2015 American Control Conference
Palmer House Hilton
July 1-3, 2015. Chicago, IL, USA

Simulation-Guided Approaches for Verification of Automotive Powertrain Control Systems

James Kapinski, Jyotirmoy Deshmukh, Xiaoqing Jin, Hisahiro Ito, and Ken Butts*

Abstract— **Automotive embedded control systems are a vital aspect of modern automotive development, but the considerable complexity of these systems has made quality checking a challenging endeavor. Simulation-based checking approaches are attractive, as they often scale well with the complexity of the system design. This paper presents an overview of simulation-guided techniques that can be used to increase the confidence in the quality of an automotive powertrain control system design. We discuss the relationship between simulation-based approaches and the broader areas of verification and powertrain control design. Also, we discuss new software tools that use simulation-guided approaches to address various aspects of automotive powertrain control design verification. We conclude by considering ongoing challenges in developing new simulation-guided tools and applying them in a powertrain control development context.**

I. INTRODUCTION

Verification and testing activities for industrial cyber-physical systems, such as automotive engine control systems, account for a significant portion of the development effort. Increasing the efficiency of the debug and certification processes will have a significant impact on the total development cost. We describe several practical methods for using simulation traces to perform verification and other quality checking activities for automotive powertrain control (PTC) systems.

Many techniques are used to debug and check the quality of control code for automotive systems. For example, static code analyzers have become standard in most integrated development environments (IDEs) [22], but these analyzers can only check for specific types of coding errors, such as variable type mismatches, and cannot check properties that depend on the behavior of the environment.

Formal methods (FMs) are used in some hardware and software development domains to verify properties of computer code, such as C programs, or finite state logical behaviors, such as field programmable gate arrays (FPGAs). Model checking is an FM technique that takes a model of the system (e.g., computer code) and a property that should hold for the system, in the form of a temporal logic formula, and returns either a certificate of correctness or a counterexample that demonstrates a specific behavior that violates the requirement [16]. Several model checking tools have been developed and successfully applied to verify communication protocols [42], hardware drivers [29], and even focused components of automotive control code [34]. Model checkers such as SLAM [8] and CBMC [15] are used by industry to verify system correctness.

*Authors are with Toyota Technical Center, 1630 W. 186th St., Gardena, CA 90248, USA {firstname.lastname@tema.toyota.com}

Theorem proof assist tools, which we call *theorem provers*, are another FM technique for verifying system correctness. While model checkers rely on a semantic evaluation of the system behaviors, a theorem prover is an interactive framework that assists the user to construct a formal proof using deductive techniques. Tools such as PVS [41] and Coq [10] have been used to verify software correctness, but these tools require user intervention and can be difficult for non-experts to use.

FM techniques provide a high degree of confidence in the system correctness, but suffer from fundamental and practical drawbacks. Fundamentally, FMs do not scale well for large, industrial systems, such as automotive control code. On the practical side, formal methods are currently difficult for control engineers to use. Engineers are often unfamiliar with the temporal logics that are used to specify the requirements used by FMs. Also, many tools require that an intermediate model be created, based on the original system model. This process is often performed manually, and so is time consuming and prone to error.

Simulation, on the other hand, is commonly used in automotive control development. Simulations provide numerical approximations of system behavior, given a mathematical model of the system. For automotive control system designs, simulations are used to a.) validate functional behavior, b.) obtain initial control parameters, and c.) obtain estimates of system performance.

Currently, simulations are often used in an ad hoc manner to perform quality checking. Engineering intuition is used to select operating conditions to demonstrate the desired behavior. In the sequel, we describe several systematic techniques that use information from simulations to perform verification and other quality checking activities.

Preliminaries. We present the framework we will use to describe simulation-guided testing approaches. General testing and verification scenarios involve a system $\mathcal{M}$, a set of model parameters P, a set of inputs U, and a property that should hold for the system φ. Here $\mathcal{M}$ could be some model of the system or it could be a physical manifestation of the system. Testing and verification activities are defined in terms of behaviors $\phi(\mathcal{M}, p, u)$, where $p \in P$ and $u \in U$. We say that $\phi(\mathcal{M}, p, u)$ is the behavior of system $\mathcal{M}$ under parameter p and input u. We use $\phi(\mathcal{M}, P, U)$ to represent the set of all possible behaviors of $\mathcal{M}$ under the parameters in P and inputs in U. In the context of *testing*, behaviors can be observed from experiments, while in the context of *simulation*, behaviors are estimated using numerical methods.

978-1-4799-8684-2/$31.00 ©2015 AACC

We assume that $\mathcal{M}$ can be evaluated to determine whether φ holds for a particular p and u. We denote that φ holds for $\phi(\mathcal{M}, p, u)$ by $\phi(\mathcal{M}, p, u) \models \varphi$. When the sets P and U are clear from the context, we use $\mathcal{M} \models \varphi$ to mean that all behaviors in $\phi(\mathcal{M}, P, U)$ satisfy φ.

The following definitions provide the testing and verification activities that we address herein.

Definition 1.1 (Testing): The testing task is to determine whether $\phi(\mathcal{M}, \hat{P}, \hat{U}) \models \varphi$ for given finite sets $\hat{P} \subseteq P$ and $\hat{U} \subseteq U$.

Testing is the most common means of evaluation of automotive control system designs. We describe various testing approaches used in the automotive industry in Sec. II.

Note that sets $\hat{P}$ and $\hat{U}$ are finite in Definition 1.1, which implies that testing cannot be used to exhaustively evaluate continuous parameter ranges. This is a significant and fundamental limitation with testing as a method of performance evaluation; it typically means that testing cannot be used to validate behaviors of $\mathcal{M}$ for all possible cases. Verification, on the other hand, addresses this problem.

Definition 1.2 (Verification): The verification task is to prove $\phi(\mathcal{M}, P, U) \models \varphi$ for a given P and U.

