2 * x_1 + 3 * x_2$. Then $\mathcal{E}(\mu)$ is: $2 * \chi_1(t) + 3 * \chi_2(t) \ > \tau$.

– For Boolean operations, the SMT encoding is inductively applied to the subformulas, i.e., if $\varphi = \neg\varphi_1$, then $\mathcal{E}(\varphi) = \neg\mathcal{E}(\varphi_1)$. If $\varphi = \varphi_1 \wedge \varphi_2$, then first we ensure that if $\mathcal{E}(\varphi_1)$ and $\mathcal{E}(\varphi_2)$ both have a free time-domain variable, then we make it the same variable, and then, $\mathcal{E}(\varphi) = \mathcal{E}(\varphi_1) \wedge \mathcal{E}(\varphi_2)$. Note that as a consequence, there is at most one free time-domain variable in any subformula.

– Consider $\varphi = \mathrm{H}_{(a,b)}(\varphi_1)$, where a, b are constants or parameters, and H is a unary temporal operator (i.e., $\Diamond, \square$). There are two possibilities:

(1) The SMT encoding $\mathcal{E}(\varphi_1)$ has one free variable t. In this case, we bound the variable t over the interval (a, b) using a quantifier that depends on the type of the temporal operator H. With $\Diamond$ we use $\exists$ as the quantifier, and with $\square$ we use $\forall$. E.g., let $\varphi = \Diamond_{(2.3,\tau)}(x > \pi)$, then $\mathcal{E}(\varphi)$ is:

$$\exists t : \quad (2.3 < t < \tau) \wedge (\chi(t) > \pi).$$

(2) The SMT encoding $\mathcal{E}(\varphi_1)$ has no free variable. This can only happen if φ_1 is $\top$ or $\bot$, or if all variables in φ_1 are bound. In the former case, the encoding is done exactly as in Case 1. In the latter case, the encoding proceeds as before, but all bound variables in the scope are *additionally offset* by the top-level free variable. Suppose, $\varphi = \square_{(0,\infty)}\Diamond_{(1,2)}(x > 10)$. Then, the encoding of the inner $\Diamond$-subformula has no free variable. Note how the bound variable of this formula is offset by the top-level free variable in the underlined portion in $\mathcal{E}(\varphi)$ below:

$$\forall t : [\exists u : [\underline{(t + 1 < u < t + 2)} \wedge (\chi(u) > 10)]].$$

– Consider $\varphi = \varphi_1 \mathbf{U}_{(a,b)} \varphi_2$, where a, b are constants or parameters. For simplicity, consider the case where φ_1 and φ_2 have no temporal operators, i.e., $\mathcal{E}(\varphi_1)$ and $\mathcal{E}(\varphi_2)$ both have exactly one free variable each. Let t_1 be the free variable in $\mathcal{E}(\varphi_1)$ and t_2 the free variable in $\mathcal{E}(\varphi_2)$. Then $\mathcal{E}(\varphi)$ is given by the formula:

$$\exists t_2 : [(t_2 \in (a, b)) \wedge \mathcal{E}(\varphi_2) \wedge \forall t_1 : [(t_1 \in (a, t_2)) \Rightarrow \mathcal{E}(\varphi_1))].$$

If φ_1, φ_2 contain no free variables, then t_1, t_2 are respectively used to offset all bound variables in their scope as before.

Using an SMT solver to check monotonicity. To check monotonicity, we check the satisfiability of the negation of each of the following assertions:

$$\mathcal{E}(\varphi(\tau)) \wedge (\tau > \tau') \wedge \neg\mathcal{E}(\varphi(\tau'))$$
$$\mathcal{E}(\varphi(\tau)) \wedge (\tau < \tau') \wedge \neg\mathcal{E}(\varphi(\tau'))$$

If either of these queries is unsatisfiable, then it means that satisfaction of φ is indeed monotonic in τ. If both queries are satisfiable, then it means that there is an interpretation for the (uninterpreted) function representing the signal $\mathbf{x}$ and valuations for τ, τ' which demonstrate the non-monotonicity of φ. We conclude by presenting a small sample of formulas for which we could prove or disprove monotonicity using the Z3 SMT solver [**?**] in Table I. The symbols **+**, **–**, and ***** represent monotonically increasing, monotonically decreasing, and non-monotonic formulas respectively.

VI. Requirement Templates for Automotive Control Systems

We now discuss a set of PSTL formulas that serve as useful template requirements for continuous and hybrid dynamical systems. We start with a general discussion on useful templates for such systems, and then present particular templates specialized to express requirements on closed-loop control systems, with an emphasis on the automotive domain. Most of the requirements discussed herein were obtained by discussions with designers, and correspond to well-known metrics and tests used to judge design quality. In what follows, we use T to represents the simulation time horizon.

A. Temporal Requirements on Hybrid behaviors

By hybrid behaviors, we mean typical behaviors of hybrid dynamical systems, i.e., a continuous-time evolution of the continuous states of the system consistent with a given set of ordinary differential equations, interleaved with discrete transitions corresponding to a discrete mode-change.

