Mining Requirements from Closed-Loop Control Models

Xiaoqing Jin
Univ. of California Riverside
jinx@cs.ucr.edu

Alexandre Donzé
Univ. of California Berkeley
donze@eecs.berkeley.edu

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

Sanjit A. Seshia
Univ. of California Berkeley
sseshia@eecs.berkeley.edu

ABSTRACT

A significant challenge to the formal validation of software-based industrial control systems is that system requirements are often imprecise, non-modular, evolving, or even simply unknown. We propose a framework for mining requirements from the closed-loop model of an industrial-scale control system, such as one specified in the Simulink modeling language. The input to our algorithm is a *requirement template* expressed in Parametric Signal Temporal Logic — a formalism to express temporal formulas in which concrete signal or time values are replaced by parameters. Our algorithm is an instance of counterexample-guided inductive synthesis: an intermediate candidate requirement is synthesized from *simulation traces* of the system, which is refined using counterexamples to the candidate obtained with the help of a *falsification* tool. The algorithm terminates when no counterexample is found. Mining has many usage scenarios: mined requirements can be used to validate future modifications of the model, they can be used to enhance understanding of legacy models, and can also guide the process of bug-finding through simulations. We present two case studies for requirement mining: a simple automobile transmission controller and an industrial airpath control model for an engine.

Categories and Subject Descriptors

D.4.7 [**Organization and Design**]: Real-time Systems and embedded systems; D.2.1 [**Software Engineering**]: Requirements/Specifications; I.2.8 [**Computing Methodologies**]: Problem Solving, Control Methods, and Search

General Terms

Verification, Theory

Keywords

Model-based design; Parametric Temporal Logics; Simulink

HSCC'13, April 8–11, 2013, Philadelphia, Pennsylvania, USA.

1. INTRODUCTION

Industrial-scale controllers used in automobiles and avionics are now commonly developed using a model-based design (MBD) paradigm [23, 28]. 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 logical, differential 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 specific control laws to regulate the behavior of the physical system. The *closed-loop model* consists of the composition of the plant and the controller.

The designer may then perform extensive *simulations* of such a model. The goal is to analyze the controller design by observing the temporal behavior of the signals of interest by exciting the exogenous, time-varying inputs to the model. An important aspect of this step is *validation*, i.e. checking if the temporal behavior of the system matches a set of *requirements*. Unfortunately, in practice, these requirements are high-level and often vague. Examples of requirements 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 design setting, requirements are rarely expressed formally, and it is common to find them written in natural language. Control designers then validate their design manually by comparing experimental time traces to these informal requirements. In some cases, they simply use simulation-data and their domain expertise to determine the quality of the design. Moreover, to date, formal validation tools have been unable to digest the format or scale of industrial-scale models. As a result, widespread adoption of formal tools has been restricted to testing syntactic coverage of the controller code, with the hope that higher coverage implies better chances of finding bugs. It is clear that even simulation-based tools would benefit from the more semantic notions of coverage offered by formal requirements.

In this paper, we propose a scalable technique to systematically mine requirements from the closed-loop model of a control system from observations of the system behavior. In addition to the model being analyzed, our technique takes as input a *template requirement*. The final output is

a synthesized requirement matching the template. In our current implementation, we assume that the model is specified in Simulink [21], 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. We remark that our technique is not restricted to Simulink models; in principle, it is applicable in any setting where the closed-loop system can be simulated, e.g., hardware-in-the-loop simulations, and tests on the physical system.

Formalisms such as Metric Temporal Logic (MTL) [2,17], and later Parametric Signal Temporal Logic (PSTL) [6] have emerged as logics well-suited to capture both the real-valued and time-varying behaviors of hybrid control systems. As PSTL is equipped with *parameters*, properties in PSTL naturally express template requirements. As an example, consider the following natural language specification: "eventually between time 0 and some unspecified time τ_1, the signal $\mathtt{x}$ 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:

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

Here, we interpret the unspecified values $\tau_1, \tau_2, \pi_1, \pi_2$ as parameters. The subset of PSTL with no parameters is referred to as STL. Robust satisfaction of MTL formulas [13] and quantitative semantics for PSTL [11] allow reasoning about how "close" a system behavior is to satisfying a given specification. Intuitively, a lower *satisfaction value* corresponds to a stronger property, making it easier for a behavior to violate the property.