Verification provides a formal proof of correctness of the system for a (possibly infinite) set of parameters and inputs. Technologies such as model checking and theorem proving can be used to performing verification for software systems. Some of these tools can be applied to automotive systems, but no mature tools exist that can be applied to detailed automotive models that capture environment behaviors, such as engine dynamics. We describe some new simulation-guided approaches for performing verification for automotive systems in Sec. IV.

Definition 1.3 (Falsification): The falsification problem is to find a $p \in P$ and $u \in U$ such that $\phi(\mathcal{M}, p, u) \not\models \varphi$.

The difference between testing and falsification is subtle. Testing determines whether a property holds for a given (finite) set of parameters and inputs, whereas falsification is an activity that searches for parameters and inputs from (possibly infinite) sets that demonstrate that $\phi(\mathcal{M}, p, u) \not\models \varphi$. Falsification technologies exist for finding bugs in automotive engine designs, some of which are described in Sec. III.

Definition 1.4 (Requirement Engineering): Requirement engineering is the process of developing an appropriate φ.

Requirement engineering remains a challenge for the automotive engineering. Many companies in the automotive industry have made significant effort to generate and document clear and concise requirements; however, challenges remain due to a.) the incompatibility of formats between documented requirements and the input to the software tools and b.) the large number of requirements. We discuss some tools and technologies for creating machine-checkable requirements in Sec III.

II. QUALITY CHECKING FOR AUTOMOTIVE CONTROL SYSTEMS

This section presents an overview of modeling and simulation techniques currently used in the automotive industry. Generally speaking, modeling is the process of developing an appropriate system model $\mathcal{M}$. Simulation is the process of obtaining particular behaviors $\phi(\mathcal{M}, p, u)$.

A. Automotive Engine Control Challenges

Automotive control software has been increasing in scale and complexity for years and is expected to continue on this trend for the foreseeable future. One main component of automotive control systems is the engine control system. Lines of code for engine control can be measured in the millions, much higher than the typical scale of code used in, for example, a typical jet airliner [12].

There are many reasons for the increase in code complexity, including the need to respond to ever-increasing government-mandated regulatory requirements on vehicle efficiency, emissions, and diagnostics. One such standard driving the increasing efficiency and emissions standards is the Corporate Average Fuel Economy (CAFE) standard, which is a U.S. regulation defining the requirements for vehicle efficiency based on an automaker's fleet (i.e., total vehicles produced) [36].

The continually increasing standards are met using various approaches, such as by adding new energy-saving and emission-reducing technologies, like exhaust gas recirculation (EGR) systems. EGR systems help to increase overall engine efficiency, but at a cost of adding a new physical system that must be regulated by the engine control unit (ECU), which increases the control software complexity. Another example of how the increasingly stringent standard is met is by including a variable geometry turbo-charger (VGT). VGT technologies allow the engine to operate more responsively, which allows the engine to respond better to transients (leading to an increase in overall efficiency), but this technology also requires an increase in ECU software complexity to control.

B. Modeling paradigms used in PTC applications

Simulation-guided approaches to verification and test require a model $\mathcal{M}$ of the system under consideration. Here, $\mathcal{M}$ can contain representations of the embedded controller, the environment (plant), or both (referred to as the *closed-loop* case).

Models of the controller can range from simple, continuous-time representations (e.g., transfer functions), to complex models that capture implementation details, such as sensor and actuator saturation, computation time and communication delays, sensor noise, and actuator error. In some cases, computer code can be automatically generated from the detailed controller models for deployment on the real-time platform.

Developing models of the environment is a significant challenge for the automotive domain. The presence of complex interactions of mechanical, electrical, and chemical phenomena make the problem of capturing the dynamical system behavior difficult. Various modeling paradigms are used based on the requirements of the modeling scenario.

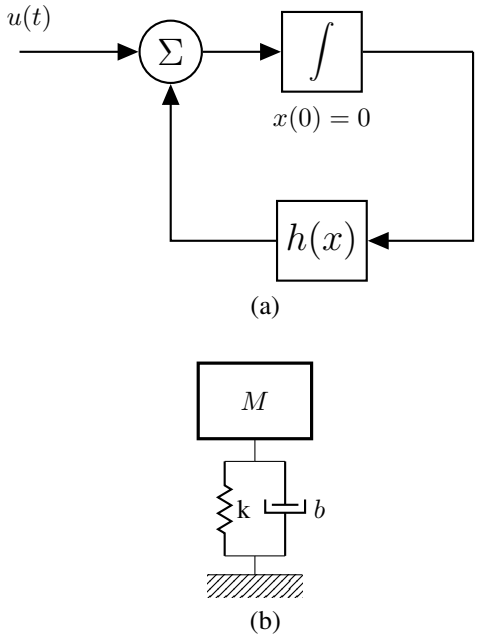

Fig. 1. (a) Example of a causal model; (b) Example of an acausal model.

Causal, lumped parameter models are the most common type used to capture plant dynamics. In this case, system dynamics are described by ordinary differential equations (ODEs) and can be modeled using block diagrams, connected by edges that represent paths for unidirectional signal flow. This is the type of model created in, for example, Simulink®.

Figure 1-(a) provides an example of a block diagram representing a causal model. The system in the figure represents the ODE $\dot{x}(t) = h(x(t)) + u(t)$, where the system takes an external input, $u(t)$, and the integrator requires an initial condition. Note that the arrows in the diagram indicate that a signal value emanates from one block and is used as input to another (e.g., the state of the integrator provides a value to the $h(x)$ function block, which the function block treats as an input).

Acausal, lumped parameter modeling techniques are also used to capture plant behaviors. In an acausal model, system dynamics are described by differential algebraic equations (DAEs). As with causal systems, acausal systems can be modeled using block diagrams; however, for acausal models, edges between blocks represent constraints involving variables from the connected subsystems. This type of model can be created using, for example, the Simscape™ tool.

