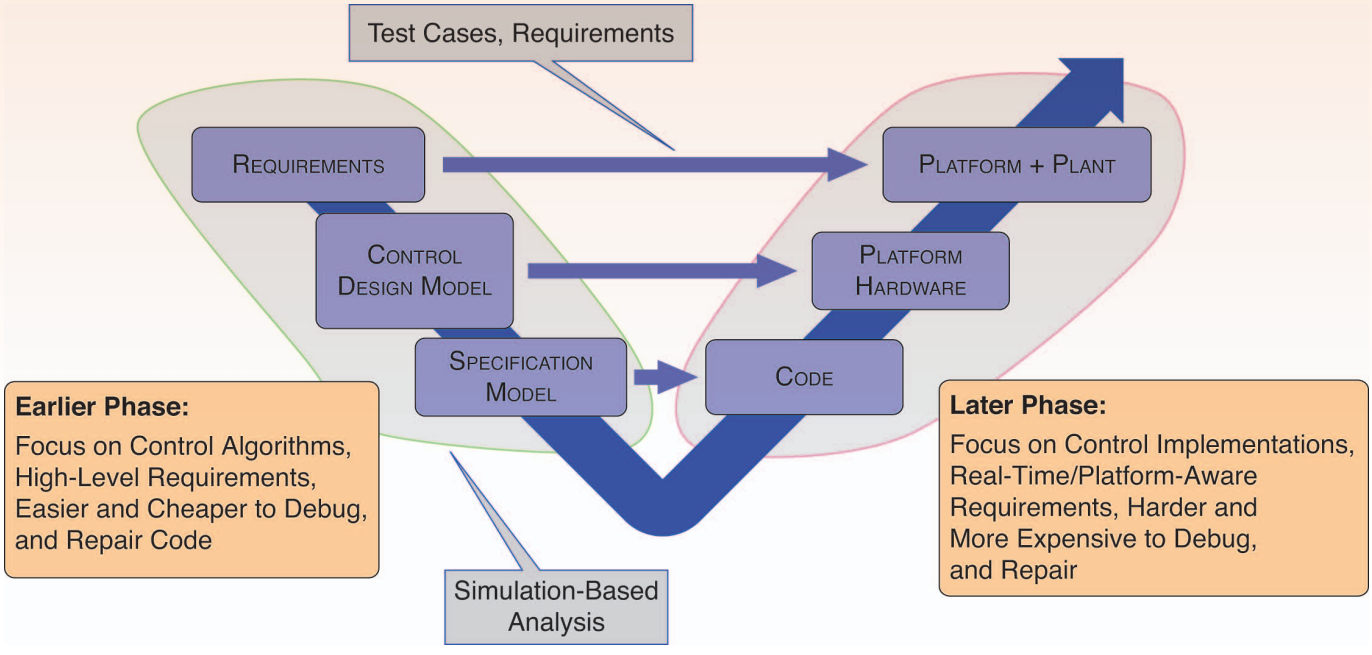
Simulation-Based Approaches for Verification of Embedded Control Systems

AN OVERVIEW OF TRADITIONAL AND ADVANCED MODELING, TESTING, AND VERIFICATION TECHNIQUES



JAMES KAPINSKI, JYOTIRMOY V. DESHMUKH, XIAOQING JIN, HISAHIRO ITO, and KEN BUTTS

Designers of industrial embedded control systems, such as automotive, aerospace, and medical-device control systems, use verification and testing activities to increase their confidence that performance requirements and safety standards are met. Since testing and verification tasks account for a significant portion of the development effort, increasing the efficiency of testing and verification will have a significant impact on the total development cost. Existing and emerging simulation-based approaches offer improved means of testing and, in some cases, verifying the correctness of control system designs.

In many domains, embedded control software has been increasing in scale and complexity for years, and this trend is expected to continue for the foreseeable future. For example, the software systems in a premium automobile may contain 100 million lines of code distributed across

Digital Object Identifier 10.1109/MCS.2016.2602089
Date of publication: 11 November 2016

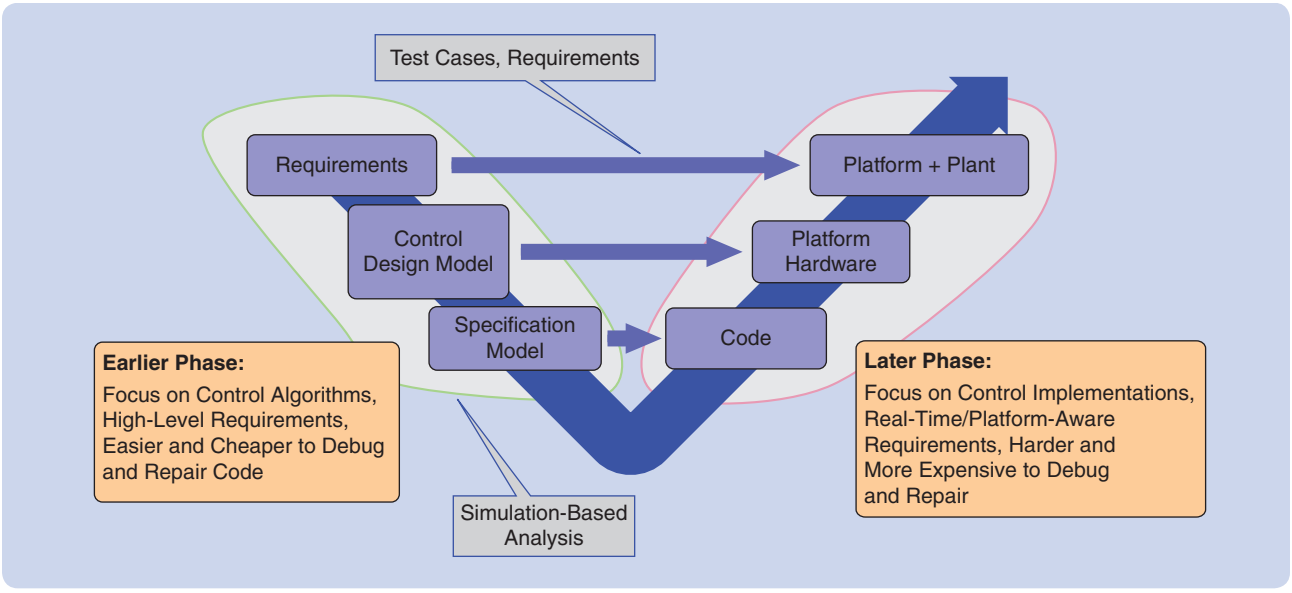


FIGURE 1 The model-based development design V. This process is used to develop reliable embedded control systems.

dozens of microprocessors [1]. Code complexity continues to increase for many reasons. One reason is the increasing level of autonomy for smart vehicles, such as the NASA Mars rovers, unmanned aerial vehicles, and self-driving automobiles. Increased autonomy is often achieved by using advanced algorithms that increase the complexity of the control software.

Another reason for increasing code complexity is the need to respond to ever-increasing government-mandated regulatory requirements, as exemplified by the vehicle efficiency, emissions, and diagnostics standards in the automotive domain. For example, the corporate average fuel economy (CAFE) standard [2], which is a U.S. government regulation that defines requirements based on the total vehicles an automaker produces, mandates thresholds for fuel efficiency and emissions. CAFE 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, at the cost of adding a new physical system that must be regulated by the electronic control unit, which increases the control software complexity.

Traditional software development processes for embedded control systems involve manually generating code in a monolithic manner and then validating the system design with experimental tests. This approach is expensive and difficult to manage for complex systems, and it results in inflexible controller designs, which are difficult to reconfigure since they are not inherently modular (that is, there are not clear separations between software components). This lack of flexibility is problematic for complex system development, where system requirements and plant parameters often evolve during the development process.

To manage the complexity, many organizations adopt a model-based development (MBD) approach, which is a process for developing embedded control systems based on models that represent the dynamic behavior of the system. The goal of the MBD process is to provide a unified framework for creating, documenting, testing, and deploying reliable embedded control systems.

The MBD process is illustrated in Figure 1. The process begins with performance requirements that define how the system should behave. Based on requirements, a control design model is created. The control design model is refined to include implementation details, such as controller sampling and saturation, resulting in a specification model. Code (software) is generated (either manually or automatically) based on the specification model. The code is then compiled for the platform hardware, which is ultimately deployed in a system that includes a real-time hardware platform interacting with a plant (a physical environment).

At each vertical level of the design V, the requirements or models that appear on the left side of the V define the behaviors that should be exhibited by the corresponding systems on the right side of the V. This mapping of specifications to desired behaviors (indicated by the horizontal lines in Figure 1) provides traceability for the performance specifications, meaning that there is a direct correlation between the design requirements or models and the corresponding system under test.

The earlier stages of the development are associated with the left side of the design V. Identifying a problem with the control design at these early stages results in less expensive rework than if the problem is identified later in the development process. In early development stages, simulation-based approaches to verification are valuable because they

offer a means to identify design problems using the models that appear on the left side of the design V.

Many techniques are used to debug and verify software for embedded control systems. Approaches can broadly be classified in terms of how well they account for the possible behaviors of the model and how well they scale. Some techniques, such as model checking, can provide formal guarantees of correctness for all behaviors of software systems, but these do not scale well for many industrial embedded control systems. Simulation, on the other hand, can be applied to models of any scale but only provides an approximation of behavior for a discrete set of operating conditions. For a discussion of the range of analysis techniques for embedded control systems, see "Spectrum of Analysis Techniques."

Simulations provide numerical approximations of system behavior, given a mathematical model of the system, and are commonly used for debugging embedded control system designs. Simulations are used to 1) validate functional behavior, 2) obtain initial calibration parameter values, 3) obtain estimates of system performance, and 4) serve as the basis for the functional and software specifications.

Currently, simulations are often used in an ad hoc manner to check for design bugs. Engineering intuition is used to select operating conditions to demonstrate the desired behavior; however, emerging techniques are available to automatically select critical operating conditions for the purposes of verification and test.

This article presents an overview of traditional and advanced modeling, testing, and verification techniques used in the development of embedded control systems. The article begins by introducing standard techniques and tools used in industry to develop and test embedded control systems. Next, emerging advanced testing approaches are presented, followed by advanced verification techniques. The article concludes with a summary of the available testing and verification approaches.

PRELIMINARIES

General testing and verification scenarios involve a system $\mathcal{M}$, a (possibly infinite) set of parameters P, a (possibly infinite) set of inputs U, and a property ψ that should hold for the system. 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 $p \in P$, $u \in U$, and u is (generally speaking) a function of time. The behavior of system $\mathcal{M}$ under parameter p and input u is denoted $\Phi(\mathcal{M},p,u)$, and $\Phi(\mathcal{M},P,U)$ is the set of all possible behaviors of $\mathcal{M}$ under the parameters in P and inputs in U. Behaviors can be obtained either from experiments, where behaviors are observed based on sensor measurements, or from *simulations*, where behaviors are estimated using numerical methods.

Assume that $\mathcal{M}$ can be evaluated to determine whether ψ holds for a particular p and u. The notation $\Phi(\mathcal{M},p,u) \models \psi$ is used to denote that $\Phi(\mathcal{M},p,u)$ satisfies ψ; conversely, $\Phi(\mathcal{M},p,u) \not\models \psi$ denotes that $\Phi(\mathcal{M},p,u)$ does not satisfy ψ. When the sets P and U are clear from the context, then $\mathcal{M} \models \psi$ is used to mean that the (possibly infinite) behaviors in $\Phi(\mathcal{M},P,U)$ satisfy ψ, and $\mathcal{M} \not\models \psi$ indicates that not all behaviors in $\Phi(\mathcal{M},P,U)$ satisfy ψ.

The following definitions provide the testing and verification activities that are addressed herein.