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

1. In the first step, the algorithm synthesizes a *candidate requirement* from a given PSTL template and a set of simulation traces of the model. The candidate requirement is the strongest STL property satisfied by the given set of traces. It is obtained by instantiating the PSTL template with the parameter values that minimize the satisfaction value of the PSTL property over the given traces.
2. It then tries to falsify the candidate requirement using a global optimization-based search, such as using stochastic-search within the tool S-Taliro [5].
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.

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. For this purpose, we use the Breach tool [9]. If the number of parameters is n, a naïve search strategy in the parameter space would have an exponential cost in n.

However, we observe that the satisfaction values of certain PSTL properties are monotonic in their parameter values. For example, consider the property $\phi = \Diamond_{[0,\tau]}(x > \pi)$. Suppose that the minimum value of a given trace $x(t)$ is 3, then, starting from a value less than 3, as π increases, the property ϕ becomes a *stronger* assertion for the trace $x(t)$, i.e., its satisfaction value decreases. Finally, when π exceeds 3, the satisfaction value becomes negative, i.e., ϕ no longer holds for $x(t)$. Thus, we can say that the satisfaction value of ϕ monotonically decreases in the parameter π. Similarly, the satisfaction value of ϕ monotonically increases in τ. 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 [6] 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 [7].

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. 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. The second application explores the use of mining as an enhanced bug-finding procedure. 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 is unlikely 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 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. 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 formulas 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, real arithmetic and uninterpreted functions, and use an SMT solver to answer the query.
3. We demonstrate the practical applicability of our technique in two case studies: (a) a simple automatic transmission controller, and (b) 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.

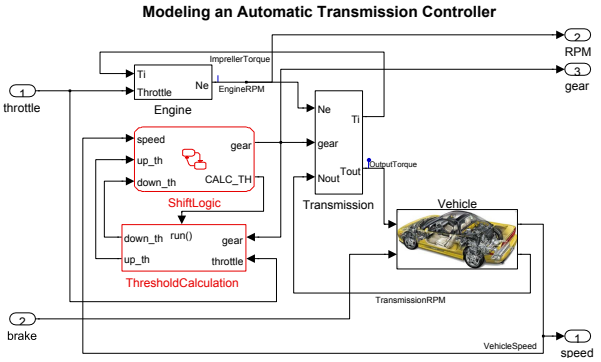

Figure 1: The closed-loop Simulink model of an automatic transmission controller. The input to the model is the throttle position and the brake torque.

The rest of the paper is as follows: In Sec. 2, we present a transmission controller as a running example. Sec. 3 presents the background, the problem formulation and an overview of our technique. We discuss our approach for finding counterexamples to candidate requirements in Sec. 4, and synthesizing parameter values for templates in Sec. 5. We present two case studies and experimental results for each in Sec. 6, and conclude with related work in Sec. 7.

2. 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 transmission ratio (`Ti`) using the current gear status, and computes the output torque from the engine speed (`Ne`), the gear status and the transmission RPM. The other two blocks represent the gear shift logic and the related threshold speed calculation. The model has two inputs: (1) the percentage of the `throttle` position, and (2) the `brake` torque.

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 we can show that these requirements are not met, as illustrated in Fig. 2.

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. In the next section, we present a technique to automatically obtain such requirements from the model.

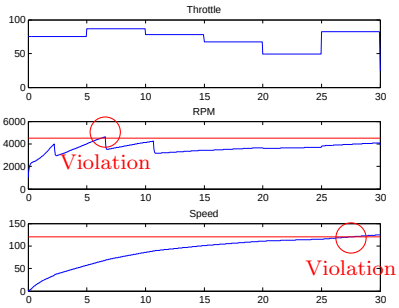

Figure 2: Falsifying trace for the automatic transmission controller and the requirement that RPM never goes beyond 4500 or speed beyond 120 mph.

3. PRELIMINARIES AND OVERVIEW

3.1 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). Additionally, the systems are closed-loop, meaning that they are obtained by composing a controller and a plant in a loop.[1]

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 $\bot$) 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 $\mathbf{x}(t) = \mathcal{S}(\mathbf{u}(t))$. It is common to drop time t, and say $\mathbf{x} = \mathcal{S}(\mathbf{u})$. 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.