Figure 1-(b) provides an example of a block diagram representing an acausal model. The block labeled M represents a physical object with mass M. The sawtooth line represents a spring with spring constant k. The element to the right of the spring represents a dashpot (i.e., a mechanical element that provides damping to the system) with damping constant b. The mass block is associated with the dynamic equation $M\ddot{x}_1(t) = gM + \sum_i F_i(t)$, where $x_1(t)$ is the vertical position of the mass, g is the acceleration due to gravity and $\sum_i F_i(t)$ is the sum of all of the external forces on the mass. The lines connecting the mass, spring, and dashpot together represent a constraint. In this case, the constraint is $\sum_i F_i(t) = -k(x_1(t) - x_2(t)) - b(\dot{x}_1(t) - \dot{x}_2(t))$. The component on the bottom of the diagram represents *ground*. The lines connecting the spring and the dashpot to ground represent the constraints $x_2(t) = 0$ and $\dot{x}_2(t) = 0$.

Acausal models allow for a more composable approach to modeling, at the expense of slightly less efficient computations of simulations. Since typical control design methods are set in a framework of ODEs, control designers are often more comfortable dealing with ODEs, and so causal models are more commonly used in the automotive domain, though we note that this is beginning to change.

In some cases where lumped parameter models are insufficient for capturing certain critical physical phenomena, distributed parameter models can be used. Partial differential equations (PDEs) are used to model this type of system. PDEs can be used, for example, in cases where vital aspects of the system required to define its behavior over time are spread across a relatively wide physical area[1].

PDEs are used sparingly in automotive control design, as they are computationally expensive. As an example, consider the dynamics of an engine aftertreatment system. For some analysis, it is necessary to accurately model the distribution of heat along the length of the catalyst, which is best represented using a PDE. While some designers will choose to model the system as a PDE, others will opt to create an ODE approximation of the dynamics by discretizing the spacial distribution of heat within the catalyst. The ODE approximation will less accurately capture the dynamics but will allow for more efficient simulations of the behaviors (for example, see [11]).

Finite element analysis (FEA) models are used to capture physical phenomena best described by boundary value problems defined over some spatial distribution (e.g., electromagnetic fields, temperature variation, stress/strain, fluid dynamics). FEA can be used to estimate critical aspects of the systems design. Note, however, that this type of model often requires a significant setup time and information from outside of the domain of control development, and it is also computationally expensive. Therefore, it is seldom used to model embedded control systems.

C. Simulation and testing for PTC applications

The term *simulation* refers to the process of obtaining a numerical estimation of system behaviors $\phi(\mathcal{M}, \hat{P}, \hat{U})$ for a specific set of operating conditions given by $\hat{P}$ and $\hat{U}$. In practice, simulations are obtained using specialized software. The simulation software usually provides an environment to specify $\mathcal{M}$, which can represent the control system software and possibly a representation of the environment. In the automotive engine control design context, $\mathcal{M}$ would include a representation of the ECU (including the control algorithm)

[1]For example, PDEs can be used if it becomes necessary to model the electrical system in an engine using a transmission line representation, where the transmission line resistance is represented by not just a single resistance value but a distributed resistance over some distance of wire.

and the engine. In the automotive industry, Simulink® is a commonly used tool for this type of activity. Engineers use simulation to perform preliminary tuning of control parameters, to estimate performance of a given control design, and also to debug the design.

Control design simulation has parallels in the program analysis domain. Some program testing standards require that each line of code be visited using some testing approach, for example, testing requirements based on the modified condition/decision coverage (MCDC) criterion [14]. In program analysis, it is common to refer to tests as *concrete executions*, or runs, of a software system. These concrete executions are analogous to simulations of an embedded control system; however, one main difference is that the software system executions are actual instances of behaviors of the system, whereas simulations of an embedded control system are approximations based on parameter estimations and simplified representations of the dynamic behaviors for the physical system.

There are inevitably errors between the estimations for the physical parameters and the actual system parameters, due to issues such as machining tolerances, imperfections in materials processing, and incorrect assumptions about the operating environment, such as ambient temperature. Further, physical phenomena are not modeled exactly. Even for the most detailed environment models, there are behaviors and interactions between system components that are not modeled (sometimes referred to as second or third order effects)[2].

The term *testing* usually refers to the process of performing experiments under various designated operating conditions (when $\mathcal{M}$ is a physical manifestation of the system), but testing can also refer to a process of collecting information from several simulations computed using various designated parameters (when $\mathcal{M}$ is a computer model of the system). The goal of testing is to check if behaviors $\phi(\mathcal{M}, p, u)$ obtained from $\mathcal{M}$ conform to certain requirements. Though testing provides a means to check if a system design meets its requirements, it does not provide an exhaustive analysis of the set of behaviors that the system can exhibit. This limitation occurs for two reasons. First, tests only evaluate the system behaviors for a specific set of operating conditions, whereas there are typically an infinite set of possible operating conditions. Second, the test conditions may not accurately reflect the manner in which the system will be used once it is deployed[3].

In the past, testing was typically done in an ad hoc manner. A control engineer would manually select some input values for a set of operating parameters, for example, P+I controller gains and initial conditions such as engine speed. The selection of such test inputs was usually based on the engineer's experience and insights about which inputs would represent nominal operating behavior, worst-case behavior, and so on. With the increasing complexity of embedded control systems, techniques relying on manual insights are no longer scalable, and are thus being replaced by automated testing approaches.

Current testing approaches for the automotive domain involve various incarnations of the control system: from computer models to experiments on the physical system. The testing approaches we describe here are easiest to apply to computer models (e.g., open-loop testing and the MIL scenario described below), but it is feasible to apply them to many different testing approaches.

Open-loop testing. Typically, the first step in testing a controller is open-loop testing to validate that the controller meets its functional requirements. In coverage-based open-loop testing, the model under test is viewed as though it were a software program, and the goal is to automatically and intelligently select inputs to the model so as to maximize a software code-coverage metric. There are several code-coverage metrics, and the most popular in the automotive domain is MCDC (see above). Tools such as the Reactis®, Simulink Design Verifier™, and TestWeaver use different approaches to perform coverage-based testing.