Dwell-time Requirements. A common requirement on a switched or hybrid system is that the system should not switch discrete modes (chatter) too often. This can be achieved by enforcing that the system *dwells* in a given discrete mode for a desired minimum amount of time. Let m be a discrete-valued signal denoting the system mode. Then the requirement specifying that the dwell-time is at least τ is specified as follows:

$$\Box_{[0,T]}\left(\begin{array}{c}(m \neq m_j)\wedge \\ \Diamond_{[0,\epsilon]}(m = m_j)\end{array}\right) \Rightarrow \Box_{[\epsilon,\tau]}(m = m_j) \qquad \text{(VI.1)}$$

Timed and Untimed Safety. A basic safety requirement for a hybrid or continuous system can be specified as follows:

$$\Box_{[0,T]}\varphi(m, \mathbf{y}). \qquad \text{(VI.2)}$$

Here, $\mathbf{y}$ is the continuous state of the system, m is the discrete mode, and $\varphi(m, \mathbf{y})$ is a bounded-time STL formula over the hybrid state-space. For example, $\varphi(m, \mathbf{y})$ could be the propositional formula $(m = m_0) \wedge (|x| < c)$. A minor variation on a basic safety requirement is *timed safety requirement*; here, the outermost temporal operator $\Box$ is also bounded by some time τ. For example, the property: $\Box_{[0,\tau]}(x < c)$.

Timed Inevitability. A timed inevitability requirement specifies that a certain temporal behavior must happen before a certain time τ expires. This is useful to specify timed reachability of a certain mode or a certain region in the state space. The template for such a property is as follows:

$$\Diamond_{[0,\tau]}\varphi(m, \mathbf{y}). \qquad \text{(VI.3)}$$

Here, $\varphi(m, \mathbf{y})$ is some bounded-time STL formula. For example, $\varphi(m, \mathbf{y})$ could be the propositional formula $(m = m_0)$.

B. Temporal Requirements on Control Systems

1) Input Profiles: So far, the requirements discussed in this paper are temporal specifications on the behavior of output signals or states of a closed-loop control system. Typically, a control system is designed to regulate the behavior of the state or outputs of a dynamical system when stimulated by an external disturbance or to respond to an external input. To quantify the performance of a control system, control designers typically make certain assumptions about the disturbances or external inputs. Often, these assumptions can be characterized using a STL formula.

1) A common assumption for control systems is for disturbance signals to have a bounded norm. Suppose $u(t)$ is a disturbance signal, then a disturbance signal with the infinity norm bounded above by D is specified by the STL formula: $\Box_{[0,T)}|u| < D$.

2) One of the basic tests that control designers use to understand the efficacy of their designs is a step response. A step input can be specified by the STL formula:

$$\Diamond_{[d,d+\delta)}(u = u_\ell) \wedge \Diamond_{[0,\delta)}(u = u_h).$$

Here, d is an initial delay, u_ℓ is a constant specifying the input value before the step, $u_h - u_\ell$ is the amplitude of the step, and δ is a small number representing the smallest simulation step time.

3) Also of interest to control designers is a pulse response. To define a pulse, we first define some parameterized events. Here u_ℓ is a parameter representing the input value before the pulse, and $u_h - u_\ell$ is a parameter representing the pulse amplitude.

$$\begin{array}{l}\texttt{rise} \equiv (u = u_\ell) \Rightarrow \Diamond_{[0,\delta)}(u = u_h) \\ \texttt{fall} \equiv (u = u_h) \Rightarrow \Diamond_{[0,\delta)}(u = u_\ell)\end{array}$$

Now, a pulse signal of period p, initial delay d, and pulse width w can be specified using the STL formula:

$$\Box_{[d,T]}\left(\begin{array}{l}\big(\texttt{rise} \Rightarrow \Diamond_{[w,w]}\texttt{fall}\big)\wedge \\ \big(\texttt{fall} \Rightarrow \Diamond_{[p-w,p-w]}\texttt{rise}\big)\end{array}\right)$$

We remark that other input profiles such as sinusoidal inputs, ramp inputs can also be specified using STL.

2) Control-theoretic Requirements on Outputs: In general, the form of requirements of interest for control systems takes the form of $\varphi_I \Rightarrow \varphi_O$, where φ_I is a STL property characterizing the input profile. Note that φ_I can be the property *true*, i.e., no assumptions are placed on the input. In what follows, we focus on the RHS of the above implication, i.e., on the requirements on the output signals.

Overshoot/Undershoot. An overshoot/undershoot requirement is a basic safety requirement. As output signals often represent physical quantities in a system, e.g., pressure, temperature, acceleration, etc., control designers try to impose requirements on the maximum "overshoot" or "undershoot", i.e., maximum and minimum excursions of a given signal from a reference value $\mathbf{y}_{ref}$. The following STL requirements respectively specify limits c_1 and c_2 on the maximum overshoot and undershoot of a signal x between the times t_1 and t_2.

$$\square_{[t_1,t_2]}(\mathbf{y} - \mathbf{y}_{ref} < c_1) \tag{VI.4}$$

$$\square_{[t_1,t_2]}(\mathbf{y}_{ref} - \mathbf{y} < c_2) \tag{VI.5}$$

As mentioned before, a step response is a standard technique in control theory to gauge the temporal behavior of a system (especially linear dynamical systems), and is often used to estimate maximum overshoot and undershoot.