3.2 Signal Temporal Logic

Temporal logics were introduced in the late 1970s [24] 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 [2], and Metric Temporal Logic (MTL) [17] were introduced later to deal with dense-time signals. More recently, Signal Temporal Logic [20] 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 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 con-

[1]Note that such systems can have exogenous inputs, e.g. a human controlling brakes provides inputs to the vehicle engine and controller system. The term "closed-loop" differs from "closed systems," which are systems with no inputs.

stant. 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:

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

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

Parametric Signal Temporal Logic (PSTL) [6] is an extension of STL introduced 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 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 3.1. *For the example from Sec. 2, 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:*

$$\psi = \Box(\texttt{speed} \leq 120) \wedge \Box(\texttt{RPM} \leq 4500). \tag{3.1}$$

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

$$\varphi(\pi_{speed}, \pi_{rpm}) = \Box(\texttt{speed} \leq \pi_{speed}) \wedge \Box(\texttt{RPM} \leq \pi_{rpm}). \tag{3.2}$$

The STL formula ψ expressed in (3.1) *is then obtained by using the valuation $v = (\pi_{\texttt{speed}} \mapsto 120, \pi_{rpm} \mapsto 4500)$. formulation.*

Problem 3.1. *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_i could either be scale parameter π or time parameter τ, the objective is to find a* tight *valuation function v such that*

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

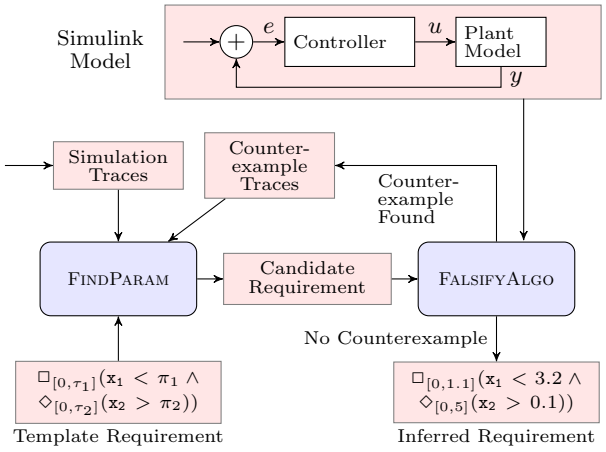

Figure 3: Flowchart for Requirement Mining

Our focus on "*tight* valuations" is to avoid mining trivial requirements or requirements that are overly conservative, e.g. "the car cannot go faster than the speed of light." We make this notion more precise in Section 5.1.

3.3 Requirement Mining Algorithm: Overview

Our algorithm for mining STL requirements from the closed-loop model in Simulink is an instance of a counterexample-guided inductive synthesis procedure [29], shown in Fig. 3. It consists of two key components:

1. A falsification engine, which, given a formula φ generates an input u such that $\mathbf{x}(t) = \mathcal{S}(u)(t) \not\models \varphi$ if there exists such a u, and returns $\perp$ otherwise. We denote this functionality by FalsifyAlgo.
2. A synthesis function denoted FindParam that given a set of traces $\mathbf{x}_1, \ldots, \mathbf{x}_k$, finds parameters $\mathbf{p}$ such that $\forall i$, $\mathbf{x}_i \models \varphi(\mathbf{p})$. We denote this function by FindParam.

4. FALSIFICATION PROBLEM

Recall that we need to implement a function

$$\mathbf{x} = \text{FalsifyAlgo}(\mathcal{S}, \varphi)$$

such that $\mathbf{x}$ 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; letting φ be a simple safety property establishes a reduction from the reachability problem for general hybrid systems, which is undecidable except for subclasses such as initialized rectangular hybrid automata [16]. For the latter subclasses the mining technique can be *complete*, i.e., absence of a counterexample means that we have found the strongest requirement. However, in general 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 something 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 that employs abstraction [15, 30]; however, in our experience, these tools have

not scaled 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 [5] and propose a falsification algorithm based on the minimization of the quantitative satisfaction of a temporal logic formula.