The Reactis Tester tool uses guided simulation to evaluate open-loop controller models; this is a patented technique to generate test inputs using a combination of random and targeted methods. The targeted phase of the tool uses data structures to store intermediate states, and constraint solving algorithms to search for previously uncovered coverage targets [5].

Simulink Design Verifier™ uses SAT-solving techniques provided by the Prover tool to automatically generate test inputs to maximize coverage criteria [37]. In our experience, this tool cannot efficiently process closed-loop PTC models, due to the complexity of the associated environment models.

Closed-loop testing. We describe several of the more commonly used closed-loop testing approaches below. The test scenarios are presented in the order in which they might typically occur during a standard development cycle.

- **Model-In-The-Loop (MIL):** In this testing scenario, $\mathcal{M}$ is a computer model containing a representation of both the controller and the environment, and simulations are computed on a host PC. MIL testing is the scenario that is most applicable to the simulation-guided approaches presented herein.
- **Software-In-The-Loop (SIL):** In this testing scenario, $\mathcal{M}$ is a computer model composed of a representation of the plant interacting with a controller, which is implemented with production computer code.
- **Processor-In-The-Loop (PIL):** In this test scenario, $\mathcal{M}$ is the real-time platform, running production code, connected directly to a host PC, which is running a computer model of the plant. The communication between the plant and the controller in this case uses a direct communication link, such as ethernet or CAN bus.
- **Harware-In-The-Loop (HIL):** In this test scenario, $\mathcal{M}$

[2]We are reminded of a quote by the mathematician George E. P. Box: "...all models are wrong, but some are useful."

[3]Consider the difference between testing an engine inside of a test cell versus driving the vehicle on a busy highway.

is composed of the real-time platform and a *virtual* plant, which is a computer model of the plant running on specialize hardware. The virtual plant receives electronic inputs from the controller actuator output and produces electronic outputs, which are received by the controller as sensor inputs. This is as opposed to the direct communications link that is used in the case of PIL testing.

The TestWeaver tool by QTronic evaluates closed-loop models using simulations guided by a search algorithm (based on proprietary heuristics) to attempt to obtain a high degree of coverage and also to violate system requirements [39]. TestWeaver relies on the user to quantize the input values and the time domain and also to manually identify system variables that are most sensitive to the inputs. This user intervention requires an understanding of the system dynamics and engineering intuition to use effectively.

In Sec. III, we describe simulation-guided approaches that overcome the limitations of simulation and testing existing tools. We consider techniques to automatically identify design bugs using systematic testing techniques based on simulation.

D. Verification for PTC applications

The term *verification* is used in the computer science literature to refer to the process of formally deciding whether a given system model satisfies a given specification [16]. Model checking is one technology that can perform verification. Originally, model checking was developed to check properties of finite-state systems. It was first applied to systems such as logic circuits and later to computer software [8].

Testing and verification are closely related, with one main difference. While testing determines whether $\phi(\mathcal{M}, \hat{P}, \hat{U}) \models \varphi$ for some finite $\hat{P}$ and $\hat{U}$, verification determines whether $\mathcal{M} \models \varphi$ over the complete parameter and input spaces P and U. In this sense, verification provides a stronger result than testing.

While testing is often performed without the benefit of a formal specification φ, a specification is required to perform verification. A specification φ for a verification tool is usually supplied in the form of a special language, call a *temporal logic*, which employs operators that are used to indicate desired system behavior over time. As an example, one such language, signal temporal logic (STL), can express timed operators over fixed time ranges [38], such as the following example:

$$\varphi \quad = \quad \Box_{[1.0,2.0]} x(t) < 10.0, \tag{1}$$

which means the signal x must remain less than 10.0 between the times $t = 1.0$ and $t = 2.0$.

Any verification procedure will return one of the following results:

- **Verified**: This results is returned if the procedure determines that φ holds for all of the cases. If a procedure returns a *verified* result only if $\phi(\mathcal{M}, P, U) \models \varphi$, we call the procedure *sound*. We call the capacity of a technique to return a sound result *soundness*.
- **Not Verified**: This result is returned if the procedure cannot certify that φ holds for all cases. In some instances a *counterexample* is also returned, which is a concrete p and u that demonstrates that $\phi(\mathcal{M}, p, u) \not\models \varphi$.

Applying verification techniques to real-time control systems, such as PTC systems, is a challenging task. These systems can be classified as *hybrid systems*, which are systems that exhibit both continuous and discrete behaviors; the general problem of verifying hybrid systems is known to be *undecidable* [30]. This undecidability result means that it is provable that no computer algorithm can be created to solve the general problem of deciding whether a given system satisfies a formal specification.

Tools exist for verifying specific classes of hybrid systems, but each suffers from significant limitations. SpaceEx verifies hybrid systems with affine continuous dynamics and polyhedral switching constraints [25]. The UPAAL tool can verify complex hybrid systems, but it is limited to timed systems (i.e., systems with dynamics define by constant derivatives) [9]. The Flow* tool [13] handles systems given by ordinary differential equations (ODEs) that are expressed as polynomial functions of the state variables. The Simulink Design Verifier™ (SLDV) from the MathWorks® handles hybrid system designs [37], but in our experience, it cannot handle large, industrial system models.

In Sec. IV, we describe emerging technologies designed to overcome the limitations of existing tools. We consider approaches to facilitate the verification process using information obtained from simulation results.

III. EMERGING SIMULATION-GUIDED TESTING APPROACHES

This section presents an overview of several emerging simulation-guided approaches to testing, which are all based on the notion of falsification. Optimization-guided falsification is an emerging approach for testing closed-loop models that intelligently obtains test inputs to expose undesirable model behaviors. The inputs to a falsification algorithm are a closed-loop model, a set of correctness requirements, and a definition of the operating conditions of the exogenous inputs to the closed-loop model. The overall architecture of an optimization-guided falsification tool is as shown in Fig. 2; two key components are a simulation engine and an optimizer.