Settling Time. A settling time requirement is a safety requirement. In a control system, a disturbance or a change in an input may lead to transient oscillations in the regulated output. It is important for these oscillations to be within the *settling region*, i.e., a region specifying the tolerated deviations from the given reference value, and for them to settle to the reference within a specified *settling time*. Let `disturbance` denote a disturbance event. Let $\mathbf{y}$ be the output signal of interest, let $\mathbf{y}_{ref}$ be the reference value for $\mathbf{y}$, and let $|\mathbf{y} - \mathbf{y}_{ref}| < \delta$ denote the settling region. The STL property specifying the requirement that the settling time is less than η is given below:

$$\square_{[0,T]}\left(\texttt{disturbance} \Rightarrow \square_{[\eta,T]}\left(|\mathbf{y} - \mathbf{y}_{ref}| < \delta\right)\right) \tag{VI.6}$$

Error measurement. In any control system, an important quantity is the error between the desired reference value and the actual signal. A standard way of measuring this error is the *root mean square* (RMS) value of the error over time. We first define an RMS error signal:

$$\mathbf{y}_{rms}(t) = \sqrt{\frac{1}{t}\int_0^t (\mathbf{y}(\tau) - \mathbf{y}_{ref})^2 d\tau}$$

RMS error is essentially the value of $\mathbf{y}_{rms}(t)$ at $t = T$. The following STL formula specifies that the RMS error is always less than c; note that this is a timed inevitability requirement.

$$\diamond_{[T,T]}(\mathbf{y}_{rms} < c) \tag{VI.7}$$

VII. Case Studies

In what follows, we present three case studies of requirement mining from the automotive doman. The first is the running example described in Sec. II, the second is an air-fuel ratio-control benchmark model [?], and the third is an industrial-scale experimental model of an airpath controller for a diesel engine.

A. Automatic Transmission Model

For the model described in Sec. II, we tested different template requirements:

1) Requirement $\varphi_{\texttt{sp\_rpm}}(\pi_1, \pi_2)$ specifying that always the `speed` is below π_1 and `RPM` is below π_2 :

$$\square(\texttt{speed} < \pi_1) \wedge \square(\texttt{RPM} < \pi_2).$$

2) Requirement $\varphi_{\texttt{rpm100}}(\tau, \pi)$ specifying that the vehicle cannot reach the speed of 100 mph in τ seconds with `RPM` always below π:

$$\neg(\diamond_{[0,\tau]}(\texttt{speed} > 100) \wedge \square(\texttt{RPM} < \pi)).$$

3) Requirement $\varphi_{\texttt{stay}}(\tau)$ specifying that whenever the system shifts to gear 2, it dwells in gear 2 for at least τ seconds:

$$\square\left(\left(\begin{array}{l}\texttt{gear} \neq 2 \ \wedge \\ \diamond_{[0,\varepsilon]}\texttt{gear} = 2\end{array}\right) \Rightarrow \square_{[\varepsilon,\tau]}\texttt{gear} = 2\right).$$

Here, the left-hand-side of the implication captures the *event* of the transition to gear 2 from another gear. The operator $\diamond_{[0,\varepsilon]}$ here is an MTL substitute for a *next-time* operator. With dense time semantics, ε should be an *infinitesimal* quantity, but in practice, we use a value close to the simulation time-step.

The above requirements have strong correlation with the quality of the controller. The first is a safety requirement characterizing the operating region for the engine parameters `speed` and `RPM`. The second is a measure of the performance of the closed loop system. By mining values for τ, we can determine how fast the vehicle can reach a certain speed, while by mining π we find the lowest `RPM` needed to reach this speed. The third requirement encodes undesirable transient shifting of gears. Rapid shifting causes abrupt output torque changes leading to a jerky ride.

Results on the mined specifications are given in Table II. We used the Z3 SMT solver [?] to show that all of the requirements are monotonic. For the second template, we tried two possible orderings for the parameters. By prioritizing the time parameter τ, we obtained the δ-tight requirement that the vehicle cannot reach 100 mph in less than 12.2s (we set δ to 0.1). As the requirement mined is δ-tight, it means that we found a trace for which the vehicle reaches 100 mph in 12.3s. Similarly, by prioritizing the scale parameter π, we found that the vehicle could reach 100 mph in 50s keeping the `RPM` below 3278 ($\delta = 5$ in that case). For the third requirement, we found that the transmission controller could trigger a transient shift as short as 0.056s. This corresponds to the up-shifting sequence 1-2-3. Using a variant of the requirement (not shown here), we verified that a (definitely undesirable) short transient sequence of the form 1-2-1 or 3-2-3 was not possible.

Based on results shown in Table II and our experience, we make some observations:

• The FindParam algorithm takes in general significantly less time than the FalsifyAlgo algorithm in the mining process. As can be expected, there is a correlation between the number of simulations and the time spent in the falsification process, and between the number of iterations and the time spent in parameter synthesis.