4.1 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:

$$\rho(\varphi, \mathbf{x}, t) \geq 0 \text{ iff } (\mathbf{x}, t) \models \varphi. \tag{4.1}$$

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 [11], different quantitative semantics for STL have been proposed. We recall the most commonly used semantics defined inductively from the quantitative semantics for predicates and inductive rules for each STL operator.

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 trivially transformed into this form). From this form, a straightforward quantitative semantics for predicate μ is defined as

$$\rho(\mu, \mathbf{x}, t) = f(\mathbf{x}(t)). \tag{4.2}$$

Then ρ is defined inductively for every STL formula using the following rules:

$$\rho(\neg\varphi, \mathbf{x}, t) = -\rho(\varphi, \mathbf{x}, t) \tag{4.3}$$

$$\rho(\varphi_1 \wedge \varphi_2, \mathbf{x}, t) = \min(\rho(\varphi_1, \mathbf{x}, t), \rho(\varphi_2, \mathbf{x}, t)) \tag{4.4}$$

$$\rho(\varphi_1 \mathbf{U}_I \varphi_2, \mathbf{x}, t) = \sup_{t' \in t \oplus I} \big(\min(\rho(\varphi_2, \mathbf{x}, t'), \inf_{t'' \in [t, t')} \rho(\varphi_1, \mathbf{x}, t'')\big) \tag{4.5}$$

Then it can be shown [11] that ρ satisfies (4.1) and thus defines a quantitative semantics for STL. Additionally, by combining (4.5), and $\Box_I \varphi \triangleq \neg\Diamond_I \neg\varphi$, we get

$$\rho(\Box_I \varphi, \mathbf{x}, t) = \inf_{t' \in t \oplus I} \rho(\varphi, \mathbf{x}, t') \tag{4.6}$$

For $\Diamond$, we get a similar expression using sup instead of inf.

Example 4.1. *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* (4.2), *we get*

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

Applying rule (4.6) *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 (4.4):

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

Informally, the 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.

4.2 STL vs. MTL Robust Satisfaction

In this section, we clarify the connection between the quantitative semantics of STL defined above and the notion of robust satisfaction of MTL as defined in [13] and used in S-Taliro. The main difference between STL and MTL lies in the definition of predicates, and so does the difference between quantitative semantics. The robust semantics of MTL is based on the definition of a *metric* on the state space of signals and the fact that each predicate is identified with a set where it holds true. Formally, let d be a metric on $\mathbb{R}^n$ with the usual extension to the *signed distance* from a point $\mathbf{x} \in \mathcal{X}$ to a set $\mathcal{X}' \subseteq \mathcal{X}$:

$$d(\mathbf{x}, \mathcal{X}') = \begin{cases} -\inf_{\mathbf{x}' \in \mathcal{X}} d(\mathbf{x}, \mathbf{x}') \text{ if } \mathbf{x} \notin \mathcal{X}' \\ \inf_{\mathbf{x}' \in \mathcal{X} \setminus \mathcal{X}'} d(\mathbf{x}, \mathbf{x}') \text{ otherwise} \end{cases}$$

For each MTL predicate μ, define its truth set $\mathcal{O}(\mu)$ as:

$$\mathbf{x}, t \models \mu \text{ iff } \mathbf{x}(t) \in \mathcal{O}(\mu) \tag{4.7}$$

and let

$$\rho_d(\mu, \mathbf{x}, t) = d(\mathbf{x}(t), \mathcal{O}(\mu)). \tag{4.8}$$

Finally, the robust satisfaction ρ_d is defined using the same inductive rules (4.3-4.5) as for an STL formula φ. From there, it is clear that the quantitative semantics of STL subsumes the robust semantics of MTL. Indeed, as each predicate in STL is associated with an arbitrary function f, this function can implement the distance d. Then, if ρ_d and ρ coincide on a set of predicates, by induction they coincide on all formulas defined on those predicates.