Definition 1 (Testing)

The testing task is to determine whether $\Phi(\mathcal{M},\hat{P},\hat{U}) \models \psi$ for given sets $\hat{P} \subseteq P$ and $\hat{U} \subseteq U$, where $\hat{P}$ and $\hat{U}$ are finite.

Testing is the most common means of evaluating embedded control system designs, but it has two significant limitations. First, testing conditions may not accurately reflect the manner in which the system will be used once it is deployed. For example, testing an engine inside of a test cell is different than driving the vehicle on a busy highway. Second, note that sets $\hat{P}$ and $\hat{U}$ are finite in Definition 1, which implies that testing cannot be used to exhaustively evaluate continuous parameter or input 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 verify whether ψ holds for all behaviors of $\mathcal{M}$. Mathematically speaking, no matter how many tests are performed, the system remains, *almost everywhere,* untested. Verification, on the other hand, addresses this problem.

Definition 2 (Verification)

The verification task is to prove $\Phi(\mathcal{M},P,U) \models \psi$ 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 perform verification for software systems (see "Formal Methods" for further details). Some of these tools can be applied to embedded control systems, but no mature tools exist that can be applied to detailed industrial models that capture plant behaviors, such as engine dynamics. If proving correctness is not possible, another approach is to assume that a bug exists and then employ a technique to actively search for the incorrect behavior.

Definition 3 (Falsification)

The falsification problem is to find a $p \in P$ and $u \in U$ such that $\Phi(\mathcal{M},p,u) \not\models \psi$.

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 $\Phi(\mathcal{M},p,u) \not\models \psi$.

Also, it is interesting to note that, from a logical standpoint, verification is equivalent to determining whether the

Spectrum of Analysis Techniques

Many types of analyses can be performed on embedded control system designs. Each analysis approach has unique benefits and shortcomings, and each applies to a specific class of system representations.

Consider the spectrum of analysis techniques presented in Figure S1, which provides a subjective classification of various analysis approaches, based on the degree of *exhaustiveness* of the approach and the *scale* of the model to which the approach can be applied. Here, exhaustiveness refers to how well the approach accounts for all possible behaviors of a model. The exhaustiveness is indicated by the horizontal position of each approach (left is less exhaustive and right is more exhaustive). The scale of each model refers to the level of detail and size of the models that can effectively be addressed by each approach. The scale is indicated by the vertical position of each approach (lower is smaller scale and higher is larger scale).

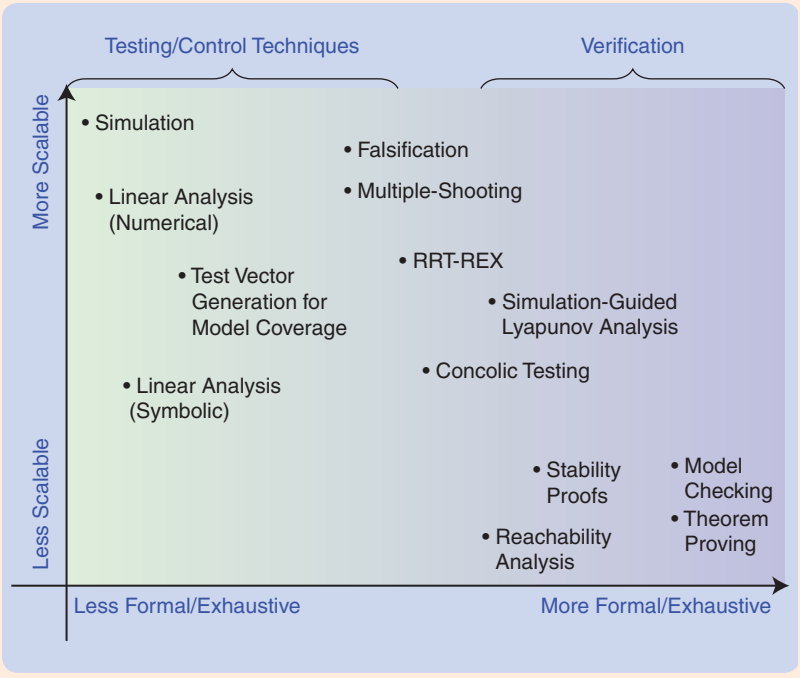


FIGURE S1 The spectrum of analysis techniques. For various types of analyses, the spectrum illustrates how thoroughly each one accounts for system behaviors and the level of complexity of the models that can be considered.

The analysis techniques on the far left side of Figure S1 are classified as "testing/control techniques," since they are based on individual (finite) sets of behaviors of the system model or provide information about only local behaviors. The analysis techniques on the right side fall under the classification of "verification" techniques, since they account for all behaviors of the system models.

Consider the simulation item in Figure S1, which is intended to refer to approaches that use simulations based on operating conditions that are either manually selected or are selected using a Monte Carlo method. This item is located at the top-left of the spectrum because it can be performed for models of any scale but provides only one example of the system behavior. Therefore, simulation scales well, but it does not provide exhaustive results.

Two different types of linear analysis appear on the spectrum, numerical and symbolic. Here, linear analysis refers to the process of applying Lyapunov's indirect (first) method to

system can be falsified—that is, whether there exists a $p \in P$ and $u \in U$ such that $\Phi(\mathcal{M},p,u) \not\models \psi$. An important consequence of falsification is that a specific $p \in P$ and $u \in U$ that demonstrates that $\Phi(\mathcal{M},p,u) \not\models \psi$ is identified. This parameter and input provide the user with valuable information that can be used to debug the design.

All testing and verification approaches rely on some form of requirements, either formal or informal, but the process of creating correct and useful requirements is an often underappreciated activity. Care should be taken to create requirements that accurately reflect the intended behavior of the system.

Definition 4 (Requirement Engineering)

Requirement engineering is the process of developing an appropriate ψ.

Requirement engineering remains a challenge for industry. Embedded control developers in many domains have made significant efforts to generate and document clear and concise requirements; however, challenges remain due to 1) the incompatibility between the form of the documented requirements and the input to existing verification and testing tools, 2) the ambiguous nature of requirements captured in natural language, 3) potential inconsistencies between requirements, and 4) the large number of requirements.

QUALITY CHECKING FOR EMBEDDED CONTROL SYSTEMS

This section presents an overview of modeling and simulation techniques currently used in industry. Generally speaking, modeling is the process of developing an appropriate

determine stability. Symbolic linear analysis uses partial derivatives of an analytic representation of the system dynamics to obtain the linearized dynamics. Numerical linear analysis uses numerical perturbation to obtain the linearized dynamics. The two methods are located on the left side of the spectrum because they only provide an indication of system model behaviors *locally*, that is, in a neighborhood of the point of linearization. The linear analysis items are located to the right of simulation because they provide results that apply to an infinite collection of behaviors (local to the point of linearization).

Test-vector generation (TVG) refers to automated processes for creating system inputs such that some coverage criteria are satisfied. TVG is located to the right of the linear analysis items because it is expected to explore a wide range of system behaviors, albeit using a finite collection of simulation traces. TVG is located near the middle of the scalability dimension because, although TVG techniques may be applied to any system for which simulation may be applied, it may not successfully achieve the desired level of coverage for large models.

Concolic testing refers to techniques that use concrete executions (simulations) of software systems combined with formal analysis of decision branching conditions to satisfy some coverage criteria. This approach is placed near the center of the spectrum because it is moderately exhaustive (it can cover many decision branches but only those associated with a fixed set of executions), and it can be applied to software models of moderate size, since the simulations can be performed on any model, but the computational cost of the branch decision analysis is prohibitive (particularly if a plant model is considered). Note that there are several different ways that the general concolic testing approach may be applied, so arguments could be made to move the location to some other region of the spectrum.

Stability proofs refers to the process of applying Lyapunov's direct (second) method to determine stability. This is placed on the right side of the spectrum because it can be used to prove properties for all behaviors of the system (for example, when the system is globally stable). Stability proofs are placed low on the scalability axis because the proofs must be constructed manually and so are difficult to apply to large-scale system models.

Reachability analysis refers to techniques that use numerical methods to conservatively approximate the set of behaviors that a closed-loop system model can exhibit. This approach is placed on the right side of the spectrum because, generally, it can provide a guarantee of correctness for all system behaviors; however, reachability analysis is not located on the extreme right because, for closed-loop system models, it may not provide exhaustive results over unbounded time. Reachability analysis is placed low on the scalability axis because it is computationally expensive and does not scale well with the complexity of the model, particularly when a plant model is considered.

Model checking and theorem proving refer to formal analysis techniques for strictly software system (open-loop) models that can provide a proof that all model behaviors satisfy a given logical property, often expressed in temporal logic. These approaches provide entirely exhaustive results for models but are computationally expensive and cannot be applied to detailed software models or to plant models of even moderate complexity. Also, some theorem provers can handle close-loop models, but these tools require significant user intervention.

The falsification, multiple-shooting, RRT-REX, and simulation-guided Lyapunov analysis techniques, which are detailed in the article, appear near the center of the spectrum. The central horizontal location is selected because these approaches automatically select operating conditions to produce simulations that explore the space of system behaviors as widely as possible. The central-to-high vertical locations are selected because these approaches can be applied to closed-loop dynamic system models of moderate-to-high complexity.

system model $\mathcal{M}$. Simulation is the process of obtaining particular behaviors $\Phi(\mathcal{M}, p, u)$.

Modeling Paradigms Used in Embedded Control Applications

Simulation-guided approaches to verification and testing require a model $\mathcal{M}$ of the system under consideration. Here, $\mathcal{M}$ can contain representations of the embedded controller, the plant, or a *closed-loop* model, which contains both. Models of the controller can range from simple, continuous-time representations (such as 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 a real-time platform.

Developing a plant model can pose a significant challenge for complex systems. Complex interactions of mechanical, hydraulic, thermal, electrical, and chemical phenomena make the problem of capturing the dynamical system behavior difficult, particularly when under a tight development schedule. In practice, a modeling paradigm is chosen based on the usage scenario for the model.

Causal, lumped-parameter models are a common type used to capture plant dynamics. Causal models have a clear distinction between the source and destination of signals that induce system behaviors. Lumped-parameter models use a discrete set of descriptive components to simplify the mathematical representation of phenomena that are continuously distributed over a physical region. In causal, lumped-parameter models, system dynamics are

Formal Methods

Formal methods (FMs) are used in some software and hardware development domains to verify properties of computer code, such as C programs, or finite-state logical behaviors, such as field programmable gate arrays. Model checking is an FM technique that takes a model of the system (such as 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 [S1]. Model checking was first applied to systems such as logic circuits and later to computer software [S2]. Several model-checking tools have been developed and successfully applied to verify communication protocols [S3], hardware drivers [S4], and even focused components of automotive control code [S5]. Model checkers such as SLAM [S2] and CBMC [S6] are used by industry to verify system correctness.

Theorem proof assist tools, referred to as *theorem provers*, are based on FM techniques for verifying system correctness. While model checkers rely on an evaluation of 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 [S7], Coq [S8], Isabelle/HOL [S9], and ACL2 [S10] have been used to verify software correctness. Recent work has combined the rigor of theorem proving using Isabelle/HOL with the performance and automation of set-based reachability for the purpose of verification for continuous dynamical systems [S11]. The KeYmaera tool can be used to prove properties about hybrid systems [S12], and recent extensions have allowed information obtained from simulations to be used to assist with the proof task [S13]. Though theorem provers can be used to provide a formal proof of correctness, the tools require user intervention and can be difficult for nonexperts to use.

STATIC ANALYSIS

Static analysis is another way to debug and check the quality of control code for embedded control systems. A key feature of static analyzers is that they operate directly on source code (or models) and do not need to evaluate system behaviors to check for problems in the design. Static code analyzers have become standard in most integrated development environments [S14], but most analyzers can only check for specific types of coding errors, such as variable-type mismatches. A notable exception is the Astrée tools, which can prove the absence of run-time errors in C programs and has been used to perform formal analysis of primary flight control software in the Airbus A340 airliner [S15].