• The space of input signals needs to be parameterized with a sensible number of signal parameters. If too many parameters are used, the search space is too big and falsification becomes difficult. This is demonstrated in the 4 first instances in Table II which are all performed on formula $\varphi_{\texttt{sp\_rpm}}$ with different input parameterization. For this formula, it is straightforward to obtain exact tigth parameter values since the maximum

	Template	dim. P_u	N_{init}	Parameter mined	Nb. Sim.	Nb. Iter.	Fals. time (sec)	Synth. time (sec)
1	$\varphi_{\texttt{sp\_rpm}}(\pi_1, \pi_2)$	1	1	$\pi_1 = 155.5$ mph, $\pi_2 = 4857$ rpm	120	11	75.89	34.26
2	$\varphi_{\texttt{sp\_rpm}}(\pi_1, \pi_2)$	2	1	$\pi_1 = 155.5$ mph, $\pi_2 = 4857$ rpm	513	24	199.02	94.23
3	$\varphi_{\texttt{sp\_rpm}}(\pi_1, \pi_2)$	2	10	$\pi_1 = 155.5$ mph, $\pi_2 = 4857$ rpm	1482	23	534.65	80.01
4	$\varphi_{\texttt{sp\_rpm}}(\pi_1, \pi_2)$	4	10	$\pi_1 = 155.5$ mph, $\pi_2 = 4857$ rpm	2362	35	862.26	185.31
5	$\varphi_{\texttt{rpm100}}(\pi, \tau)$	3	2	$\pi = 3682$ rpm, $\tau = 49.90$ s	1475	23	885.31	143.83
6	$\varphi_{\texttt{rpm100}}(\tau, \pi)$	3	2	$\tau = 12.20$ s, $\pi = 4997$ rpm	340	3	182.32	2.60
7	$\varphi_{\texttt{stay}}(\pi)$	3	5	$\tau = 1.05$ s	77	5	50.21	6.84
8	$\varphi_{\texttt{stay}}(\pi)$	3	100	$\tau = 0.1367$ s	243	7	116.14	9.40
9	$\varphi_{\texttt{stay}}(\pi)$	3	1000	$\tau = 0.0586$ s	1246	8	608.24	9.55

TABLE II: Results on mining requirements for the automatic transmission control model. For each instance, we indicate the template formula used, the dimensionality of the input parameter space, the value of N_{init} used, the parameter values returned, the number of simulations, the number of iterations of the mining algorithm, the time spent in falsification and the time spent in parameter synthesis.

speed and rpm corresponds to the case of constant acceleration with maximum throttle. The correct values are found for all instances, but the time needed to obtain these parameters significantly increases with dimensionality of $\mathcal{P}_u$.

• Requirements involving discrete modes are challenging because they induce “flat” quantitative satisfaction functions that are challenging to optimizers and thus have limited value in guiding the falsifier. This is illustrated by the performance of the mining algorithm with template $\varphi_{\texttt{stay}}$ (instances 7-9). The satisfaction function is locally “flat” due to the fact that the predicate $\texttt{gear} = 2$ induces piecewise-constant integer quantitative satifaction (equal to -1 if `gear` is 1, 0 if `gear` is 2, etc). For small N_{init}, the algorithm stops after a few iterations because the optimizations around the N_{init} initial valuations are stuck in those locally flat regions. On the other hand, for higher values of N_{init}, the input parameter space is better covered by the initial sampling, hence a better valuation for τ is found.

B. Air-Fuel Ratio Control Model

Next, we consider the model of a fuel control system for a gasoline engine presented in [?]. The model consists of an air-fuel ratio (AFR) controller and a model of the engine dynamics specifying the mean behavior of the engine over the various combustion cycle phases. While the model presented in [?] allows four discrete modes of operation, we are interested in mining requirements in the nominal mode (called the `normal` mode of operation). The basic purpose of the control system is to regulate the AFR quantity to a reference value (known as the stoichiometric value). The experimental results shown in Table III use the following requirements.

The requirement $\varphi_{\texttt{abs\_over}}(\pi)$ specifies the absolute value of the deviation of AFR from the reference value. In other words, it specifies the maximum allowed overshoot or undershoot. The requirements $\varphi_{\texttt{overshoot}}(\pi)$ and $\varphi_{\texttt{undershoot}}(\pi)$ separately specify the maximum value for the overshoot and the minimum value for the undershoot respectively.

The requirement $\varphi_{\texttt{settling\_time}}(\tau, \pi)$ specifies the settling time for the AFR signal when the throttle angle input of the model is excited by a train of pulses. In our first experiment mining this requirement, we prioritize the settling region π, i.e., we wish to find the smallest region in which the AFR signal settles, at the cost of allowing a longer time (τ) for the transients. In the second experiment, we wish to find the smallest time at which the AFR signal settles, but at the cost of settling in a larger region. Both experiments are valuable, as the first experiment is an indicator of how tightly the control system can track the reference value, while the second experiment indicates the control system's response time.