4.3 Solving the Falsification Problem

The objective of the falsification problem is: 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 $\mathbf{x}$ such that $\rho(\varphi, \mathbf{x}, 0) < 0$. Hence FalsifyAlgo can be implemented by solving

$$\text{Solve } \rho^* = \min_{\mathbf{u} \in \mathcal{U}} \rho(\varphi, \mathcal{S}(\mathbf{u}), 0) \tag{4.9}$$

Then if $\rho^* < 0$, we return $\mathbf{u}^* = \arg\min_{\mathbf{u} \in \mathcal{U}} \rho(\varphi, \mathcal{S}(\mathbf{u}), 0)$, otherwise, $\mathcal{S} \models \varphi$. The undecidability of the falsification problem is reflected here in the fact that the minimization problem (4.9) 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, such as Monte-Carlo [22], and the cross-entropy method [25]. In our implementation, we first instrumented S-Taliro as a falsification tool (made possible by the connection between STL and MTL described in the previous section) and then extended Breach with a new falsification engine which attacks (4.9) 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 $v(\mathbf{k}) \in \mathcal{P}_u$.

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

Algorithm 1: FINDPARAM algorithm.

2. Sample signal-parameters in a uniform, random fashion 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.

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. One motivation for implementing a falsification module in Breach has been to get more flexibility in the definition of input parameters than available in the version of S-TALIRO that we used. In the experimental section, we discuss some results using both S-TALIRO-based falsification and the above algorithm. We found in particular that the choice of input parameters, of N_{init} and the tuning of Nelder-Mead algorithm (which provides a trade-off between global randomized exploration and local optimization) were crucial for the performance of the falsifier.

5. PARAMETER SYNTHESIS

5.1 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 $\varphi(v(p_1), \ldots, v(p_n))$ (abbreviated as $\varphi(v)$ in the following). This problem is a dual of the falsification problem (4.9) formulated as:

$$\max_v \rho(\varphi(v), \mathbf{x}, 0). \tag{5.1}$$

However, there is an important difference that the cost function can be expressed as a closed-form expression of the decision variable v whereas for (4.9) as a function of $\mathbf{u}$. By taking advantage of this knowledge, (5.1) can be solved more efficiently, in particular as we will see, if formulas satisfy the important property of *monotonicity*:

DEFINITION 5.1. *A PSTL formula $\varphi(p_1, \cdots, p_n)$ is monotonically increasing with respect to p_i if for every signal $\mathbf{x}$,*

$$\forall v, v' : \left(\begin{array}{l} \mathbf{x} \models \varphi(\ldots, v(p_i), \ldots) \\ \wedge\ v'(p_i) \geq v(p_i) \end{array} \right) \Rightarrow \mathbf{x} \models \varphi(\ldots, v'(p_i), \ldots)$$

[2]We restrict our attention to one trace though in the mining process, FINDPARAM has to work on a set of traces. The generalization to multiple traces is straightforward.

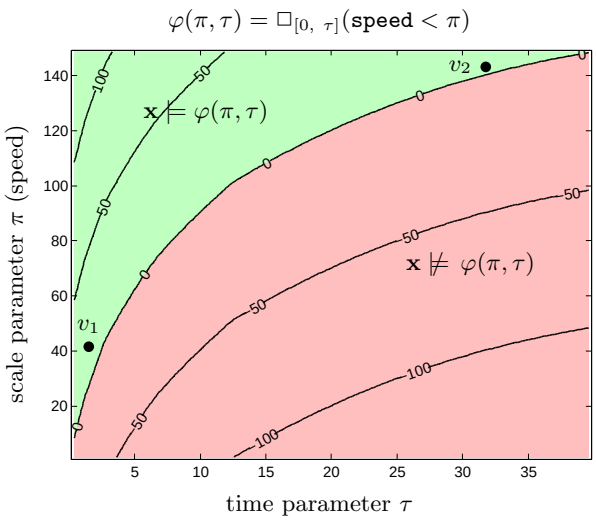

Figure 4: Validity domain of a simple formula for a trace x obtained from the automatic transmission model. The FindParam algorithm will return valuation v_1 (resp. v_2) depending if time (resp. scale) parameter is optimized first. The contour lines are isolines for the satisfaction function ρ.