FORMAL METHODS CHALLENGES

FM techniques provide a proof of system correctness but suffer from fundamental and practical drawbacks. Fundamentally, FMs do not scale well for large, industrial systems. On the practical side, FMs 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; this challenge is common to other analysis techniques, including simulation-based approaches. Also, many tools require that an intermediate model be created, based on the original system model. The task of creating intermediate models is often performed manually and so is time consuming and prone to error.

REFERENCES

[S1] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.,* vol. 16, no. 5, pp. 1512–1542, 1994.

[S2] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with slam," *Commun. ACM,* vol. 54, no. 7, pp. 68–76, July 2011.

[S3] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas, "PVS: Combining specification, proof checking, and model checking," in *Computer Aided Verification,* R. Alur and T. A. Henzinger, Eds. New York: Springer, 1996, pp. 411–414.

[S4] T. A Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *Model Checking Software,* T. Ball and S. K. Rajamani, Eds. New York: Springer, 2003, pp. 235–239.

[S5] T. Kaga, M. Adachi, I. Hosotani, and M. Konishi, "Validation of control software specification using design interests extraction and model checking," in *Proc.* SAE *World Congr. and Exhibition*, 2012.

[S6] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems* (Lecture Notes in Computer Science, vol. 2988), J. Kurt and A. Podelski, Eds. Berlin, Germany: Springer, 2004, pp. 168–176.

[S7] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)* (Lecture Notes in Artificial Intelligence, vol. 607), D. Kapur, Ed. Saratoga, NY: Springer-Verlag, 1992, pp. 748–752.

[S8] Y. Bertot and P. Cast'eran. *Interactive Theorem Proving and Program Development*. New York: Springer, 2004.

[S9] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Berlin, Germany: Springer-Verlag, 2002.

[S10] M. Kaufmann, J. Strother Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Norwell, MA: Kluwer Academic, 2000.

[S11] F. Immler, "Verified reachability analysis of continuous systems," in *Proc. 21st Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, 2015, pp. 37–51.

[S12] A. Platzer and J.-D. Quesel, "KeYmaera: A hybrid theorem prover for hybrid systems," in. *IJCAR,* (Lecture Notes in Computer Science, vol. 5195), A. Armando, P. Baumgartner, and G. Dowek, Eds. New York: Springer, 2008, pp. 171–178.

[S13] N. Arechiga, J. Kapinski, J. Deshmukh, A. Platzer, and B. Krogh, "Forward invariant cuts to simplify proofs for safety," in *Proc. Int. Conf. Embedded Software*, Amsterdam, The Netherlands, 2015, pp. 227–236.

[S14] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools. Electronic notes in theoretical computer science," in *Proc. 3rd Int. Workshop on Systems Software Verification*, 2008, vol. 217, pp. 5–21.

[S15] D. Delmas and J. Souyris, "Astrée: From research to industry," in *Static Analysis,* (Lecture Notes in Computer Science, vol. 4364), H. Riis Nielson and G. Filé, Eds. Berlin, Germany: Springer, 2007, pp. 437–451.

described by ordinary differential equations (ODEs) and can be modeled using block diagrams connected by edges that represent paths for unidirectional signal flow. This type of model can be created in, for example, Simulink or Ptolemy II [3].

Figure 2(a) provides an example of a block diagram representing a causal model. The system in the figure represents the ODEs describing a spring-mass-damper system,

$$\dot{x}_1(t) = x_2(t),$$
$$\dot{x}_2(t) = g - \frac{k}{M}x_1(t) - \frac{b}{M}x_2(t),$$

where $x_1(t)$ and $x_2(t)$ are the position and velocity of the mass, respectively, k is the spring constant, b is the damping constant, M is the mass, and g is the acceleration due to gravity. The integrators require initial conditions, $x_1(0)$ and $x_2(0)$, which determine the initial configuration of the system. The arrows in the diagram indicate that a signal value emanates from one block and is used as input to another. For example, the state of the integrator $x_1(t)$ provides a value to the k/M multiplier block, which the multiplier 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. 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 or an environment that supports the Modelica language (such as Dymola or MapleSim) or the VHDL-AMS language (for example, Simplorer).

Figure 2(b) is a block diagram representing an acausal model of the spring-mass-damper system described previously. 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 (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 more computation time for 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 have traditionally been used in control design. However, this is beginning to change, due to the increasing complexity of the systems under development.

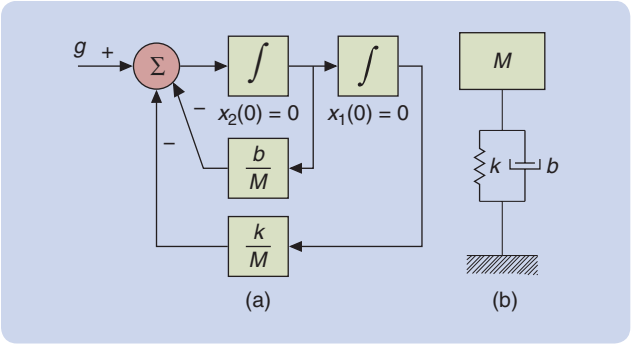


FIGURE 2 Examples of dynamic models: (a) an example of a causal model and (b) an example of an acausal model.

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 that are required to define its behavior over time are spread across a relatively wide physical area.

PDEs are used sparingly in embedded control design because producing simulations for PDEs is computationally expensive. As an example, consider the dynamics of an automotive engine after-treatment system, which is responsible for reducing the amount of toxic pollutants emitted by the vehicle. For some analyses, it is necessary to accurately model the distribution of heat along the length of the catalyst (a critical component of the after-treatment system), which is best represented with 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 spatial 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 (see [4], for example).

Finite-element analysis (FEA) models are used to capture physical phenomena best described by boundary-value problems defined over some spatial distribution, for example, electromagnetic fields, temperature variation, stress/strain, and 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, FEA is seldom used to model embedded control systems.

Simulation and Testing for Embedded Control 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 collection of operating conditions given by finite

sets $\hat{P}$ 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 plant. Simulink, Dymola, and Simplorer are commonly used tools for this type of activity. Engineers use simulation to perform preliminary tuning of control parameters, estimate performance of a given control design, and also debug the design.

Control design simulation has parallels in the program-analysis domain. Some program testing standards require that each decision path in the control code be exercised using some testing approach, for example, testing requirements based on the modified condition/decision coverage (MCDC) criterion [5]. 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 of behaviors. Discrepancies inevitably exist between simulations and behaviors of the corresponding embedded control system due not only to parameter-estimation error and modeling simplifications but also to numerical computations involved in estimating the solutions to differential equations.

Software-Centric Versus System-Centric Perspectives

Either the software-centric perspective or the system-centric perspective can be taken when using simulations to check embedded control designs. The software-centric perspective assumes that all of the correct behaviors of the system are formally defined and captured by the requirements. The system-centric perspective respects that the requirements may not entirely characterize the correct system behavior, due to the unique challenges presented by embedded control systems.

Figure 3 illustrates the way that simulations are used to test control designs using a typical software-centric perspective. A model is created manually by a designer based on requirements. Simulation-based checks are performed, and the resulting behaviors are checked against the requirements. If any of the behaviors explored through simulation violate the requirements, then the control design model is enhanced to eliminate the violating behavior. Once a user-defined number of simulations are found to satisfy the requirements, the model is used to proceed with the next development phase. This process could be used, for example, when validating the model represented by the control design model block on the left side of the design V in Figure 1.

Although it is used for some embedded control-system designs, the simulation-based validation process illustrated in Figure 3 does not take into consideration the unique challenges presented by embedded control systems. Specifically, Figure 3 does not account for the inability to create a set of requirements that captures all intended behaviors. This deficiency is particularly apparent for *cyberphysical* systems, which are systems whose performance critically depends on the plant behavior.

There are two main reasons why it is particularly difficult to create formal requirements for cyberphysical systems. First, it is not always possible for the design engineers to predict all the ways in which the physical and environmental components will interact with each other. Indeed, it is not always possible to even predict the existence of some interactions. Without a priori knowledge about all possible component interactions, it is difficult to create requirements to cover all behaviors that can emerge as a result of the interactions. The second reason is that there are some qualitative system behaviors that are difficult to capture with formal requirements. For example, consider a system that the designer expects to behave in a manner generally consistent with a second-order linear system (a second-order decaying exponential). This qualitative feature may be expected by the engineer, and failure to achieve this qualitative characteristic may indicate incorrect behavior. While commonly used performance indicators such as overshoot and settling time could be formally characterized, other qualitative aspects of the expected behavior such as the *smoothness* or the *near sinusoidal* behavior would be difficult to characterize with prevalent requirement formalisms used by verification tools.

Figure 4 illustrates a simulation-based model testing process that respects the challenges unique to cyberphysical systems. The process is similar to the one illustrated in Figure 3, with some key differences. Some formal requirements are included, but also expected behaviors exist in the form of engineering insight. The engineering insight can be provided by the system architect as well as the model designer, which can be used to create or refine the model. Results from simulations that violate either the formal requirements or the engineering insight trigger a redesign of the model and also enhance the engineering insight. Further, results from integration tests, which occur on the right side of the design V shown in Figure 1, can provide valuable feedback

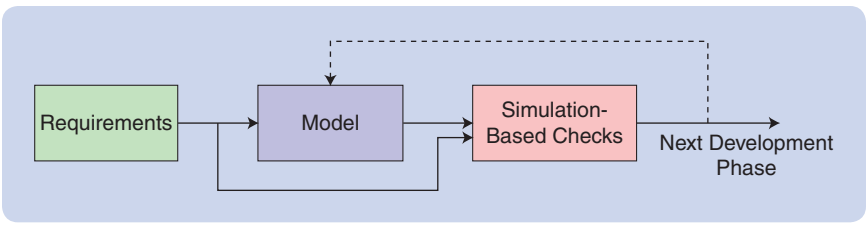


FIGURE 3 The software-centric view of an embedded control design testing process. This process assumes that intended behaviors are thoroughly captured by the requirements.

(about, for example, previously unknown physical and environmental component interactions) and can be used to enhance both the engineering insight as well as the formal requirements.

Take, for example, the spring-mass-damper system shown in Figure 2(a) and (b). Although this system contains only a plant (no controller), the process in Figure 4 could be used to validate the model. Consider formal requirements that specify acceptable maximum settling time and overshoot for the system, which would correspond to requirements, and also consider the informal expectation from the designer that the system behaves in a typical linear manner, which would correspond to engineering insight. The model contains a representation of the system and is intended to satisfy the formal and informal requirements. Simulations performed at the simulation-based checks stage indicate whether the modeled system meets the formal and informal requirements; if not, then the model is refined. Once the model is found to satisfy the informal and formal requirements, it is used as a basis for the next stage of the development process. Eventually, a physical system is implemented, and integration testing results are made available. If the results from the tests indicate that either the formal or informal requirements did not sufficiently capture the intended behaviors, then the engineering insight and requirements are updated accordingly. For example, it may be that the desired time constant and overshoot are not physically realizable due to unmodeled nonlinearities in the spring behaviors. These testing results, which indicate that the system does not satisfy the behavior specified by the requirements, trigger an iteration in the development process, whereby the engineering insight and requirements are updated and the model is refined, tested, and used in the subsequent iteration of the development process.