	Abstract fuel control model		
Template	Parameter value	Time (sec)	#Iter.
$\varphi_{\texttt{abs\_over}}(\pi)$	$\pi = 9.76e-3$	3111	2
$\varphi_{\texttt{overshoot}}(\pi)$	$\pi = 8.78e-3$	3201	2
$\varphi_{\texttt{undershoot}}(\pi)$	$\pi = -9.76e-3$	3121	2
$\varphi_{\texttt{settling\_time}}(\pi, \tau)$	$\tau = 1.405$, $\pi = 0.005$	3502	7
$\varphi_{\texttt{settling\_time}}(\tau, \pi)$	$\tau = 1.244$, $\pi = 0.0075$	3117	5
$\varphi_{\texttt{rms}}(\pi)$	$\pi = 0.040$	3301	4

TABLE III: Results on mining requirements for the abstract fuel control model of [?]. The stopping criterion for the last falsification step was set to 1000 simulations.

C. Diesel Engine Model

Next, we consider two different versions of an industrial-scale, closed-loop Simulink model of an experimental airpath controller for a diesel engine. The original model has more than 4000 Simulink blocks such as data store memories, integrators, 2D-lookup tables, functional blocks with arbitrary Matlab functions, S-Function blocks, and blocks that induce switching behaviors such as level-crossing detectors and saturation blocks. The models takes two signals as input: the fuel injection rate and the engine speed. The output signal is the intake manifold pressure denoted by x. For proprietary reasons, we suppress the mined values of the parameters and the time-domain constants from our requirements. We replace the time-domain constants by symbols such as c_1 and c_2. As before, we use T to represent the simulation time-horizon.

We note that in this case study, we have available two sets of results. In previous work that appears in [?], we mined requirements on an older version of the closed-loop diesel airthpath control system. We first summarize the results obtained therein. We then present the results[3] of mining requirements on a new version of the model, which incorporates the feedback that we provided to the designers through our first set of mining experiments.

[3]In Fig. 5 and Fig. 6 respectively corresponding to the two experiments, we suppress the values along the plot-axes for proprietary reasons. We remark that the actual values are irrelevant and the intention is to show the shape of the design behaviors.

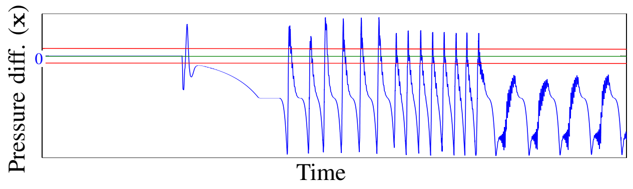


Fig. 5: The simulation trace (in blue) for the signal $\mathbf{x}$ denoting the difference between the intake manifold pressure and its reference value[4]found when mining $\varphi_{\texttt{settling\_time}}(\tau, \pi)$ displays unstable behavior. The maximum error threshold that we expected to mine is depicted in red. The ideal $\mathbf{x}$ signal is in green.

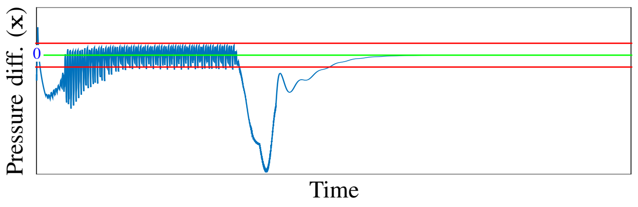


Fig. 6: The simulation trace (in blue) for the signal $\mathbf{x}$ denotes the worst case settling time for the difference between the intake manifold pressure and its reference value found by mining $\varphi_{\texttt{settling\_time}}(\tau, \pi)$ in the newer diesel engine model.

We found from the designers that characterizing the overshoot behavior is important for the intake manifold pressure signal. The inputs to the closed-loop model are a step function to the fuel injection rate input at time c_1, and a constant value for the engine speed input. The first requirement is:

$$\varphi_{\texttt{overshoot}}(\pi) = \Box_{(c_1,T)}(\mathrm{x} < \pi).$$

This template characterizes the requirement that the signal x never exceeds π during the time interval (c_1, T), i.e., it finds the maximum peak value (i.e., π) of the step response. Our mining algorithm obtained 7 intermediate candidate requirements that were falsified by S-TALIRO, till we found a requirement that it could not falsify in its 8^{th} iteration. The total number of simulations was 7000 over a period of 13 hours.

Next, we chose to mine the settling behavior of the signal. The *settling time* is the time after which the amplitude of signal is always within a small *error* from its calculated ideal reference value. We wish to mine both the error and how fast the signal settles. Such a template requirement is given by the following PSTL formula:

$$\varphi_{\texttt{settling\_time}}(\tau, \pi) = \Box_{[\tau,\infty)}(|\mathrm{x}| < \pi).$$

It specifies that the absolute value of x is always less than π starting from the time τ to the end of the simulation. The smaller the settling time and the error, the more stable is the system. We found out from the control designer that a smaller settling time needs to be prioritized over the error (as long as the error lies within the 10% of the signal amplitude), so we prioritize minimizing τ over minimizing π.

After 4 iterations, the procedure stopped as the inferred value for τ was very close to the end of the simulation trace, but the error was still larger than the tolerance. The implication here is that the algorithm pushed the falsifier to finding a behavior in the model that exhibits *hunting behavior*, or oscillations of magnitude exceeding the tolerance. This output signal is shown in Fig. 5. This behavior was unexpected; discussions with the designers revealed that it was a real bug. Investigating further, we traced the root-cause to an incorrect value in a lookup table; such lookup tables are commonly used to speed up the computation time by storing pre-computed values approximating the control law.