The model $\mathcal{M}$ takes as input initial conditions and parameter values for the closed-loop model (i.e., p_i), and time-series values for the model inputs (i.e., u_i), and numerically computes time-series values for the model outputs (i.e., $\phi(\mathcal{M}, p_i, u_i)$). The optimizer encapsulates two key elements: a cost function mapping each output $\phi(\mathcal{M}, p_i, u_i)$ to some real number, and a search procedure that, based on the current model inputs (p_i, u_i) and the cost $c(y_i)$, picks promising values (p_{i+1}, u_{i+1}) for the model inputs and parameter values. Note that the cost function is defined with respect

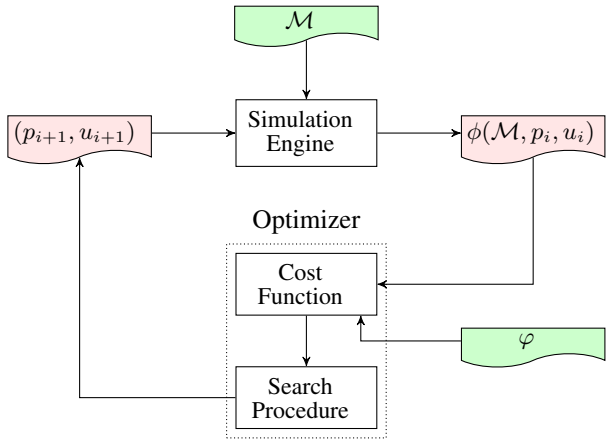

Fig. 2. Optimization-guided Falsification

to the correctness requirement(s), and can always be defined such that a negative cost corresponds to a violation of the requirements. Thus, the goal of the optimizer is to minimize the cost defined on the output space by picking appropriate (p_i, u_i) pairs. The tool proceeds iteratively evaluating costs $c(y_i)$ corresponding to (p_i, u_i) pairs until it finds a negative cost or the user-specified limit on the number of iterations expires.

We now elaborate on specific instances of the optimization-guided falsification framework.

A. S-TaLiRo

The S-TaLiRo tool uses temporal logic requirements and a number of stochastic optimization algorithms to perform optimization-guided falsification [4].

Temporal logic has long been used within the formal verification community to specify requirements on the time-varying behavior of a system. S-TaLiRo allows specification of requirements using Metric Temporal Logic (MTL). The key elements of this logic are Boolean propositions over signal values, temporal operators such as *globally* ($\Box$), *eventually* ($\Diamond$), and *until* ($\mathbf{U}$) indexed by time-intervals, and syntactically well-formed combinations thereof. For example, consider the formula $\Box_{[0,10]}p \implies \Diamond_{[0,1]}q$, where p is a Boolean proposition defined as $x > 0$ and q is defined as $y < 0$. This formula means that globally for all times t between 0 and 10 seconds, if $x(t) > 0$ is *true*, then eventually between times t and $t+1$, $y(t) < 0$ is *true*. In [24], the authors introduce a quantitative interpretation for metric temporal logic that, given an MTL formula φ and a signal (trace) specified as a sequence of time-value pairs, computes a *robustness* value. The interpretation is that if the robustness value is positive, the trace satisfies φ, and if negative it does not satisfy φ. Thus, S-TaLiRo simply uses the robustness value of a trace with respect to the given requirement(s) as a cost function.

The search for input and parameter values for S-TaLiRo uses stochastic global optimization algorithms. The first step is to parametrize the input signal space using a finite set of evenly spaced control points. For example, if a model has 1 exogenous input u with 3 control points, and the simulation time horizon is 10 seconds, then S-TaLiRo picks 3 values corresponding to $u(0)$, $u(5)$ and $u(10)$. The actual $u(t)$ is then obtained by using a user-defined interpolation (piecewise linear, splines, etc.) across the chosen control points. The stochastic global optimization algorithms then try to find values for the control points that lead to an output with a lower cost. In particular, S-TaLiRo permits use of algorithms such as uniform random search, simulated annealing, cross entropy, genetic algorithms and ant-colony optimization. The details of the algorithms are beyond the scope of this paper, and we refer the interested reader to [40], [3], [43], [4] for further details.

B. Breach

The Breach tool [19] uses Signal Temporal Logic (STL), which is a variant of Metric Temporal Logic to specify requirements. While the quantitative interpretation of STL varies only slightly from MTL, it allows for more efficient algorithms [20] to compute the robustness value of a trace with respect to an STL property. The other main difference is that Breach uses a nonlinear global optimizer based on the Nelder-Mead simplex-based algorithm.

Breach supports mining temporal requirements from closed-loop models[4]. In such a framework, the user provides template requirements where certain parameter values are left unspecified. The tool integrates an efficient parameter synthesis algorithm to obtain candidate requirements from simulation traces with its falsification core that attempts to falsify the candidate requirement. Counterexamples obtained by falsification are used to refine the candidate requirement, and the mining procedure terminates when a user-specified bound on the number of iterations expires or no counterexample is found by the falsifier [32]. The requirement mining tool can itself be thus used as a more nuanced falsification tool, where the optimizer used for falsification is guided by intermediate candidate requirements. Other features of Breach include the ability to compute the sensitivity of traces to the initial conditions, which can be used to obtain completeness results by performing systematic simulation.

C. RRT-REX

As an alternative to coverage-based testing that uses metrics inspired by software testing, previous research has explored the Star-discrepancy measure for coverage; this is a statistical notion that quantifies how well a continuous state-space is covered (by states discovered with simulations) [17]. The key idea in this approach is to use the Star-discrepancy measure to guide state-space exploration using the Rapidly exploring Random Trees (RRT) algorithm.