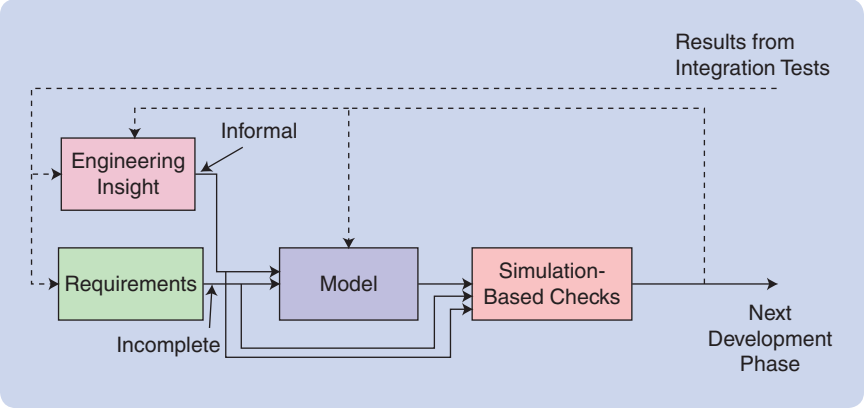


FIGURE 4 A system-centric view of an embedded control design testing process. This perspective respects the challenges unique to cyberphysical systems.

The process shown in Figure 4 and described above can be used, for example, to create and validate the model represented by the control design model block on the left side of the design V in Figure 1. Iterations in the design V process based on integration testing results that are shown to be inconsistent with design requirements are costly. Because of this, care should be taken to create thorough, consistent, and realistic requirements and accurate system models to avoid the cost associated with excessive iterations in the development process.

One challenge in applying any simulation-based testing process is the difficulty in accounting for inaccuracies that appear in the plant models. Inevitably, discrepancies will exist 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, broadly speaking, physical phenomena are not modeled exactly. Physical processes are sometimes neglected entirely, but even for the most detailed environment models, some behaviors and interactions between system components are not modeled (sometimes referred to as second- or third-order effects) because they are assumed to be noncritical for capturing the intended behaviors of the control system. Techniques from robust control design, such as H-infinity control, can be used to design controllers to account for modeling inaccuracies for some systems, but these techniques are difficult to apply to industrial problems that involve nonlinear plant dynamics and other complexities such as actuator saturation and computation delays [6].

Testing Scenarios

Traditional testing approaches use ad hoc techniques for selecting test cases. In ad hoc testing, a control engineer manually selects some inputs to the system (for example, a step input) and a set of operating parameters (for example, proportional and integral controller gains and initial conditions on the state variables). The selection of such test inputs and parameters is usually based on the engineer's experience and insights about which inputs represent nominal operating behavior, worst-case behavior, and so on. With the increasing complexity of embedded control systems, methods, such as ad hoc techniques, that rely on engineering insights are no longer scalable and are thus being replaced by automated testing approaches.

Current testing approaches for embedded control systems involve various manifestations of the control system, from computer models to experiments on the physical system. The testing approaches described herein are easiest to apply to computer models (as in the model-in-the-loop testing scenario described below), but it is feasible to apply them to many different testing scenarios. The next sections

describe open- and closed-loop testing scenarios commonly used in industry.

Open-Loop Testing

Typically, the first step in testing a controller is open-loop testing to validate that it meets its functional requirements. In open-loop testing based on code-coverage metrics, the plant model is neglected, and the controller model is tested as though it were a computer program. The goal of the testing process is to automatically select inputs to the controller model that maximize a software code-coverage metric. There are several code-coverage metrics, such as MCDC [5], which is the most popular in the automotive domain. Tools such as Reactis, Simulink Design Verifier (SLDV), 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 [7].

SLDV uses SAT-solving techniques provided by the Prover tool to automatically generate test inputs to maximize coverage criteria [8], [9]. SLDV is intended for open-loop (discrete-time) controller models since it cannot process closed-loop (hybrid) models.

Closed-Loop Testing

There are several commonly used closed-loop testing approaches. The test scenarios are presented below 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 plant, 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*: In this testing scenario, $\mathcal{M}$ is a computer model composed of a representation of the plant interacting with a controller that 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 that is running a computer model of the plant. In this case, the communication between the plant and the controller uses a direct communication link, such as an Ethernet connection or a controller area network bus, and the system is not run in real time but, rather, uses a synchronization mechanism to synchronize the controller with the PC running the plant simulation.
- *Hardware-in-the-loop*: In this testing scenario, $\mathcal{M}$ is composed of the real-time platform and a virtual plant, which could be a computer model of the plant running in real time on specialized hardware or a combination of a computer model and physical components connected electronically. 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.
- *Integration and calibration*: In this scenario, all subsystems are connected together with the actual plant to tune the control parameters and validate the performance of the closed-loop system.

The TestWeaver tool by QTronic uses simulations of closed-loop models to attempt to obtain a high degree of coverage and also to violate system requirements [10]. TestWeaver uses a search algorithm that is based on proprietary heuristics. The tool 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.

Verification for Embedded Control 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. Broadly speaking, verification can be performed using *formal methods*, which are a rich set of concepts and techniques; for further details, see "Formal Methods."

Testing and verification are closely related, with one main difference. While testing determines whether $\Phi(\mathcal{M}, \hat{P}, \hat{U}) \vDash \psi$ for some finite $\hat{P}$ and $\hat{U}$, verification determines whether $\mathcal{M} \vDash \psi$ over the infinite set of parameters and inputs 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 ψ, a specification is required to perform verification. A specification ψ for a verification tool is usually supplied in the form of a special language such as 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, can express timed operators over fixed time ranges [11], such as

$$\psi \equiv \square_{[1.0,2.0]} x(t) < 10.0, \tag{1}$$

which means the signal x must remain lower than 10.0 between the times $t = 1.0$ and $t = 2.0$. For further details, see "Temporal Logic." Any verification procedure will return either of the following:

- *Verified*: This result is returned if the procedure determines that ψ holds for all of the cases. When a procedure returns a *verified* result only if $\Phi(\mathcal{M}, P, U) \vDash \psi$, the procedure is called *sound*. The

Temporal Logic

In the late 1970s, Amir Pnueli [S16] introduced temporal logic to computer science to reason formally about the temporal behaviors of reactive systems, which are systems that are designed to continually interact with an environment. This work was recognized with the 1996 ACM A.M. Turing Award, one of the highest honors for a researcher in computer science. The use of temporal logic was originally to reason about input–output systems with Boolean, discrete-time signals, and heavily focused on the verification, specification, and synthesis of concurrent systems. Several temporal logics were introduced to reason about real-time signals, including timed propositional temporal logic [S17] and metric temporal logic (MTL) [S18]. These logics typically allowed reasoning over Boolean signals but over dense-time domains. More recently, signal temporal logic (STL) [S19] was proposed in the context of analog and mixed-signal circuits as a specification language for constraints on real-valued signals. Syntactically, an STL formula is defined recursively. The basic unit of an STL formula is an atomic formula that expresses constraints on signals, and any formula is defined using the negation of a subformula or using Boolean combinations (conjunctions, disjunctions) of subformulas, or using temporal operators applied to subformulas. Atomic formulas, without loss of generality, can be reduced to a form $f(\mathbf{x}) \bowtie 0$, where $\mathbf{x}$ represents the name of signal (a function from $\mathbb{R}^{\geq 0}$ to $\mathbb{R}^n$), $\bowtie \in \{<, \leq, >, \geq, =\}$, and f is an arbitrary function from $\mathbb{R}^n$ to $\mathbb{R}$. A temporal formula is formed using temporal operators "always" (denoted as $\Box$), "eventually" (denoted as $\Diamond$) and "until" (denoted as $\mathbf{U}$). Each temporal operator is indexed by an interval I over $R^{\geq 0} \cup \{\infty\}$; this can be an open interval (a,b), a closed interval $[a,b]$, open-closed $(a,b]$ or closed-open $[a,b)$. Several example STL formulas are

$$\psi_1 \equiv \Box_{[0,100]}(\texttt{boost\_pressure\_error} < c_1), \tag{S1}$$

$$\psi_2 \equiv \Box_{[0,10]}(\texttt{rising\_edge} \Rightarrow \Diamond_{[0,2]}|\mathrm{y}| < 0.1), \tag{S2}$$

$$\psi_3 \equiv \Box_{[0,100]}(\texttt{gear} = 1 \wedge \Diamond_{[0,\epsilon]}(\texttt{gear} = 2 \wedge \Diamond_{[\tau,\tau+\epsilon]}\texttt{gear} = 1)), \tag{S3}$$

$$\psi_4 \equiv \Box_{[0,\infty)}\left(\begin{matrix}\texttt{throttle} = 0 \Rightarrow (\texttt{fuel\_cut} = \texttt{on}) \\ \mathbf{U}((N_e \leq 650) \wedge \Diamond_{[0,1]}(\texttt{fuel\_cut} = \texttt{off}))\end{matrix}\right). \tag{S4}$$

The requirement ψ_1 in (S1) specifies that for all times t in $[0, 100]$, the physical quantity `boost_pressure_error` is always lower than c_1 kPa. This STL requirement can be used to characterize maximum allowed overshoot (or undershoot). The requirement ψ_2 (S2) specifies that for all times t in [0, 10], whenever the Boolean proposition `rising_edge` is *true*, then eventually within 2 s (that is, within $[t, t+2]$), the absolute value of y is lower than 0.1. This requirement can be used to capture settling behavior of a signal, and the time bound on the inner temporal operator ($\Diamond$) captures settling time. The requirement ψ_3 (S3) specifies that if the *gear* changes from one to two within a small time (ϵ), then it stays at two for at least τ s before changing back to one. Such a requirement can be used to specify the dwell time on a discrete mode of the system. Finally, the requirement ψ_4 (S4) specifies a causal behavior of the system. Requirement ψ_4 says that if the `throttle` angle is zero, the `fuel_cut` mode must remain `on` until the engine speed (N_e) drops below 650 r/min, and then the `fuel_cut` mode must turn `off`.

The syntax for the logic MTL is similar to STL. The only difference is that MTL requires that formulas be defined over Boolean signals; continuous-valued signals can be considered by converting them to Boolean signals based on given logical predicates over the continuous signals. A key feature of MTL and STL is that both logics are equipped with quantitative semantics, which is a function mapping a given signal trace and an STL/MTL formula ψ to a real value [S20], [S21]. This value is an indicator of the degree of satisfaction of ψ; positive values indicate that the trace satisfies ψ, negative values denote violation of ψ, and the magnitude indicates the robustness margin. In other words, a positive value δ indicates that the signal can be perturbed by up to δ before it violates ψ. STL and MTL differ in how they define the signed distance of a signal trace from an atomic predicate, which impacts the computational complexity of the quantitative semantics for these logics.

REFERENCES

[S16] A. Pnueli, "The temporal logic of programs," in *Proc. Symp. Foundations of Computer Science*, 1977, pp. 46–57.

[S17] R. Alur and T. A. Henzinger, "A really temporal logic," in *Proc. Symp. Foundations of Computer Science*, 1989, pp. 164–169.

[S18] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Syst.*, vol. 2, no. 4, pp. 255–299, 1990.

[S19] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Proc. Formal Modeling and Analysis of Timed Systems Conf.*, 2004, pp. 152–166.

[S20] G. E. Fainekos and G. J. Pappas, "Robustness of temporal logic specifications for continuous-time signals," *Theoretical Comp. Sci.*, vol. 410, no. 42, pp. 4262–4291, 2009.

[S21] A. Donzé and O. Maler, "Robust satisfaction of temporal logic over real-valued signals," in *Proc. Formal Modeling and Analysis of Timed Systems Conf.*, 2010, pp. 92–106.

capacity of a technique to return a sound result is called *soundness*.

» *Not Verified*: This result is returned if the procedure cannot certify that ψ holds for all cases. In some instances a *counterexample* is also returned, which is a concrete p and u that demonstrates that $\Phi(\mathcal{M}, P, U) \not\models \psi$. The procedure may return Not Verified even when $\Phi(\mathcal{M}, P, U) \models \psi$, because the underlying technique overestimates the cases where $\Phi(\mathcal{M}, p, u) \not\models \psi$.