This experiment demonstrates the use of mining as an advanced, guided debugging strategy. Instead of verifying correctness with a concrete formal requirement, the process of trying to infer what requirement a model must satisfy can reveal erroneous behaviors that could be otherwise missed.

The counterexample we found helped the designers to rectify the erroneous behavior. Incidentally, the designers also chose to refactor the model by eliminating some blocks to reduce the computation time for the control code. This is reflected in the decreased simulation time, which in turn leads to a reduction in the time required for mining requirements. The resulting new version contains around 3000 blocks. The results are shown in Table IV. Here, we list the number of simulations, the total elapsed time (in hours), and the number of iterations of the mining algorithm. Here, through extensive simulation, the worst-case behavior we found on the new version of the model for $\varphi_{\texttt{settling\_time}}$ is shown in Fig. 6, with the absence of the previous hunting behavior. Through this example, we demonstrate that requirement mining process could be use to help designers detect corner cases in a design and ensure quality in design evolution.

Template	#Sim.	Time (hour)	#Iter.
$\varphi_{\texttt{overshoot}}(\pi)$	4733	4.12	5
$\varphi_{\texttt{settling\_time}}(\tau, \pi)$	100828	9.15	18

TABLE IV: Requirement mining results on the new diesel control model. The stopping criterion for the last falsification step was set to 2000 simulations.

VIII. Related Work

Mining requirements from programs and circuits is well-studied in the field of computer science [?], [?], [?], [?], [?], [?], [?], [?]. In computer science, the word "requirement" is often synonymous with "specification". These techniques vary based on what is mined, e.g., automata, temporal rules, and sequence diagrams. They also differ on the input to the mining tool; e.g., techniques based on static analysis or model checking operate on the source code, dynamic techniques mine from execution traces. Work on mining temporal rules [?], [?] involves learning an automaton to capture the temporal behavior and focusses on API usage in libraries, and specification automata encode legal interaction-patterns between library components. In contrast to most software programs with discrete-time semantics, the behavioral requirements that we mine are for systems with both continuous and discrete-time semantics. It may be worthwhile to see if automata-based mining could be adapted to the hybrid systems domain. The work closest to the proposed approach appears in [?], where the authors introduce Parametric MTL (PMTL), which is able to specify single time or scale parameters in MTL formulas. These parameters are estimated using stochastic optimization within the tool S-TALIRO. We remark that we provide a way to reason about monotonicity of PSTL formulas with arbitrary number of parameters, and also allow mining non-monotonic PSTL formulas (albeit less efficiently).

Asarin et al. [?] introduce Parametric Signal Temporal Logic and use it to infer properties of continuous-time signals. This technique only statically infers specifications, where given signals are queried, without mention (and, a fortiori, actuation) of a model that produces these signals. Kong et al. [?] also focus on inferring temporal logic patterns from data. They define a fragment of PSTL that allows a separation of cause and effect formulas. With this structural separation, the authors can impose a lattice structure on the space of the PSTL formulas, allowing for simultaneous parameter estimation and structural identification. Note that in our prior work [?], as well as in this paper, we do not address the problem of learning the structure of the PSTL formula. An important part of our future work will involve exploring the space of PSTL templates, using the algorithms developed in this paper.

We note that falsification of a given STL formula by a model behavior is a key component of our framework. The complement of the falsification problem is the verification problem. Reachability analysis tools such as SpaceEx [?], Flow* [?], C2E2 [?], HyCreate [?], and CORA [?], are verification tools based on overapproximating the set of reachable behaviors of a given dynamical system. These tools can check (in a sound fashion) whether a given model contains a finite time behavior that violates a specified safety property. Such safety properties are expressed using regions in the state-space that should not be reached. Verification algorithms do not typically produce counterexamples (which we need in our counterexample-guided mining algorithm), thus their use in a falsification setting is unlikely. If verification tools are able to support checking general temporal logic specifications, then it may be possible to use them as a final step to check if the mined requirement is satisfied by *all* model behaviors.

To the best of our knowledge, this work is among the first to address the specification mining problem for cyber-physical systems. From a broader perspective, the literature reports several attempts to apply formal methods to industrial-scale block-based design tools such as Simulink. There is prior work [?] on verifying simple safety properties using sensitivity analysis. Other approaches that are able to work with Simulink diagrams include approaches to transform Simulink diagrams into models amenable to formal verification [?], [?], [?] or approaches to perform guided symbolic simulation using user-provided block-level annotations [?], [?]. When successful, such approaches provide very strong guarantees. However, in the former class of approaches, the type of blocks that can be handled is usually limited and we are not aware of scalable analysis tools for models representing general hybrid systems. The approaches based on symbolic simulation could be interesting alternatives for falsification.