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

In the second part of this section, we characterize this notion more precisely. We impose an additional constraint to the parameter synthesis problem: we require that the STL formula mined be "tightly" satisfied by the system up to a given precision $\delta > 0$. Formally,

DEFINITION 5.2. *The signal $\mathbf{x}$ δ-satisfies $\varphi(v)$ for p_i denoted by $\mathbf{x} \models^i_\delta \varphi(v)$ iff $\mathbf{x} \models \varphi(v)$ and there exists a valuation v' such that $\forall j \neq i$, $v'(p_j) = v(p_j)$, $|v(p_i) - v'(p_i)| \leq \delta$ and $\mathbf{x} \not\models \varphi(v')$. The signal $\mathbf{x}$ δ-satisfies $\varphi(v)$, denoted by $\mathbf{x} \models_\delta \varphi(v)$ if $\forall i$, $\mathbf{x} \models^i_\delta \varphi(v)$.*

The rationale is that for a specification to be useful it should not be too conservative. The implication is that it is not enough to find a satisfying valuation, we also need to optimize it for each parameter to get δ-satisfaction. If there is more than one parameter, then the solution is not unique. In fact, all valuations that are at a distance δ from the boundary of the *validity domain* of φ and $\mathbf{x}$ (the set of valuations v for which $\mathbf{x} \models \varphi(v)$) are valid solutions. In [6], the authors note that if the formula is monotonic, then this boundary 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 (Algorithm 1) for monotonic formulas that takes advantage of this property, and implement it in the BREACH tool. Algorithm 1 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 parameters to optimize first by specifying a priority ordering for the input parameters. Note that different orderings can give drastically different results.

Formula	Monot.	Time
$\Box_{(0,\infty)}(x < \pi)$	$+$	<0.09 s
$\Box_{[s,s+1]}(x \geq 3 \Rightarrow \Diamond_{(0,\infty)} x < 3)$	$-$	0.1 s
$\Box_{(0,100)}((x < \pi) \Rightarrow \Diamond_{(0,5)}(x > \pi))$	$-$	<0.09 s
$\texttt{gear}_i \mathbf{U}_{(s,s+5)} \texttt{gear}_{i+1}$	*	0.13 s

Table 1: Proving monotonicity with an SMT solver.

EXAMPLE 5.1. *Consider* $\varphi(\pi, \tau) = \Box_{[0,\ \tau]}(\texttt{speed} < \pi)$ *and the scenario that the vehicle constantly accelerates with the value of* `throttle` *set at* 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.*

5.2 Satisfaction monotonicity

We first show that checking if an arbitrary PSTL formula is monotonic in a given parameter is undecidable.

THEOREM 5.1. *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 [1] 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 $\forall v', \mathbf{x} : \ \mathbf{x} \models \psi(v'(\tau))$, i.e., checking the validity of the PSTL formula $\psi(\tau)$.

Thus, 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 [6], 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 [6]. 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 end-points are parameterized, such as the following:

$$\Box_{[\tau,\tau+1]}((x \geq 3) \Rightarrow \Diamond_{(0,\infty)}(x < 3)) \tag{5.2}$$

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 (5.2) 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 $\mu \doteq g(\mathbf{x}) > \tau$, 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., $\Diamond, \Box$). 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 $\Box$ 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, $\varphi = \Box_{(0,\infty)}\Diamond_{(1,2)}(x > 10)$. 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 two assertions below:

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

Template	S-Taliro-based falsification					Breach-based falsification				
	Parameter values	Fals.	Synth.	#Sim.	Sat./$\mathbf{x}$	Parameter values	Fals.	Synth.	#Sim.	Sat./$\mathbf{x}$
$\varphi_{\texttt{sp\_rpm}}(\pi_1, \pi_2)$	(155 mph, 4858 rpm)	55 s	12 s	255	0.004 s	(155 mph, 4858 rpm)	197.2 s	23.1 s	496	0.043 s
$\varphi_{\texttt{rpm100}}(\pi, \tau)$	(3278.3 rpm, 49.91 s)	6422 s	26.5 s	9519	0.327 s	(3273 rpm, 49.92 s)	267.7 s	10.51 s	709	0.026 s
$\varphi_{\texttt{rpm100}}(\tau, \pi)$	(4997 rpm, 12.20 s)	8554 s	53.8 s	18284	0.149 s	(4997 rpm, 12.20 s)	147.8 s	5.188 s	411	0.021 s
$\varphi_{\texttt{stay}}(\pi)$	1.79 s	18886 s	0.868 s	130	147.2 s	0.102 s	430.9 s	2.157 s	1015	0.032 s

Table 2: Results on mining for the automatic transmission control model. We compare runs of the mining algorithm using either S-Taliro or Breach as falsifiers. In each case and for each template formula, we give the parameters valuations found, the time spent in falsification and in parameter synthesis, the number of simulations and the averaged time spent computing the quantitative satisfaction of the formula by one trace.