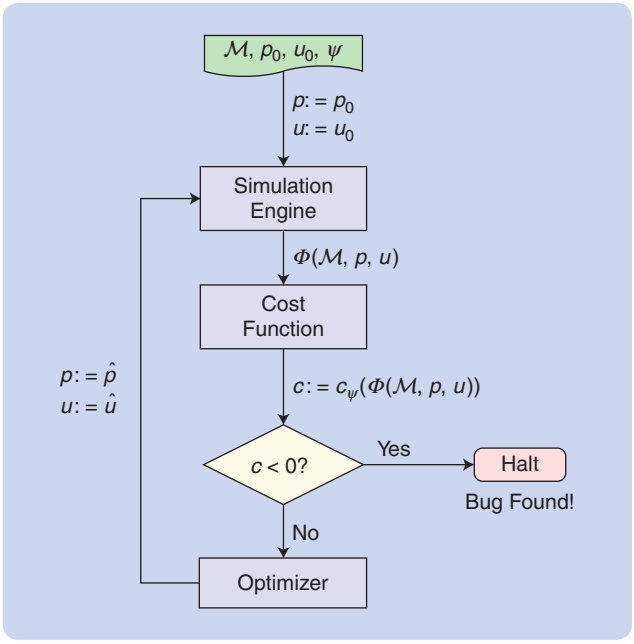


FIGURE 5 Optimization-guided falsification. This procedure is used to automatically search for behaviors in a control system design model that violate requirements.

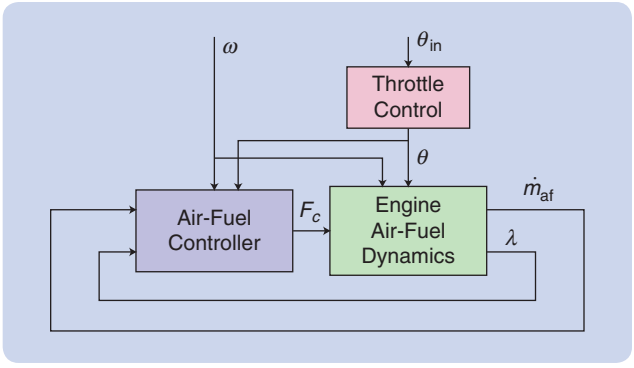


FIGURE 6 Overview of an automotive air-fuel control system example.

Problems Applying Verification Approaches

Applying verification techniques to embedded control systems is a challenging task. These systems can often 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* [12]. This undecidability result means that it is provable that no computer algorithm can decide whether any arbitrary hybrid system satisfies any given formal specification.

Tools exist for verifying specific subclasses of hybrid systems, but each suffers from significant limitations. SpaceEx verifies hybrid systems with affine continuous dynamics and polyhedral switching constraints [13]. The UPAAL tool can verify complex hybrid systems, but it is limited to *timed automata*, which are systems with continuous dynamics defined by derivatives fixed at 1.0 [14]. The Flow* tool [15] handles systems given by ODEs that are expressed as polynomial functions of the state variables. The C2E2 tool can verify safety properties of hybrid systems but only if the designer provides sufficient annotations to the models in the form of discrepancy functions, which characterize the maximum rate at which pairs of trajectories can diverge from each other [16]. SLDV provides property-proving capability for Simulink models but only for open-loop, discrete-time (nonhybrid) models [8].

EMERGING SIMULATION-GUIDED TESTING APPROACHES

Several recently developed tools use simulation-guided approaches to testing based on the notion of falsification. Optimization-guided falsification is an emerging approach for testing closed-loop models by intelligently obtaining test inputs to expose undesirable model behaviors. The inputs to a falsification algorithm are a closed-loop model, a set of correctness requirements, a specification of the system parameters, and a definition of the exogenous inputs to the closed-loop model. The overall architecture of an optimization-guided falsification tool is shown in Figure 5; two key components are a simulation engine and an optimizer.

The procedure illustrated in Figure 5 takes as input the model $\mathcal{M}$, initial parameter values for the closed-loop model p_0, initial time-series values for the model inputs u_0, and the property to be falsified ψ. The simulation engine numerically computes time-series values for the system behaviors $\Phi(\mathcal{M},p,u)$. The cost function converts the system behaviors $\Phi(\mathcal{M},p,u)$ into a numeric cost value $c_\psi(\Phi(\mathcal{M},p,u))$ based on ψ. Note that the cost function is defined with respect to the correctness requirement(s) and can always be defined such that a negative cost corresponds to a violation of the requirements. If the cost is lower than zero, then the procedure halts, since the property has been falsified $(\Phi(\mathcal{M},p,u) \not\models \psi)$. If the cost is not lower than zero, then the optimizer selects a new set of operating conditions $\hat{p}$ and $\hat{u}$, and the procedure continues, with the simulation engine producing the next set of behaviors. The process continues until a falsifying behavior is found or until a user-specified limit on the number of iterations is reached.

Example: Automotive Fuel Control System

Consider the automotive powertrain control example illustrated in Figure 6. The model $\mathcal{M}_{FCS}$ is a simplified representation of a fuel-control subsystem (FCS) in an automotive engine and contains an air-fuel controller (controller) and engine air-fuel dynamics (plant). The plant subsystem contains a mean-value model of the engine dynamics including the throttle, intake manifold air, and fuel dynamics. The purpose of the controller is to regulate the ratio of air to fuel in the engine to a given reference value. The controller has four modes of operation: the startup mode, the normal mode, the power mode, and the fault mode.

As illustrated in Figure 6, the engine speed ω is an input to both the controller and the plant. A throttle control subsystem converts a throttle position command θ_{in} to throttle position θ. The output of the plant to the controller is the air-fuel ratio λ and the intake manifold inlet mass airflow rate $\dot{m}_{af}$. The controller output to the plant is the fuel rate command F_c.

The internal state variables for the system (not shown in the figure) include intake manifold pressure, two state variables associated with the sensor used to measure λ, one state variable associated with the throttle control, one state variable associated with the amount of fuel stored in the fuel film, and the state variables associated with a variable transport delay. The transport delay is used to model the time it takes for the exhausted gas to travel from the engine through the exhaust system to the point where the air-fuel ratio is measured by a sensor. The transport delay gives rise to a delay differential equation (DDE); theoretically, this DDE requires an infinite number of state variables to represent with an ODE. For a detailed description of the model, see [17], or see [18] for a description of a modified version along with corresponding models available for download.

The performance of the controller is sensitive to the accuracy of sensors and actuators. To capture these imperfections, the FCS model contains multiplicative error terms that model calibration and other tolerances in the corresponding components. Multiplicative error terms are present in the air-fuel ratio sensor, the fuel injection actuator, and the mass air-flow sensor. In the experiments that follow, the multiplicative error terms associated with the fuel injection actuator and the mass air flow sensor are set to 1.0 (0.0% error), and the multiplicative error parameter for the air-fuel ratio sensor error r_{AF} is assumed to be between 0.99 to 1.01 ($\pm 1.0\%$).

One requirement for the FCS is that the air-fuel ratio λ should not deviate from the reference value λ_{ref} by more than 2% after 11.0 s, which is 1.0 s after the transition to the normal mode occurs (at 10.0 s). This requirement can be captured informally as

$$\psi_{\text{FCS}} := \text{“}|\mu(t)| < 0.02, \text{ for } t \geq 11.0,\text{”} \tag{2}$$

where $\mu(t) := (\lambda(t) - \lambda_{\text{ref}})/\lambda_{\text{ref}}$. To check whether this requirement is violated for the FCS, a falsification tool will formulate a cost $c_{\text{FCS}}(\cdot)$ that maps output behaviors of $\mathcal{M}_{\text{FCS}}$ to a real value, based on ψ_{FCS}. The set of parameters P_{FCS} is the set of all values of r_{AF} for which $0.99 \leq r_{\text{AF}} \leq 1.01$, and the set of inputs U_{FCS} is given by sets of functions $\theta_{in}(\cdot)$ and $\omega(\cdot)$, defined over $0.0 \leq t \leq 50.0$ s. $\theta_{in}(\cdot)$ is a piecewise-constant function, such that $0.0 \leq \theta_{\text{in}}(\cdot) < 61.2^{\circ}$. $\omega(\cdot)$ is a constant signal with a value between 900 and 4000 r/min. The control system operates in the normal mode during the interval $11.0 \leq t \leq 50.0$ s. The goal of the falsification tool is to identify some parameter $p_{\text{FCS}} = r_{\text{AF}}$ and some input $u_{\text{FCS}} = (\theta_{in}(\cdot), \omega(\cdot))$, such that $c_{\text{FCS}}(\phi(\mathcal{M}_{\text{FCS}}, p_{\text{FCS}}, u_{\text{FCS}})) < 0$, which indicates that $\phi(\mathcal{M}_{\text{FCS}}, p_{\text{FCS}}, u_{\text{FCS}}) \not\models \psi_{\text{FCS}}$.

Simulation-Guided Falsification Tools

The S-TaLiRo and Breach tools, as well as the RRT-REX and multiple-shooting-based approaches, use simulations to perform falsification. These tools and techniques are described below.

S-TaLiRo

The key feature of S-TaLiRo is its ability to transform a falsification problem into an optimization problem by parameterizing the search space, which corresponds to input signals or model parameters [19] (visit [20] to download S-TaLiRo). First, users provide information about the range of values for inputs and parameters. Then, for each continuous exogenous input, the user selects the number of control points used to construct the input signals. For example, if a model has one exogenous input u with three control points, and the simulation time horizon is 10 s, then S-TaLiRo selects three values corresponding to $u(0)$, $u(5)$, and $u(10)$ (control points by default are distributed evenly across the time horizon). The actual $u(t)$ is then obtained by using a user-defined interpolation (such as constant, piecewise constant, piecewise linear, or splines) across the chosen control points.

The user provides the requirements using a temporal logic language. Then, an optimizer turns these requirements into cost functions to be minimized. S-TaLiRo supports various strategies for the stochastic optimization engine, such as simulated annealing [21], genetic algorithms, uniform random sampling, ant-colony optimization [22], and the cross-entropy method [23].

Consider the following example describing the S-TaLiRo tool applied to the FCS example. The S-TaLiRo settings used to address the example are as follows. Since $\omega(\cdot) = \omega_c$ is a constant signal, only one control point is selected between 900 and 4000 r/min. To simplify the falsification task, the signal $\theta_{in}(\cdot)$ is restricted to the class of pulse trains, with an amplitude $0.0 \leq a < 61.2^{\circ}$ and a period $5.0 \leq T_p \leq 10.0$ s. For each simulation, S-TaLiRo chooses one real constant value from 0.99 to 1.01 for the system parameter r_{AF}. For the optimization solver, simulated annealing is selected to falsify the requirement that the air-fuel ratio λ should not deviate from the reference value λ_{ref} by more than 2% during the interval $11.0 \leq t \leq 50.0$ s. The decision variables for the optimizer are the parameters ω_c, a, and T_p. A maximum limit of 1000 simulations is permitted to falsify the requirements. S-TaLiRo successfully identifies a falsifying behavior in about 400 s.