ACKNOWLEDGMENT

We thank the anonymous referees for their comments, and James Kapinski, Koichi Ueda, and Ken Butts for help with Simulink models, experiments, and insightful discussions. The second and fourth authors were funded in part by Toyota via the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, and by the MultiScale Systems Center (MuSyC), funded under the Focus Center Research Program, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

REFERENCES

[1] M. Althoff. Reachability Analysis of Nonlinear Systems using Conservative Polynomialization and Non-Convex Sets. In *Proc. of Hybrid Systems: Computation and Control*, pages 173–182, 2013.

[2] R. Alur, T. Feder, and T. A. Henzinger. The Benefits of Relaxing Punctuality. *J. ACM*, 43(1):116—-146, Jan. 1996.

[3] R. Alur and T. A. Henzinger. A Really Temporal Logic. *J. ACM*, 41(1):181–203, 1994.

[4] R. Alur, A. Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic Analysis for Improving Simulation Coverage of Simulink/Stateflow Models. In *Proc. of Int. Conf. on Embedded Software*, pages 89–98, 2008.

[5] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java Classes. *ACM SIGPLAN Notices*, 40(1):98–109, 2005.

[6] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *ACM Sigplan Notices*, 37(1):4–16, 2002.

[7] Y. S. R. Annapureddy and G. E. Fainekos. Ant Colonies for Temporal Logic Falsification of Hybrid Systems. In *Proc. of the 36th Annual Conf. of the IEEE Industrial Electronics Society*, pages 91–96, 2010.

[8] Y. S. R. Annapureddy, C. Liu, G. E. Fainekos, and S. Sankaranarayanan. S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257, 2011.

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

[10] S. Bak and M. Caccamo. Computing Reachability for Nonlinear Systems with HyCreate. In *Demo and Poster Session at Hybrid Systems: Computation and Control*, 2013.

[11] H. A. Bardh Hoxha and G. Fainekos. Benchmarks for Temporal Logic Requirements for Automotive Systems. In *Workshop on Applied Verification for Continuous and Hybrid Systems*, 2014.

[12] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[13] X. Chen, E. Abraham, and S. Sankaranarayanan. Flow*: An Analyzer for Non-Linear Hybrid Systems. In *Proc. of Computer Aided Verification*, pages 258–263, 2013.

[14] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, page 337–340, 2008.

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

[16] A. Donzé, T. Ferrère, and O. Maler. Efficient Robust Monitoring for STL. In *Proc. of Computer Aided Verification*, pages 264–279, 2013.

[17] A. Donzé, B. Krogh, and A. Rajhans. Parameter Synthesis for Hybrid Systems with an Application to Simulink Models. In *Proc. of Hybrid Systems: Computation and Control*, pages 165–179, 2009.

[18] A. Donzé and O. Maler. Robust Satisfaction of Temporal Logic over Real-Valued Signals. In *Proc. of Formal Modeling and Analysis of Timed Systems*, pages 92–106, 2010.

[19] P. S. Duggirala, S. Mitra, and M. Viswanathan. Verification of Annotated Models from Executions. In *Intl. Conf. on Embedded Software*, 2013.

[20] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[21] G. E. Fainekos, S. Sankaranarayanan, K. Ueda, and H. Yazarel. Verification of Automotive Control Applications using S-TaLiRo. In *Proc. of the American Control Conference*, 2012.

[22] G. Frehse, C. Le Guernic, A. Donzé, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable Verification of Hybrid Control Systems. In *Proc. of Computer-Aided Verification*, 2011.

[23] T. Henzinger, P. Kopke, A. Puri, and P. Varaiya. What's Decidable about Hybrid Automata? In *Proc. of the Symposium on Theory of Computing*, pages 373–382, 1995.

[24] P. Herber, R. Reicherdt, and P. Bittner. Bit-precise formal verification of discrete-time matlab/simulink models using smt solving. In *Proc. International Conference on Embedded Software*, 2013.

[25] P. Hunter, J. Ouaknine, and J. Worrell. Expressive completeness for metric temporal logic. In *Proc. of Logic in Computer Science*, pages 349–357, 2013.

[26] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Powertrain Control Verification Benchmark. In *Proc. of Hybrid Systems: Computation and Control*, pages 253–262, 2014.

[27] X. Jin, A. Donzé, and G. Ciardo. Mining Weighted Requirements from Closed-loop Control Models. In *Workshop on Numerical Software Verification (NSV)*, 2013.

[28] X. Jin, A. Donzé, J. V. Deshmukh, and S. A. Seshia. Mining Requirements from Closed-loop Control Models. In *Proc. of Hybrid Systems: Computation and Control*, 2013.

[29] A. Kanade, R. Alur, F. Ivančić, S. Ramesh, S. Sankaranarayanan, and K. C. Shashidhar. Generating and Analyzing Symbolic Traces of Simulink/Stateflow Models. In *Proc. of Computer Aided Verification*, pages 430–445, 2009.

[30] Z. Kong, E. A. G. Austin Jones, Ana Medina Ayala, and C. Belta. Temporal logic inference for classification and prediction from data. In *Proc. of Hybrid Systems: Computation and Control*, 2014.

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

[32] C. Lee, F. Chen, and G. Rosu. Mining Parametric Specifications. In *Proc. of Int. Conf. on Software Engineering*, page 591–600, 2011.