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 [8] in Table 1. The symbols $+$, $-$, and * represent monotonically increasing, decreasing and non-monotonic formulas respectively.

6. CASE STUDIES

In what follows, we present our evaluation of both S-Taliro and an extension of Breach for falsification. We also show the performance of the parameter synthesis algorithm implemented with the robust satisfaction engine of Breach. We use the transmission controller model to benchmark the different options within our approach.

6.1 Automatic Transmission Model

For the model described in Sec. 2, 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 :

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

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:

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

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:

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

Here, the left-hand-side of the implication captures the *event* of the transition from gear 2 to 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 2. We used the Z3 SMT solver [8] to show that all of the requirements were monotonic. As expected, the FindParam algorithm takes only a fraction of the total time in the entire mining process. 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 12.3 s. 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 0.112s. 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.

The comparison between S-Taliro and Breach falsifiers shows better overall performance with the extended Breach-based falsifier, in the sense that it found stronger requirements using less number of simulations and computational time. However, we cannot conclude that the new falsifier will always outperform S-Taliro, due to the stochastic nature of the problem and a lack of thorough comparison with the different flavors of optimization used within S-Taliro. Based on results shown in Table 2 and our experience, we make some observations:

- 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. For instance, the short transient shifting of $\varphi_{\texttt{stay}}$ was found by introducing a signal-parameter controlling the time of initial acceleration, and by preventing acceleration and braking at the same time. We remark that extending Breach to enforce such constraints over the input signal space is a key reason for its better performance, and a fair comparison would be possible only after repeating these steps for S-Taliro.
- 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 related to the problem of finding a good metric between discrete states in hybrid systems. This was particularly an issue when mining

the $\varphi_{\texttt{stay}}$ requirement. We were able to tune our falsifier by turning off its local optimization phase, and using uniform random sampling, which led us to obtaining a tighter requirement than with S-TALIRO.

• We found that while both falsifiers are expected to exhibit run-times linear in the size of the traces and the formula [11,14], in some cases, BREACH runs faster. In particular, S-TALIRO is more sensitive to parameter priorities. For the same template $\varphi_{\texttt{rpm100}}$, depending on which parameter τ or π is prioritized, S-TALIRO performs differently. This can be explained by the fact that τ affects the horizon of the temporal operator $\Diamond$. We conjecture that the difference in run-times and mined parameter values for the $\varphi_{\texttt{stay}}$ template is due to our inability to express signal-parameterization in S-TALIRO, but these comparisons require more dedicated studies on various benchmarks before drawing firm conclusions.

6.2 Diesel Engine Model

Next, we consider an industrial-scale, closed-loop Simulink model of an experimental airpath controller for a diesel engine. It 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 behavior such as level-crossing detectors, multiports, 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.

Discussions with control designers revealed that characterizing overshoot behavior of the intake manifold pressure is important. The inputs to the closed-loop model are a step function to the fuel injection rate at time c_1, and a constant value for the engine speed. The first requirement is:

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

This template characterizes the requirement that the signal x never exceeds π during the time interval (c_1, ∞), 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:

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

It specifies that the absolute value of $\mathbf{x}$ 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 stops as the inferred value for τ is very close to the end of the simulation trace, but the error is 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.

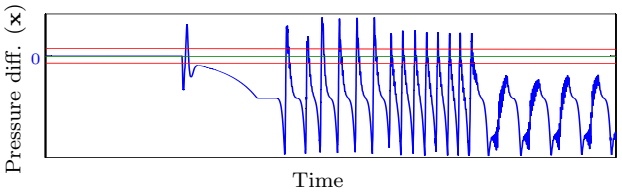

Figure 5: The simulation trace x (in blue) denoting the difference between the intake manifold pressure and its reference value[4] found when mining $\varphi_{\texttt{settling\_time}}(\tau, \pi)$ displays unstable behavior. The maximum error threshold is depicted in red.

This experiment demonstrates the use of requirement 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. In the course of our experiments, we encountered other suspicious (for instance Zeno-like) behaviors, which we suspect to be either an error in the model, or an improper tuning of the numerical solver leading to discontinuities in the dynamics.