The RRT algorithm builds a tree in which a node at depth i in the tree represents a continuous state $\mathbf{x}(t_i)$ of the model at time t_i , and edges $(\mathbf{x}(t_i), \mathbf{x}(t_{i+1}))$ are labeled by inputs u_i that cause the model to evolve from $\mathbf{x}(t_i)$ to $\mathbf{x}(t_{i+1})$ under the action of the input signal segment from

[4]We remark that the tool S-TaLiRo also supports mining temporal requirements.

time t_i to t_{i+1} parameterized by the value u_i. A rooted path in the tree $\mathbf{x}(t_0), \mathbf{x}(t_1), \ldots, \mathbf{x}(t_n)$ corresponds to a partial simulation trace of the model-output corresponding to the input signal $u(t)$ obtained by splicing together the input signals corresponding to the values $u_0, u_1, \ldots, u_{n-1}$ on the edges in the path.

RRT-REX [1] is a new tool to explore the state-space of a Simulink® model that provides additional guidance to the RRT algorithm by associating temporal robustness values for partial simulation traces corresponding to the paths in the tree. Specifically, the RRT algorithm is biased to grow the tree from an existing node such that the new suffix leads to a partial simulation trace with a lower robustness value. While preliminary evaluations look promising, a mature implementation of the tool is under development.

D. Multiple-shooting based approaches

In [46], the authors propose using short disconnected simulation traces to find an abstract example of a falsifying trace, and using an optimizer to attempt to splice the short traces together to identify a concrete falsifying trace. The preliminary results for finding rare bugs in linear hybrid systems seemed promising. A significant recent revision of this work removes the dependence on an optimizer [2]. The key idea here is to incorporate elements of Counterexample-guided Abstraction Refinement. At each step the tool performs a short simulation of time δt from a chosen start state to obtain a trace segment, and then randomly chooses a new set of start states by sampling in an ϵ-neighborhood of the end-point of the trace segment. The algorithm is initialized by randomly sampling the initial set of states. A *segmented trace* is a collection of trace segments thus obtained, such that the beginning of the first segment is an initial state and the end-point of the last segment is an unsafe state. Once the algorithm obtains a few promising segmented trajectories it can then choose to refine these traces by decreasing ϵ or δt. At the end of each search step, before refining the traces, the user can choose to perform a concretization step that involves doing a complete simulation from the initial states corresponding to the most promising segmented trajectories. Multiple-shooting based falsification shows promise for finding bugs in closed-loop models. A next step for these approaches is to investigate their applicability to practical closed-loop control models, where the models could be described in Simulink® or where there may be practical concerns regarding full state-observability.

IV. EMERGING SIMULATION-GUIDED VERIFICATION APPROACHES

We discuss emerging techniques to perform verification for PTC systems using simulation-guided approaches and focus on two promising new techniques that leverage information gleaned from simulations to facilitate a verification task. This idea is related to similar ideas in program verification. Some program verification technologies employ satisfiability modulo theories (SMT) tools, which are used to solve a general class of logical queries [18]. When these techniques fail due to complexity of the verification task, it is possible to simplify the problem by adding constraints to the verification query. Appropriate constraints can be constructed based on concrete system executions.

A. Simulation-guided Lyapunov analysis

A system is asymptotically stable if any system trace converges to an equilibrium point of the system. Asymptotic stability can be used to prove a system satisfies desirable performance criteria, such as the convergence of the system state to a given reference value. An effective technique to prove stability is to supply a so-called *Lyapunov function* v that satisfies the following *Lyapunov conditions* in a given domain: (1) v is positive everywhere in the domain (except at the equilibrium, where it is 0) , and (2) the time-derivative of v along the system dynamics is negative everywhere in the domain (except at the equilibrium, where it is 0). In addition to proving stability, Lyapunov functions can also be used for characterizing performance bounds and to perform safety verification. We elaborate on each of these three verification tasks that may be addressed using Lypaunov functions below.

- **Stability**: For this verification task, $\varphi :=$"The system is stable". To verify this property, it is sufficient to identify a function v that satisfies (1) and (2).
- **Performance Bounds**: For this verification task, $\varphi :=$"The system remains with some set $\mathcal{S}$". To verify this property, it is sufficient to identify a function v that satisfies (1) and (2) and a levelset $\hat{\mathcal{S}}$ of v (e.g., $\hat{\mathcal{S}} = \{\mathbf{x} | v(\mathbf{x}) \leq l\}$), such that $\hat{\mathcal{S}} \subseteq \mathcal{S}$. Note that this only guarantees φ if the system is initiated within $\hat{\mathcal{S}}$. For this reason, it is often preferable to identify a $\hat{\mathcal{S}}$ that is of maximal size.
- **Barrier Certificate**: For this verification task, $\varphi :=$"Given that the system is initiated in $\mathcal{X}_0$, it will never reach $\mathcal{F}$". To verify this property, it is sufficient to identify a function v that satisfies (1) and (2) and levelset $\hat{\mathcal{S}}$ of v such that $\mathcal{X}_0 \subseteq \hat{\mathcal{S}}$ and $\hat{\mathcal{S}} \cap \mathcal{F} = \emptyset$.

The search for a Lyapunov function is widely recognized as a hard problem. In [35] the authors present a simulation-guided approach to synthesize Lyapunov functions. The key idea, which is based on [44], is to assume that the desired Lyapunov function has a certain parameterized template form: a Sum-of-Squares polynomial of fixed degree. A set of linear constraints on the parameters in the Lyapunov function are obtained from concrete simulation traces. Given a set of such constraints, the search for a Lyapunov function reduces to solving a linear program (LP) to obtain a *candidate Lyapunov function*.