A plot of the falsifying behavior is shown in Figure 7(a). The figure shows the input accelerator angle $\theta_{\text{in}}(t)$ and the normalized air-fuel ratio $\mu(t)$. The signals in the figure are shown for $11.0 \leq t \leq 50.0$ s, because the property ψ_{FCS}

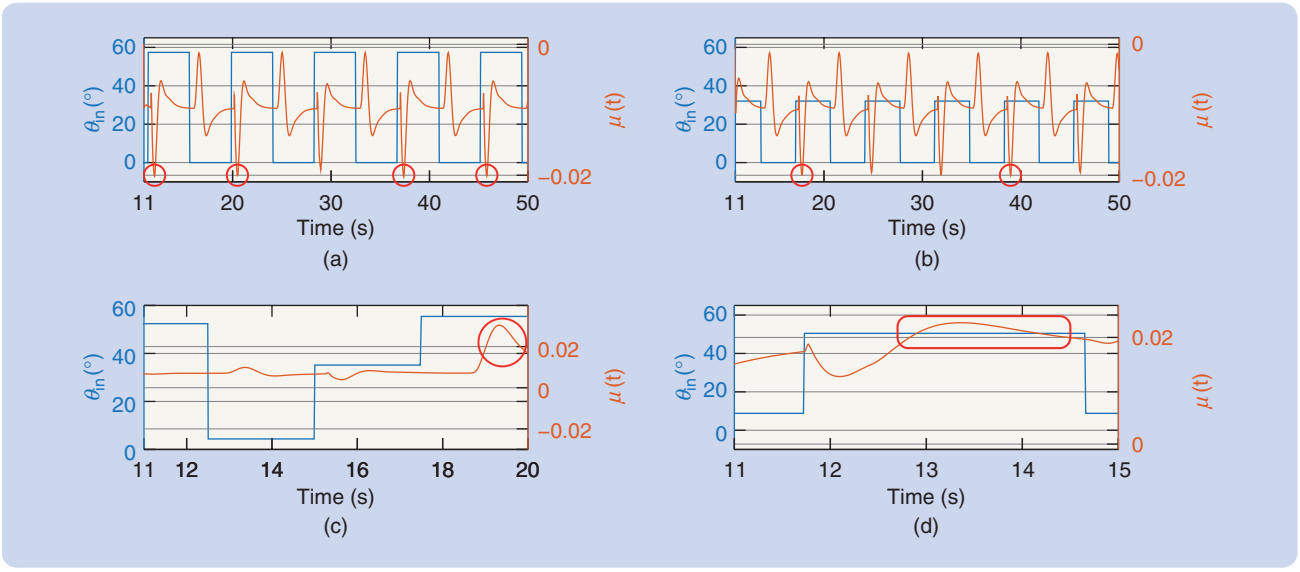


FIGURE 7 Results from falsification tools. Each plot illustrates a falsifying behavior found using one of the falsification tools featured in the article. Falsifying behavior found (a) by S-TaLiRo, (b) by Breach, (c) by RRT, and (d) by S3CAM.

specifies bounds on air-fuel ratio behavior during this period. The red circles indicate instants where $\mu(t) < -0.02$. Note that the behavior violates ψ_{FCS} since the magnitude of $\mu(t)$ is greater than 0.02 for $11.0 \leq t \leq 50.0$ s.

Breach

Similar to S-TaLiRo, Breach [24] also parameterizes exogenous inputs using ranges and control points (visit [25] to download Breach). A key difference is that Breach treats both model parameters and exogenous inputs in a uniform fashion. Moreover, Breach allows more freedom to place control points on the timeline. For instance, given a time horizon of ten for the input signal $u(t)$ with four control points, users can put control points at $u(0)$, $u(5)$, $u(6.5)$, and $u(10)$ to focus on the time duration $5.5 \leq t \leq 6$. Another difference is that Breach uses a nonlinear global optimizer based on the Nelder–Mead simplex-based algorithm [26]. Thus, when using Breach, users need to set optimization options, such as the number of Nelder–Mead iterations and the number of restarts. Note that the performance of the optimization varies for different optimization options.

Consider the following example of the Breach tool applied to the FCS example. The example has four parameters. The first parameter is the constant engine speed ω_c, where $900 \leq \omega_c \leq 4{,}000$ r/min. The second one is the air-fuel ratio sensor tolerance r_{AF}, where $0.99 \leq r_{\text{AF}} \leq 1.01$. The other two parameters specify the throttle command signal, where, as in the S-TaLiRo example, $\theta_{in}(\cdot)$ is assumed to be a pulse train with amplitude given by $0.0 \leq a < 61.2^\circ$ and period given by $5.0 \leq T_p \leq 10.0$ s. Since each of the parameters can be treated as a constant through the simulation, one control point is used for each parameter. Breach implements a more efficient algorithm to compute the cost function value than S-TaLiRo, given a requirement in a temporal logic format [27]. Breach falsifies the requirement in about 30 s by identifying a falsifying behavior in which the air-fuel ratio λ deviates from the reference value λ_{ref} by more than 2%. A plot of the falsifying behavior is shown in Figure 7(b). The figure shows the input accelerator angle $\theta_{in}(t)$ and the normalized air-fuel ratio $\mu(t)$ for the period $11.0 \leq t \leq 50.0$. The red circles indicate two instants where $\mu(t) < -0.02$. Note that the behavior violates ψ_{FCS} since the magnitude of $\mu(t)$ is greater than 0.02 for $11.0 \leq t \leq 50.0$ s.

Both S-TaLiRo and Breach support the *mining* of temporal requirements from closed-loop models, which is an automated procedure to obtain reasonable formal requirements from a design model. 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 simulations 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 [28]. 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.

RRT-REX

RRT-REX is a falsification framework that leverages notions of coverage to optimize bug-finding performance [29]. Using a concept related to coverage-based testing for software systems, RRT-REX uses a coverage metric called the star-discrepancy measure, which applies to the infinite

system states that exist in embedded control systems. Star-discrepancy is a statistical notion that quantifies how well a continuous state space is covered (by states discovered with simulations) [30]. 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 corresponding to u_i from time t_i to t_{i+1}. A rooted path in the tree $\mathbf{x}(t_0),\mathbf{x}(t_1),\ldots,\mathbf{x}(t_n)$ corresponds to a *partial* simulation (a simulation over an abbreviated time horizon) of the model output for 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 takes as input a Simulink model, a requirement in the form of a temporal logic formula, and several numerical parameters, such as bounds on the state variables and the length of the time segments. RRT-REX explores the state space of the Simulink model with the goal of maximizing the coverage achieved by the points in the tree constructed by the RRT algorithm. The tool provides additional guidance to the RRT algorithm by associating cost values (based on a given property ψ) for partial simulations 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 with a lower robustness value. While preliminary evaluations look promising, a mature implementation of the tool is under development.

Applying the RRT-REX tool to the FCS example presents some unique challenges. The FCS example has a subsystem modeling the exhaust gas transport dynamics as a variable transport delay. A delay function describes the input–output relation $y(t) = u(t-\Delta)$, where Δ is some real number. In the FCS example, the value of Δ is a nonlinear function of the model states (provided in the form of a lookup table). As RRT-REX constructs an exploration tree in the state space, systems with delays are a challenge because they correspond to a system with an infinite set of state variables. For the purposes of constructing the tree of simulations, RRT-REX does not attempt to store information corresponding to the states associated with the delay. Nevertheless, the underlying simulation framework, Simulink, is assumed to accurately capture the behaviors associated with the delay. Thus, while the tree does not store information regarding the delay states, the simulations corresponding to the nodes in the tree accurately capture the system behaviors, including behaviors associated with the delay component.

For the FCS example, RRT-REX searches a seven-dimensional model state space (ignoring the states associated with delays), by choosing input values between 0° and 61.0° at 2.5-s increments for the throttle angle input (except for the first 10.0 s, where the input is held constant). For the engine-speed input, one value between 900 and 4000 r/min is randomly chosen at the beginning of the simulation and held constant throughout the run of the tool (that is, for each partial simulation, the engine speed is held constant). Similarly, one value for r_{AF} from 0.99 to 1.01 is selected and used for each partial simulation. RRT-REX is able to find inputs that falsify the requirement that λ should not deviate from λ_{ref} by more than 2.0% during the interval $10.0 \le t \le 50.0$ s in an average of 325 s (since the procedure is stochastic, the average is provided over ten separate trials). A plot of one of the falsifying behaviors is shown in Figure 7(c). The figure shows the input accelerator angle $\theta_{\mathrm{in}}(t)$ and the normalized air-fuel ratio $\mu(t)$ for $t \ge 11.0$ s. The red circle indicates an instance where $\mu(t) > 0.02$. Note that the behavior violates ψ_{FCS} since the magnitude of $\mu(t)$ is greater than 0.02 for $t \ge 11.0$ s. The behavior for this example only extends to $t = 20.0$ s instead of continuing to $t = 50.0$ s, as in the other examples; this is because the RRT algorithm explores a tree of behaviors forward in time and is able to halt as soon as a falsifying behavior is identified.

Multiple-Shooting-Based Approaches

Using short disconnected partial simulations to find an abstract example of a falsifying simulation and using an optimizer to attempt to splice the partial simulations together to identify a concrete falsifying simulation is proposed in [31]. A significant recent revision of this work removes the dependence on an optimizer [32]. 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 simulation segment and then randomly chooses a new set of start states by sampling in an ϵ-neighborhood of the end point of the simulation segment. The algorithm is initialized by randomly sampling the initial set of states. A *segmented trace* is a collection of simulation 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 traces, 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 tool can choose to perform a concretization step that involves doing a complete simulation from the initial states corresponding to the most promising segmented traces. Ongoing investigations with this approach include evaluating 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.

A prototype tool called S3CAM implements the approach to splice segmented traces to obtain simulations that falsify a safety property. Several issues must be addressed to apply the S3CAM tool to the FCS example. Since S3CAM explicitly requires full observability of the

state space and an explicit representation for the states, certain model features are not supported by the tool. These features include delays because, as mentioned above, a delay element represents an infinite number of states, and S3CAM requires the model to have a finite dimensional, compact state space. To investigate the practical aspects of S3CAM, the FCS model was simplified to remove the delay elements, approximating these with a first-order delay.

The settings used to apply the S3CAM to the FCS model are described below. The engine speed input is held constant during the simulations and is picked from the range 900–4000 r/min, and the throttle angle input is parameterized as a pulse signal with period between 5.0 and 10.0 s and amplitude between 0.0° and 61.1°. As with the other experiments described above, the multiplicative error factor on the air-fuel ratio sensor measurement was assumed to be in the range 0.99–1.01, and the remaining sensors and actuators were assumed to have no errors. Using these settings, the S3CAM tool attempted to falsify the requirement ψ_{FCS} using segmented traces of 0.01-s duration, over a time horizon that extended to 50.0 s. No falsifying trace was found before the memory limit of the machine was reached. To demonstrate the falsifying performance of S3CAM, the tool was applied again, this time using a larger air-fuel ratio measurement error term range of 0.98–1.02 and a time horizon out to 15.0 s. Using this larger parameter range, the S3CAM was able to identify a simulation that falsifies the property ψ_{FCS} in approximately 30 s. A plot of the falsifying behavior is shown in Figure 7(d). The figure shows the input accelerator angle $\theta_{\text{in}}(t)$ and the normalized air-fuel ratio $\mu(t)$ for $t \geq 11.0$ s. The red oval indicates an instance where $\mu(t) > 0.02$. Note that the behavior violates ψ_{FCS} since $\mu(t) > 0.02$ for $11.0 \leq t \leq 50.0$ s.

EMERGING SIMULATION-GUIDED VERIFICATION APPROACHES

Verification of embedded control systems using traditional techniques is difficult or impossible for even the simplest embedded control systems. To increase the effectiveness of existing verification tools, new approaches are using information gleaned from simulations to assist the verification tools. Two emerging techniques address the issue of verification for embedded control systems using information obtained from simulations: simulation-guided Lyapunov analysis and simulation-guided contraction analysis.

Simulation-Guided Lyapunov Analysis

Informally, a dynamical system is asymptotically stable if any behavior converges to an equilibrium point of the system. Asymptotic stability can be used to prove that 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 *Lyapunov function* $v: \mathbb{R}^n \to \mathbb{R}$, where n is the number of state variables, that satisfies *Lyapunov conditions* in a given neighborhood of an equilibrium point:

a) v is positive everywhere in the domain (except at the equilibrium, where it is zero).
b) The time-derivative of v along the system dynamics is negative everywhere in the domain (except at the equilibrium, where it is zero).