[33] W. Li, A. Forin, and S. A. Seshia. Scalable Specification Mining for Verification and Diagnosis. In *Proc. of Design Automation Conference*, page 755–760, 2010.

[34] O. Maler and D. Nickovic. Monitoring Temporal Properties of Continuous Signals. In *Proc. of Formal Modeling and Analysis of Timed Systems/ Formal Techniques in Real-Time and Fault Tolerant Systems*, pages 152–166, 2004.

[35] T. Nghiem, S. Sankaranarayanan, G. E. Fainekos, F. Ivancic, A. Gupta, and G. J. Pappas. Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In *Proc. of Hybrid Systems: Computation and Control*, pages 211–220, 2010.

[36] G. Nicolescu and P. J. Mosterman. *Model-Based Design for Embedded Systems*. CRC Press, 2009.

[37] A. Pnueli. The Temporal Logic of Programs. In *Proc. of Foundations of Computer Science*, pages 46–57, 1977.

[38] S. Sankaranarayanan and G. E. Fainekos. Falsification of Temporal Properties of Hybrid Systems using the Cross-Entropy Method. In *Proc. of Hybrid Systems: Computation and Control*, 2012.

[39] S. Sankaranarayanan, F. Ivancic, and A. Gupta. Mining Library Specifications using Inductive Logic Programming. In *Proc. of Int. Conf. on Software Engineering*, page 131–140, 2008.

[40] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia. Static Specification Mining using Automata-based Abstractions. *IEEE Trans. on Software Engineering*, 34(5):651–666, 2008.

[41] Simulink. *version 8.0 (R2012b)*. The MathWorks Inc., Natick, Massachusetts, 2012.

[42] S. Skogestad and I. Postlethwaite. *Multivariable feedback control: Analysis and Design*. Wiley, 2007.

[43] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial Sketching for Finite Programs. *ACM SIGPLAN Notices*, pages 404–415, 2006.

[44] A. Tiwari. HybridSAL Relational Abstracter. In *Proc. of Computer Aided Verification*, pages 725–731, 2012.

[45] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating Discrete-Time Simulink to Lustre. *ACM Trans. on Embedded Comput. Syst.*, 4(4):779–818, 2005.

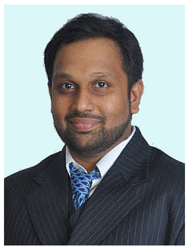
[46] W. Weimer and G. Necula. Mining Temporal Specifications for Error Detection. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, page 461–476, 2005.

[47] H. Yang, B. Hoxha, and G. Fainekos. Querying Parametric Temporal Logic Properties on Embedded Systems. In *Int. Conf. on Testing Software and Systems*, pages 136–151, 2012.

[48] C. Zhou and R. Kumar. Semantic Translation of Simulink Diagrams to Input/Output Extended Finite Automata. *Discrete Event Dynamic Systems*, 22(2):223–247, 2012.

Xiaoqing Jin Xiaoqing Jin is a research engineer with the Model-Based Development group at the Toyota Technical Center in Los Angeles. She received the B.Eng. and M.S. degrees in Computer Science from the Wuhan University, China in 2005 and 2007 respectively, and the Ph.D. degree in Computer Science from the University of California Riverside in 2013. Her work at Toyota focuses on advanced research on verification and validation techniques for automotive control systems modeled as nonlinear and hybrid dynamical systems. Her research interests include techniques for modeling, monitoring, analysis, and formal verification of large scale control systems.

Alexandre Donzé Alexandre Donzé is a research scientist at the University of California, Berkeley in the department of Electrical Engineering and Computer Science. He received his Ph.D. degree in Mathematics and Computer Science from the University of Joseph Fourier at Grenoble in 2007. He worked as a post-doctoral researcher at Carnegie Mellon University in 2008, and at Verimag in Grenoble from 2009 to 2012. His research interests are in simulation-based design and verification techniques using formal methods, Signal Temporal Logic (STL) with applications to cyber-physical systems and systems biology.



Jyotirmoy V. Deshmukh Jyotirmoy V. Deshmukh is a research engineer at Toyota Technical Center in Los Angeles. His research interests are in the broad area of formal verification of cyberphysical systems, automatic synthesis and repair of systems, and temporal logic. His current focus is in the area of automotive control systems, nonlinear and hybrid dynamical systems. He received the Ph.D. degree from the University of Texas at Austin in 2010, and worked a post-doctoral researcher at the University of Pennsylvania from 2010-2012.

Sanjit A. Seshia Sanjit A. Seshia received the B.Tech. degree in Computer Science and Engineering from the Indian Institute of Technology, Bombay in 1998, and the M.S. and Ph.D. degrees in Computer Science from Carnegie Mellon University in 2000 and 2005 respectively. He is currently an Associate Professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. His research interests are in dependable computing and computational logic, with a current focus on applying automated formal methods to embedded and cyber-physical systems, electronic design automation, computer security, and synthetic biology. He has served as an Associate Editor of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. His awards and honors include a Presidential Early Career Award for Scientists and Engineers (PECASE) from the White House, an Alfred P. Sloan Research Fellowship, the Prof. R. Narasimhan Lecture Award, and the School of Computer Science Distinguished Dissertation Award at Carnegie Mellon University.