A key step is then to use a stochastic global optimizer to search the region of interest for states that violate the Lyapunov conditions for the given candidate. The search is guided by a cost function that is based on the time-derivative of the candidate Lyapunov function; if the minimum cost is less than zero, then the minimizing argument provides a witness that the candidate Lyapunov function is invalid. After the global optimizer obtains counterexamples $\phi(\mathcal{M}, P_c, U_c)$, the associated linear constraints are included in the LP

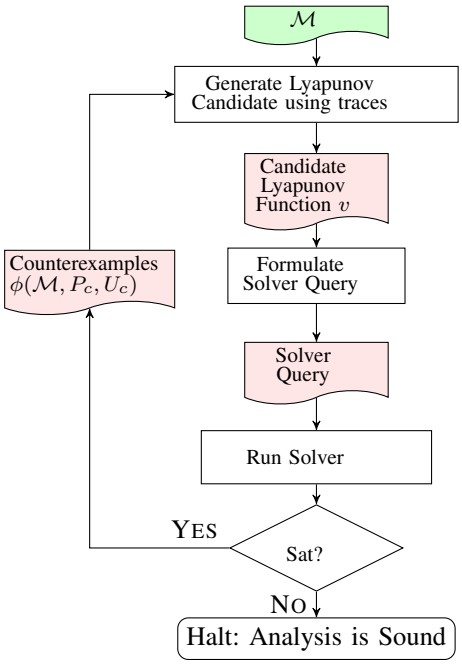

Fig. 3. Incorporating solver technologies to verify soundness of the Lyapunov anlyses.

problem and the candidate Lyapunov function is updated. The process terminates when the global optimizer is unable to identify counterexamples. As global optimization is not exhaustive, the approach uses nonlinear SMT solvers based on interval constraint-propagation (such as dReal [26]) and symbolic tools relying on quantifier elimination as implemented by the `Reduce` command in Mathematica [45] to check validity of the final candidate Lyapunov function. If the check fails, the approach adds the violating points as further constraints, and restarts computation. Figure 3 illustrates the procedure.

To illustrate the applicability of this approach to the PTC domain, we briefly describe Example 5 from [35], which involves the verification of an air-fuel (A/F) ratio control system. The specification is that the A/F ratio should remain within 10% of the stoichiometric value if the system is initiated within a given set ($\|\mathbf{x}\|_2 < 0.02$). For this system, there are four system states, and $\mathcal{M}$ is given by a set of nonlinear ODEs. A procedure similar to the one illustrated in Fig. 3 was able to identify barrier certificate for this system in $1,413.73$ seconds using simulations that explored a total of $258,078$ system states. Verification by the dReal tool took $1,157.42$ seconds. We refer the reader to the paper for further details regarding this example.

A related approach was proposed by Gupta et al. [28] for program analysis. In that work, the authors used information from program executions to construct linear constraints, then a constraint solver is employed to determine valuations for each of a set of parameters. The result is a ranking function or a program invariant, which can be used to perform verification for the program code.

B. Simulation-guided contraction analysis

In contraction analysis or incremental stability analysis, instead of asking whether system trajectories converge to a known nominal behavior, one asks the question if the trajectories converge to each other. This type of property could establish, for example, that the system converges to a reference trajectory. Such a property is established with the help of a *contraction metric*, which is analogous to a Lyapunov function, but in the tangent space of the manifold describing the system's state-space. Of particular interest is the application of contraction analysis to bounded-time reachability analysis [33], [21], [31].

Related work uses *auto-bisimulation* functions to show that for a given finite-time simulation trace with initial condition $\mathbf{x}(0)$, finite-time trajectories that start inside a ball of radius δ centered at $\mathbf{x}(0)$ lie within a "tube" of radius δ around $\mathbf{x}(t)$ [27], [23]. This allows performing a comprehensive verification analysis by exploring only a few candidate simulation traces.

The ability to perform contraction analysis hinges on the ability to identify a contraction metric for a given system, which is a difficult task in general. The search for a contraction metric can, in some cases, be formulated as a convex optimization problem, which can be solved using semidefinite programming optimization tools [6]. Recent work [7] proposes a simulation-guided approach to find contraction metrics, in similar vein to the approach described in [35]. Here, the contraction metric is defined by a matrix, the entries of which are polynomials of some fixed degree but undetermined coefficients. Using simulation traces, a set of necessary constraints for the contraction metric are obtained, each constraint is a linear matrix inequality (LMI). A solver based on semidefinite-programming is used to obtain a candidate contraction metric. As in [35], a tandem of global optimization and nonlinear solving technology (to solve slightly more sophisticated contraction constraints) is used to obtain counterexamples to refine the contraction metric. If no counterexample is found, the algorithm terminates.

V. CONCLUSIONS

We discussed particular challenges that the automotive industry faces in modeling and quality checking for powertrain control (PTC) software systems. We described several modeling paradigms and testing scenarios used in the automotive industry, and we presented a series of new automated techniques that can be used to perform verification and quality checking for automotive control applications.

We summarize the new techniques in Table I. For each technique in the table, we indicate the section in which the technique appears in the paper, the type of analysis that the technique performs (i.e., falsification or verification), and the relative maturity level of the technology. Maturity levels are indicated as a number between 1 and 5; 1 indicates that the technique has only been implemented as an academic tool, and 5 indicates that the technique is implemented in a fully commercialized and supported tool.

TABLE I

Emerging Simulation-guided Techniques for PTC design analysis.

Maturity Scale (1–5): **1**: Acadmemic implementation, **2**: Significant user effort required, **3**: Can handle some industrial models, **4**: Ready for industrial deployment, but not commercialized, **5**: Fully commercialized and supprted.

Technique	Section	Type of Analysis	Maturity
S-TaLiRo	III-A	Falsification	4
Breach	III-B	Falsification	4
RRT-REX	III-C	Falsification	3
Multiple-shooting	III-D	Falsification	2
Lyapunov Analysis	IV-A	Verification	1
Contraction Analysis	IV-B	Verification	1

REFERENCES

[1] Rrt-rex: Efficient guiding strategies for testing of temporal properties of hybrid systems. *Pending publication*, 2014.

[2] Jyotirmoy V. Deshmukh Aditya Zutshi, Sriram Sankaranarayanan and James Kapinski. Multiple-shooting, CEGAR-based falsification for hybrid systems. In *To appear in Proc. of Embedded Software*, 2014.

[3] Yashwanth Singh Rahul Annapureddy and Georgios E. Fainekos. Ant colonies for temporal logic falsification of hybrid systems. In *Proceedings of the 36th Annual Conference of IEEE Industrial Electronics*, pages 91–96, 2010.