In addition to proving stability, Lyapunov functions can also be used to characterize performance bounds and to perform safety verification. Three verification tasks that may be addressed using Lyapunov functions are as follows:

- *Stability*: For this verification task, $\psi :=$ "The system is stable." To verify this property, it is sufficient to identify a function v that satisfies a) and b).
- *Performance bounds*: For this verification task, $\psi :=$ "The system remains within some set $\mathcal{S}$." To verify this property, it is sufficient to identify a function v that satisfies a) and b) and a sublevel set $\hat{\mathcal{S}}$ of v, where $\hat{\mathcal{S}} = \{x \mid v(x) \leq l\}$, such that $\hat{\mathcal{S}} \subseteq \mathcal{S}$. Note that this only guarantees ψ 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, $\psi :=$ "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 a) and b) and sublevel set $\hat{\mathcal{S}}$ 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 [33] the authors present a simulation-guided approach to synthesize Lyapunov functions. The key idea, which is based on [34], is to assume that the desired Lyapunov function has a certain parameterized template form, namely a sum-of-squares polynomial of fixed degree. A set of linear constraints on the parameters in the Lyapunov function are obtained from simulations. 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*.

The procedure is illustrated in Figure 8. The procedure takes as input the system dynamics $\mathcal{M}$, initial parameters p_0, initial inputs u_0, an initial candidate Lyapunov function v_0, and the property to be proved ψ. The initial candidate Lyapunov function can be selected randomly based on a given template. Based on the inputs, the process simulation-based falsification uses 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 lower than zero, then the minimizing argument provides a witness that the candidate Lyapunov function is invalid. After the global optimizer obtains counterexamples, the associated linear constraints are included in an LP problem and the linear program solver obtains a solution

corresponding to an updated candidate Lyapunov function v. Based on the updated candidate, formulate solver query constructs a logical query over the system variables that is only satisfiable if $\mathcal{M} \not\models \psi$. Run solver is used to check whether the query is satisfiable. If the query is satisfiable (sat), then the solver returns counterexample conditions P_c, U_c that demonstrate $\mathcal{M} \not\models \psi$. these counterexample conditions are used in the next iteration of simulation-based falsification to improve the candidate Lyapunov function. If the query is not satisfiable, then the procedure terminates (halt) with the result $\mathcal{M} \models \psi$.

A key component of the procedure illustrated in Figure 8 is run solver, which employs nonlinear satisfiability modulo theories (SMT) tools. SMT tools use automated reasoning techniques to solve a general class of logical queries [35]. SMT tools based on interval constraint-propagation (such as dReal [36]) can be used to check the validity of the queries and will return counterexample conditions. Symbolic tools relying on quantifier elimination as implemented by the `Reduce` command in Mathematica [37] are also used to check validity of the queries, but these tools do not provide counterexample conditions; these tools are used in conjunction with interval constraint-propagation tools to provide counterexamples.

A related approach [38] performs program analysis. In that work, the authors use 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.

Example: Lyapunov Analysis for Automotive Fuel Control System

To better understand the simulation-guided Lyapunov approach, consider the following simplified version of the FCS example, which is taken from [33, ex. 5]. In this version of the FCS example, the discrete-time controller is replaced with a continuous-time approximation, and several other simplifications are made to the dynamic equations, such as removing the variable transport delay. Further, θ_{in} and ω are assumed to be fixed at 15.0° and approximately 1909.9 r/min (200.0 rad/s), respectively, and the multiplicative error terms are fixed at 1.0 (0.0%). Also, the system states are translated so that the equilibrium point coincides with the origin. The resulting model $\mathcal{M}_{\text{Lyap}}$ is a nonlinear ODE with four state variables and no input ($U_{\text{Lyap}} = \varnothing$). The specification is that λ should remain within 10% of a given λ_{ref}, or

$$\psi_{\text{Lyap}} := \text{“}|\mu(t)| < 0.1,\text{”} \tag{3}$$

if the system is initiated within a small region around the equilibrium point, given by the set $P_{\text{Lyap}} = \{x \,|\, \|x\|_2 < 0.02\}$. The verification task is to prove $\phi(\mathcal{M}_{\text{Lyap}}, P_{\text{Lyap}}, U_{\text{Lyap}}) \models \psi_{\text{Lyap}}$,

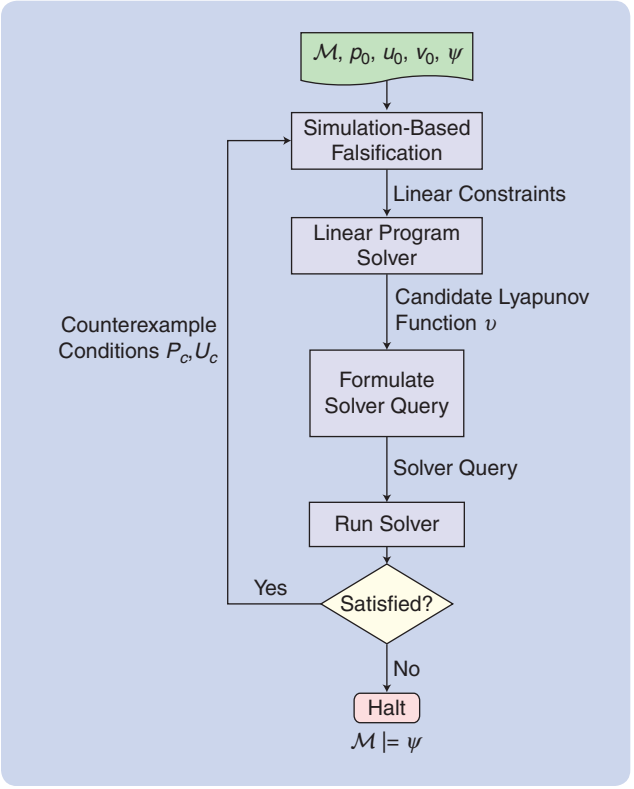


FIGURE 8 Automatic Lyapunov analysis. This procedure can be used to discover Lyapunov functions, forward invariant sets, or barrier certificates for control-design models.

which can be established by a barrier certificate. If a function v is identified that satisfies a) and b), and a sublevel set $\hat{\mathcal{S}}$ of v contains P_{Lyap} and excludes the set of states such that $|\mu(t)| \geq 0.1$, then (3) holds.

The procedure illustrated in Figure 8 was able to identify a barrier certificate that establishes (3) for this version of the FCS. The candidate barrier was found in 1413.73 s using simulations that explored a total of 258,078 system states. Verification by the dReal tool took 1157.42 s.

Since this is a verification example, it assumes that the controller is given, and the task is to verify that the closed-loop system satisfies performance bounds. An alternative way to approach the general engineering task of developing a correct control system would be to synthesize the controller using a correct-by-design technique that provides guarantees on the performance bounds. Lyapunov-related control synthesis techniques compute control-invariant sets, which are regions of the state space that the system will remain within for some control output and that satisfy performance bounds. The controller measures (or estimates) the system state, and the control output is selected to maintain the system within the control invariant set (see [39], for example). These techniques can be applied to nonlinear dynamical systems, like the FCS example, but they are computationally intensive and may not scale well with the dimension of the system.

Simulation-Guided Contraction Analysis

In contraction analysis or incremental stability analysis, instead of asking whether system trajectories converge to a known nominal behavior, the question is whether 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 instead of showing stability to an equilibrium point, as does a Lyapunov function, a contraction metric shows that pairs of trajectories converge. Contraction analysis can be used to perform bounded-time estimation of reachable sets of behaviors for the purpose of verification [40]–[42].

Related work uses *auto-bisimulation* functions to show that, for a given finite-time simulation with initial condition $x(0)$, finite-time trajectories that start inside a ball of radius δ centered at $x(0)$ lie within a "tube" of radius δ around $x(t)$ [43], [44]. This allows for performing a comprehensive verification analysis by exploring only a few candidate simulations.

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 [45]. Recent work [46] proposes a simulation-guided approach to find contraction metrics, in similar vein to the approach described in [33]. Here, the contraction metric is defined by a matrix, the entries of which are polynomials of some fixed degree but with undetermined coefficients. Using simulations, a set of necessary constraints for the contraction metric are obtained, each constraint is a linear matrix inequality. A semidefinite programming solver is used to obtain a candidate contraction metric. As in [33], 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.

Example: Contraction Analysis for Automotive Fuel Control System

To illustrate the simulation-guided contraction analysis technique, consider the model $\mathcal{M}_{\text{Lyap}}$ described above. In [46], a contraction metric for this system is constructed within the region $P_{\text{Lyap}} = \{x \mid \|x\|_2 < 0.01\}$. This contraction metric could be used to conservatively estimate the reachable set of states (behaviors). The conservative estimate of behaviors could then be used to prove properties for the system.

The C2E2 tool [16] uses a related approach to verify properties of hybrid systems using discrepancy functions instead of contraction metrics. Discrepancy functions are based on system dynamics and are used to characterize the rate at which pairs of behaviors diverge from each other. The C2E2 tool uses as input a Simulink model of the system annotated with a corresponding discrepancy function. The annotations are used to estimate the set of reachable system states to prove properties. In [47], C2E2 is used to prove properties, such as worst-case air-fuel ratio error, for a simplified version of system $\mathcal{M}_{\text{FCS}}$.

CONCLUSIONS

Industrial embedded control system designs are increasing in scale and complexity, which presents industry with modeling and quality-checking challenges. The MBD paradigm can be used to manage the complexity, and it provides the ability to simulate the modeled system. The simulations obtained from development models can be used to test control algorithms, identify flaws, and increase the quality of the design.

Several traditional and emerging tools and techniques use simulation-based approaches for embedded control system design, test, and verification. Table 1 summarizes

TABLE 1 Emerging simulation-guided techniques for PTC design analysis. Maturity scale (1–5): 1) academic implementation, 2) significant user effort required, 3) can handle some industrial models, 4) ready for industrial deployment, but not commercialized, and 5) fully commercialized and supported.

Technique/Tool	Type of Analysis	Can Include Plant	Scale of Model	Maturity
Reactis	Coverage		System	5
TestWeaver	Falsification	✓	System	5
SLDV (TVG)	Coverage	✓	System	5
SLDV (verifier)	Verification		Module	5
S-TaLiRo	Falsification	✓	System	4
Breach	Falsification	✓	System	4
RRT-REX	Falsification	✓	Feature	3
Multiple shooting	Falsification	✓	Module	2
Lyapunov analysis	Verification	✓	Module	1
Contraction analysis	Verification	✓	Module	1

the techniques discussed in this article. For each technique in the table, the second column indicates the type of analysis that the technique performs (coverage, falsification, or verification). The third column indicates whether the analysis can effectively include a representation of the plant dynamics. The fourth column indicates the scale of the model that can be addressed effectively. The model scale is subjective and is described by three categories: module (small), feature (medium), and system (large). Module-scale models implement some focused functional behavior. For controller models (without a plant), this could be a single functional behavior, like computing the desired spark timing for an automotive engine, and may consist of 100–1000 lines of code. For closed-loop models (with a plant), the functional behavior could be a single PID loop implementing control over a plant with 1–4 state variables. Feature-scale models implement some functionality that requires a composition of several modules. For example, this could be a collection of modules that is used to implement a startup mode for a jet engine. System-scale models represent some complete control design, such as a controller for an electronic insulin pump. The relative maturity level of the technology or tool is indicated in the last column. Maturity levels are indicated as a number between 1 and 5, where 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.

ACKNOWLEDGMENT

The authors thank Aditya Zutshi for providing the experimental results for the S3CAM example.