7. RELATED WORK

Mining requirements from programs and circuits is well-studied in the field of computer science [3, 4, 12, 18, 19, 26, 27, 32]. In computer science, the word "requirement" is often synonymous with "specification". Specification mining techniques vary based on the kind of specifications mined; examples include automata, temporal rules, and sequence diagrams. They also vary based on the input to the miner; techniques based on static analysis or model checking operate on the source code, while dynamic techniques mine from execution traces. Mining temporal rules [3,27] involves learning an automaton that captures the temporal behavior and typically focusses on API usage in libraries. The individual components within such libraries are often *terminating* programs, and specification automata encode legal interaction-patterns between components. In contrast to the software world, where most programs have 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 [33], in which the authors introduce Parametric MTL (PMTL), which adds a *single* time or scale parameter to MTL formulas. This parameter is then estimated using stochastic optimization

[4]Note that the values along the axes have been suppressed for proprietary reasons. We remark that the actual values are irrelevant and the intention is to show an oscillating behavior arising from a real bug in the design.

within 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).

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. In [10], the authors verify simple safety properties using sensitivity analysis. Other approaches try to transform Simulink diagrams into models amenable to formal verification [31, 34]. When successful, such approaches provide very strong guarantees. However the type of blocks that can be handled is usually limited and we are not aware of scalable analysis tools for models representing hybrid systems.

Acknowledgment

We thank the anonymous referees for their comments, and James Kapinski, Koichi Ueda, and Ken Butts for providing Simulink models, help with experiments, and insightful discussions. The first author was supported in part by National Science Foundation grant CCF-1018057. The second and fourth authors were funded in part by Toyota Motor Corporation via the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, and by the Multi-Scale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

8. REFERENCES

[1] R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116—-146, Jan. 1996.

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

[3] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. *ACM SIGPLAN Notices*, 40:98–109, 2005.

[4] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *ACM SIGPLAN Notices*, 37:4–16, 2002.

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

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

[7] 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.

[8] L. De Moura and N. Bjørner. Z3: An efficient SMT,solver. page 337–340, 2008.

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

[10] A. Donzé, B. Krogh, and A. Rajhans. Parameter synthesis for hybrid systems with an application to Simulink models. In *Hybrid Systems: Computation and Control*, pages 165–179, 2009.

[11] A. Donzé and O. Maler. Robust satisfaction of temporal logic over real-valued signals. In *Formal Modeling and Analysis of Timed Systems*, pages 92–106, 2010.

[12] 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.

[13] G. E. Fainekos and G. J. Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42):4262–4291, 2009.

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

[15] 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 *Computer-Aided Verification*, 2011.

[16] T. Henzinger, P. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Symposium on Theory of Computing*, pages 373–382, 1995.

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

[18] C. Lee, F. Chen, and G. Rosu. Mining parametric specifications. In *International Conference on Software Engineering*, page 591–600, 2011.

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

[20] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Formal Modeling and Analysis of Timed Systems*, pages 152–166, 2004.

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

[22] 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 *Hybrid Systems: Computation and Control*, pages 211–220, 2010.

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

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

[25] S. Sankaranarayanan and G. E. Fainekos. Falsification of temporal properties of hybrid systems using the cross-entropy method. In *Hybrid Systems: Computation and Control*, pages 125–134, 2012.

[26] S. Sankaranarayanan, F. Ivancic, and A. Gupta. Mining library specifications using inductive logic programming. In *International Conference on Software Engineering*, page 131–140, 2008.

[27] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. *IEEE Trans. on Software Eng.*, 34(5):651–666, 2008.

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

[29] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *Architectural Support for Programming Languages and Operating Systems*, pages 404–415, 2006.

[30] A. Tiwari. HybridSal relational abstracter. In *Computer-Aided Verification*, pages 725–731, 2012.

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

[32] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, page 461–476, 2005.

[33] H. Yang, B. Hoxha, and G. Fainekos. Querying parametric temporal logic properties on embedded systems. In *Int. Conference on Testing Software and Systems*, pages 136–151, 2012.

[34] C. Zhou and R. Kumar. Semantic translation of Simulink diagrams to input/output extended finite automata. *Discrete Event Dynamic Systems*, 22:223–247, 2012.