[4] Yashwanth Annpureddy, Che Liu, Georgios E. Fainekos, and Sriram 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.

[5] Anonymous. Model-based testing and validation of control software with reactis. 2003.

[6] Erin M. Aylward, Pablo A. Parrilo, and Jean-Jacques E. Slotine. Stability and robustness analysis of nonlinear systems via contraction metrics and SOS programming. *Automatica*, 44(8):2163–2170, 2008.

[7] Ayca Balkan, Jyotirmoy V. Deshmukh, James Kapinski, and Paulo Tabuada. Simulation-guided contraction analysis. *Indian Control Conference (*to appear*)*, 2015.

[8] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, July 2011.

[9] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Hakansson, Paul Petterson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 125–126. IEEE, 2006.

[10] Yves Bertot and Pierre Cast'eran. *Interactive Theorem Proving and Program Development*. Springer, June 2004.

[11] Edward J Bissett. Mathematical model of the thermal regeneration of a wall-flow monolith diesel particulate filter. *Chemical Engineering Science*, (7/8):1233–1244, January 1984.

[12] Robert N. Charette. This Car Runs on Code. *IEEE Spectrum*, February 2009.

[13] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *Computer Aided Verification*, pages 258–263. Springer, 2013.

[14] John Joseph Chilenski and Steven P Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[15] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer Berlin Heidelberg, 2004.

[16] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

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

[18] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[19] Alexandre Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *CAV*, pages 167–170, 2010.

[20] Alexandre Donzé, Thomas Ferrère, and Oded Maler. Efficient robust monitoring for STL. In *Computer Aided Verification*, pages 264–279. Springer, 2013.

[21] Parasara Sridhar Duggirala, Sayan Mitra, and Mahesh Viswanathan. Verification of annotated models from executions. In *Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada, September 29 - Oct. 4, 2013*, pages 1–10, 2013.

[22] Pr Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217(0):5 – 21, 2008. Proceedings of the 3rd International Workshop on Systems Software Verication (SSV 2008).

[23] Georgios E. Fainekos, Antoine Girard, and GeorgeJ. Pappas. Temporal logic verification using simulation. In Eugene Asarin and Patricia Bouyer, editors, *Formal Modeling and Analysis of Timed Systems*, volume 4202 of *Lecture Notes in Computer Science*, pages 171–186. Springer Berlin Heidelberg, 2006.

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

[25] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In *Computer Aided Verification*, pages 379–395. Springer, 2011.

[26] Sicun Gao, Jeremy Avigad, and Edmund M Clarke. δ-complete decision procedures for satisfiability over the reals. In *Automated Reasoning*, pages 286–300. Springer, 2012.

[27] Antoine Girard and George J. Pappas. Approximate bisimulation: A bridge between computer science and control theory. *Eur. J. Control*, 17(5-6):568–578, 2011.

[28] Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. From tests to proofs. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems*, pages 262–276, 2009.

[29] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *Model Checking Software*, pages 235–239. Springer, 2003.

[30] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? In *Journal of Computer and System Sciences*, pages 373–382. ACM Press, 1995.

[31] Zhenqi Huang and Sayan Mitra. Computing bounded reach sets from sampled simulation traces. In *Hybrid Systems: Computation and Control (part of CPS Week 2012), HSCC'12, Beijing, China, April 17-19, 2012*, pages 291–294, 2012.

[32] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V. Deshmukh, and Sanjit A. Seshia. Mining requirements from closed-loop control models. In *Proceedings of the 16th international conference on Hybrid systems: computation and control*, pages 43–52, 2013.

[33] A. Agung Julius, Georgious E. Fainekos, Madhukar Anand, Insup Lee, and George J. Pappas. Robust Test Generation and Coverage for Hybrid Systems. In *Proc. of Hybrid Systems: Computation and Control*, 2007.

[34] T. Kaga, M. Adachi, I. Hosotani, and M. Konishi. Validation of control software specification using design interests extraction and model checking. In *Proc. of the 2012 World Congress and Exhibition*.

[35] James Kapinski, Jyotirmoy V. Deshmukh, Sriram Sankaranarayanan, and Nikos Aréchiga. Simulation-guided lyapunov analysis for hybrid dynamical systems. In *Hybrid Systems: Computation and Control*, 2014.

[36] Thomas H. Klier and Joshua Linn. Newvehicle characteristics and the cost of the corporate average fuel economy standard. *RAND Journal of Economics*, 43(1):186–213, 2012.

[37] Florian Leitner and Stefan Leue. Simulink design verifier vs. SPIN a comparative case study. In *Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems*, 2008.

[38] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *In: Proceedings of FORMATS-FTRTFT. Volume 3253 of LNCS*, pages 152–166. Springer, 2004.

[39] Jakob Mauss Mugur Tatar. Systematic test and validation of complex

embedded systems. In *Embedded Real Time Software and Systems*, 2014.

[40] Truong Nghiem, Sriram Sankaranarayanan, Georgios E. Fainekos, Franjo Ivancic, Aarti Gupta, and George 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.

[41] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[42] Sam Owre, Sreeranga Rajan, John M Rushby, Natarajan Shankar, and Mandayam Srivas. Pvs: Combining specification, proof checking, and model checking. In *Computer Aided Verification*, pages 411–414. Springer, 1996.

[43] Sriram Sankaranarayanan and Georgios E. Fainekos. Falsification of temporal properties of hybrid systems using the cross-entropy method. In *ACM International Conference on Hybrid Systems: Computation and Control*, 2012.

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

[45] Stephen Wolfram. *The Mathematica® Book, Version 4*. Cambridge University Press, 1999.

[46] Aditya Zutshi, Sriram Sankaranarayanan, Jyotirmoy V Deshmukh, and James Kapinski. A trajectory splicing approach to concretizing counterexamples for hybrid systems. In *Decision and Control (CDC), 2013 IEEE 52nd Annual Conference on*, pages 3918–3925. IEEE, 2013.