AUTHOR INFORMATION

James Kapinski (jim.kapinski@toyota.com) received the Ph.D. degree in electrical and computer engineering from Carnegie Mellon University (CMU) in 2005 and was a postdoctoral researcher at CMU from 2007 to 2008. He went on to found and lead Fixed-Point Consulting, serving clients in the defense, aerospace, and automotive industries. Since 2012, he has been with the Model-Based Development Group at the Toyota Technical Center in Los Angeles, serving as a principal engineer. His work at Toyota focuses on advanced research into verification techniques for embedded software for powertrain control systems. His research interests include verification techniques for embedded control system designs and analysis of hybrid dynamical systems. He is a Senior Member of the IEEE. He can be contacted at 1630 W. 186th St., Gardena, CA 90248 USA.

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

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

Hisahiro Ito received the Ph.D. degree in computational engineering and science (polymer physics) from Nagoya University in 2003. He lead the development of the software-in-the-loop simulator for Toyota's engine and transmission control software from 2009 to 2012. He is currently responsible for the integration of verification and validation techniques into development processes for large-scale, complex control system models.

Ken Butts is an executive engineer in the Powertrain Control Department at the Toyota Technical Center, Ann Arbor, Michigan, which he joined in 2003. His main responsibility is to investigate and apply model-based methods to improve engine control design and calibration productivity. He has worked in the field of automotive electronics and control since 1982, almost exclusively in research and advanced development of powertrain controls. He has a B.E. degree in electrical engineering from General Motors Institute (now Kettering University), an M.S. degree in electrical engineering from the University of Illinois, and a Ph.D. in electrical engineering systems from the University of Michigan. He is a Senior Member of the IEEE.

REFERENCES

[1] R. N. Charette. (2009, Feb.). This car runs on code," *IEEE Spectr.*, [Online]. Available: http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code

[2] United States Environmental Protection Agency, "EPA and NHTSA set standards to reduce greenhouse gases and improve fuel economy for model years 2017-2025 cars and light trucks," U.S. EPA, Office of Transportation and Air Quality, Tech. Rep. No. EPA-420-F-12-051, Aug. 2012.

[3] C. Ptolemaeus, Ed. *System Design, Modeling, and Simulation Using Ptolemy II*. Mountain View, CA : Ptolemy, 2014.

[4] E. J. Bissett, "Mathematical model of the thermal regeneration of a wall-flow monolith diesel particulate filter," *Chem. Eng. Sci.*, vol. 34, no. 7/8, pp. 1233–1244, Jan. 1984.

[5] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Eng. J.*, vol. 9, no. 5, pp. 193–200, 1994.

[6] S. Skogestad and I. Postlethwaite, *Multivariable Feedback Control: Analysis and Design*. New York: Wiley, 2005.

[7] Reactive Systems, Inc,(2003). Model-based testing and validation of control software with REACTIS. [Online]. Available: http://www.reactive-systems.com/papers/bcsf.pdf
[8] F. Leitner and S. Leue. (2008). Simulink design verifier vs. SPIN a comparative case study. *Proc. 13th Int. Workshop on Formal Methods for Industrial Critical Systems* [Online]. Available: http://www.inf.uni-konstanz.de/soft/research/publications/pdf/FMICS2008_FINAL.pdf
[9] A. A. Gadkari, A. Yeolekar, J. Suresh, S. Ramesh, S. Mohalik, and K. C. Shashidhar, "Automotgen: Automatic model oriented test generator for embedded control systems," in *Computer Aided Verification* (Lecture Notes in Computer Science, vol. 5123), G. Aarti and S. Malik, Eds. Berlin, Germany: Springer, 2008, pp. 204–208.
[10] J. Mauss and M. Tatar. "Systematic test and validation of complex embedded systems," in *Proc. Embedded Real Time Software and Systems*, Toulouse, France, 2014, pp. 1–10.
[11] M. Oded and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Proc. FORMATS-FTRTFT, LNCS*, 2004, vol. 3253, pp. 152–166.
[12] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" *J. Comp. Syst. Sci.*, vol. 57, no. 1, pp. 373–382, Aug. 1995.
[13] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "Spaceex: Scalable verification of hybrid systems," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. New York: Springer, 2011, pp. 379–395.
[14] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, "Uppaal 4.0," in *Proc. IEEE 3rd Int. Conf. Quantitative Evaluation of Systems*, 2006, pp. 125–126.
[15] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. New York: Springer, 2013, pp. 258–263.
[16] P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok, "C2E2: A verification tool for stateflow models," in *Proc. 21st Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, 2015, pp. 68–82.
[17] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts, "Powertrain control verification benchmark," in *Proc. ACM 17th Int. Conf. Hybrid Systems: Computation and Control*, 2014, pp. 253–262.
[18] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. (2014). Benchmarks for model transformations and conformance checking [Online]. *Proc. Int. Workshop Applied Verification for Continuous and Hybrid Systems. Cyber-Physical Systems Virtual Organization.* Available: http://cps-vo.org/node/16854
[19] Y Annpureddy, C Liu, G E. Fainekos, and S Sankaranarayanan, "S-TaLiRo: A tool for temporal logic falsification for hybrid systems," in *Proc. Tools and Algorithms for the Construction and Analysis of Systems,* 2011, pp. 254–257.
[20] S-taliro toolbox [Online]. Available: https://www.assembla.com/spaces/s-taliro_public/subversion/source
[21] T. Nghiem, S. Sankaranarayanan, G. E. Fainekos, F. Ivancic, A. Gupta, and G. J. Pappas, "Monte-Carlo techniques for falsification of temporal properties of non-linear hybrid systems," in *Proc. Hybrid Systems: Computation and Control*, 2010, pp. 211–220.
[22] Y. Singh, R. Annapureddy, and G. E. Fainekos, "Ant colonies for temporal logic falsification of hybrid systems," in *Proc. 36th Annu. Conf. IEEE Industrial Electronics*, 2010, pp. 91–96.
[23] S. Sankaranarayanan and G. E. Fainekos, "Falsification of temporal properties of hybrid systems using the cross-entropy method," in *Proc. ACM Int. Conf. Hybrid Systems: Computation and Control*, 2012, pp. 125–134.
[24] D. Alexandre. "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *Proc. Int. Conf. Computer Aided Verification*, 2010, pp. 167–170.
[25] Breach toolbox [Online]. Available: http://www.eecs.berkeley.edu/~donze/breach_page.html
[26] J. A. Nelder and R. Mead, "A simplex method for function minimization," *Computer J.*, vol. 7, no. 4, pp. 308–313, 1965.
[27] A. Donzé, T. Ferrère, and O. Maler, "Efficient robust monitoring for STL," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. . New York: Springer, 2013, pp. 264–279.
[28] X. Jin, A. Donzé, J. V. Deshmukh, and S. A. Seshia, "Mining requirements from closed-loop control models," in *Proc. 16th Int. Conf. Hybrid Systems: Computation and Control*, 2013, pp. 43–52.
[29] T. Dreossi, T. Dang, A. Donzé, J. Kapinski, J. Deshmukh, and X. Jin, "Efficient guiding strategies for testing of temporal properties of hybrid systems," in *NASA Formal Methods*, K. Havelund, G. Holzmann, and R. Joshi, Eds. New York: Springer, 2015.
[30] D. Thao and T. Nahhal, "Coverage-guided test generation for continuous and hybrid systems," *Formal Methods. Syst. Design*, vol. 34, no. 2, pp. 183–213, 2009.
[31] A. Zutshi, S. Sankaranarayanan, J. V. Deshmukh, and J. Kapinski, "A trajectory splicing approach to concretizing counterexamples for hybrid systems," in *Proc. IEEE 52nd Annu. Conf. Decision and Control*, 2013, pp. 3918–3925.
[32] J. V. Deshmukh, A. Zutshi, S. Sankaranarayanan, and J. Kapinski, "Multiple-shooting, CEGAR-based falsification for hybrid systems," in *Proc. 14th Int. Conf. Embedded Software*, 2014, pp. 5.
[33] J. Kapinski, J. V. Deshmukh, S. Sankaranarayanan, and N. Aréchiga, "Simulation-guided Lyapunov analysis for hybrid dynamical systems," in *Proc. Hybrid Systems: Computation and Control Conf.*, 2014, pp. 133–142.
[34] U. Topcu, P. Seiler, and A. Packard, "Local stability analysis using simulations and sum-of-squares programming," *Automatica*, vol. 44, no. 10, pp. 2669–2675, 2008.
[35] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, 2011.
[36] G. Sicun, J. Avigad, and E. M. Clarke. "δ-complete decision procedures for satisfiability over the reals," in *Automated Reasoning*, B. Gramlich, D. Miller, and U. Sattler, Eds. New York: Springer, 2012, pp. 286–300.
[37] W. Stephen, *The Mathematica Book, Version 4*. Cambridge, U.K.: Cambridge Univ. Press, 1999.
[38] A. Gupta, R. Majumdar, and A. Rybalchenko, "From tests to proofs," in *Proc. Tools and Algorithms for Construction and Analysis of Systems*, 2009, pp. 262–276.
[39] I. M. Mitchell, S. Kaynama, M. Chen, and M. Oishi, "Safety preserving control synthesis for sampled data systems," *Nonlinear Anal.: Hybrid Syst*, vol. 10, pp. 63–82, Nov. 2013.
[40] A. Agung Julius, G. E. Fainekos, M. Anand, I. Lee, and G. J. Pappas, "Robust test generation and coverage for hybrid systems," in *Proc. Hybrid Systems: Computation and Control*, 2007, pp. 329–342.
[41] P. S. Duggirala, S. Mitra, and M. Viswanathan, "Verification of annotated models from executions," in *Proc. Int. Conf. Embedded Software*, Montreal, QC, 2013, pp. 1–10.
[42] H. Zhenqi and S, Mitra, "Computing bounded reach sets from sampled simulation traces," in *Proc. Hybrid Systems: Computation and Control (part of CPS Week 2012)*, Beijing, China, 2012, pp. 291–294.
[43] G. Antoine and G. J. Pappas, "Approximate bisimulation: A bridge between computer science and control theory," *Eur. J. Control*, vol. 17, no. 5-6, pp. 568–578, 2011.
[44] G. E. Fainekos, A. Girard, and G. J. Pappas, "Temporal logic verification using simulation," in A. Eugene and P. Bouyer, Eds. *Formal Modeling and Analysis of Timed Systems, volume 4202 of Lecture Notes in Computer Science*. Berlin, Germany: Springer, 2006, pp. 171–186.
[45] E. M. Aylward, P. A. Parrilo, and J.-J. E. Slotine, "Stability and robustness analysis of nonlinear systems via contraction metrics and SOS programming," *Automatica*, vol. 44, no. 8, pp. 2163–2170, 2008.
[46] A. Balkan, J. V. Deshmukh, J. Kapinski, and P. Tabuada, "Simulation-guided contraction analysis," in *Proc. Indian Control Con.*, 2015, pp. 71–78.
[47] P. S. Duggirala, C. Fan, S. Mitra, and M. Viswanathan, "Meeting a powertrain verification challenge," in *Proc. Int. Conf. Computer Aided Verification*, 2015, pp. 536–543.
[48] A. Pnueli, "The temporal logic of programs," in *Proc. Symp. Foundations of Computer Science*, 1977, pp. 46–57.
[49] A. Rajeev and T. A. Henzinger, "A really temporal logic," in *Proc. Symp. Foundations of Computer Science*, 1989, pp. 164–169.
[50] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Syst.*, vol. 2, no. 4, pp. 255–299, 1990.
[51] M. Oded and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Proc. Formal Modeling and Analysis of Timed Systems*, 2004, pp. 152–166.
[52] G. E. Fainekos and G. J. Pappas. "Robustness of temporal logic specifications for continuous-time signals," *Theoretical Comp. Sci.*, vol. 410, no. 42, pp. 4262–4291, 2009.
[53] A. Donzé and O. Maler, "Robust satisfaction of temporal logic over real-valued signals," in *Proc. Formal Modeling and Analysis of Timed Systems*, 2010, pp. 92–106.