Combining Induction, Deduction, and Structure for Verification and Synthesis

Sanjit A. Seshia
UC Berkeley
sseshia@eecs.berkeley.edu

Abstract—**Even with impressive advances in formal methods, certain major challenges remain. Chief amongst these are environment modeling, incompleteness in specifications, and the hardness of underlying decision problems.**

In this paper, we characterize two trends that show great promise in meeting these challenges. The first trend is to perform verification by reduction to synthesis. The second is to solve the resulting synthesis problem by integrating traditional, deductive methods with inductive inference (learning from examples) using hypotheses about system structure. We present a formalization of such an integration, show how it can tackle hard problems in verification and synthesis, and outline directions for future work.

I. Introduction

Formal methods is a field of computer science and engineering concerned with the rigorous mathematical specification, design, and verification of systems [1], [2]. The field has made enormous strides over the last few decades. Verification techniques such as model checking [3], [4], [5] and theorem proving (see, e.g. [6], [7], [8]) are used routinely in the computer-aided design of integrated circuits and have been widely applied to find bugs in software and embedded, cyber-physical systems. However, certain problems in formal methods remain very challenging, stymied by computational hardness or requiring a very high level of tedious, manual effort in the verification process. In this paper, we outline these challenges for formal methods, discuss recent promising trends, and generalize these trends into a systematic methodology for tackling the challenges.

Let us begin by examining the traditional view of verification, as a decision problem with three inputs (see Figure 1):

1. A model of the system to be verified, S;
2. A model of the environment, E, and
3. The property to be verified, Φ.

The verifier generates as output a YES/NO answer, indicating whether or not S satisfies the property Φ in environment E. Typically, a NO output is accompanied by a counterexample, also called an error trace, which is an execution of the system that indicates how Φ is violated. Other debugging information may also be provided. For a YES answer, some formal verification tools also include a proof or certificate of correctness, which can be checked by an independent tool. The first point to note is that this view of verification is high-level and a bit idealized; in particular, it somewhat de-emphasizes the challenge in generating the inputs to the verification procedure. In practice, one does not always start with models S and E — these might have to be generated from implementations. To create the system model S, one

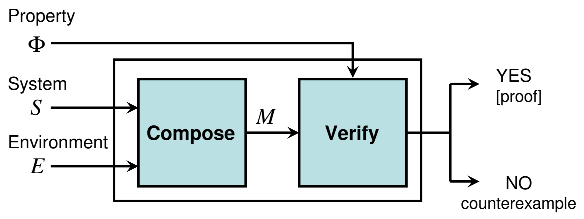

Fig. 1. A view of formal verification.

might need to perform automatic abstraction from code that has many low-level details. Similarly, the generation of an environment model E is usually a manual process, involving writing constraints over inputs, or a state machine description of the parts of the system S communicates with. Bugs can be missed due to incorrect environment modeling. In systems involving third-party components, not all details of the environment might even be available. Finally, the specification Φ is rarely complete and sometimes inconsistent, as has been noted in industrial practice (see, e.g., [9]). Indeed, the question "when are we done verifying?" often boils down to "have we written enough properties (and the right ones)?"

The second point we note is that Figure 1 omits some inputs that are crucial in successfully completing verification. For example, one might need to supply hints to the verifier in the form of inductive invariants or pick an abstract domain for generating suitable abstractions. One might need to break up the overall design into components and construct a compositional proof of correctness (or show that there is a bug). These tasks requiring human input have one aspect in common, which is that they involve a *synthesis sub-task* of the overall verification task. This sub-task involves the synthesis of *verification artifacts* such as inductive invariants, abstractions, environment assumptions, input constraints, auxiliary lemmas, ranking functions, and interpolants, amongst others. Thus, verification can be viewed as being performed via *reduction to synthesis*. One often needs human insight into at least the form of these artifacts, if not the artifacts themselves, to succeed in verification.

Finally, it has been a long-standing goal of the fields of electrical engineering and computer science to automatically synthesize systems from high-level specifications. In fact, the genesis of model checking lies in part in the automatic synthesis problem; the seminal paper on model checking by Clarke and Emerson [3] begins with this sentence:

> *"We propose a method of constructing concurrent programs in which the synchronization skeleton of the program is automatically synthesized from a*

high-level (branching time) Temporal Logic specification."

In automatic *formal synthesis*, one starts with a specification Φ of the system to be synthesized, along with a model of its environment E.[1] The goal of synthesis is to generate a system S from a class of systems $\mathcal{C}_S$ that satisfies Φ when composed with E. Figure 2 depicts the synthesis process. Modeled thus,

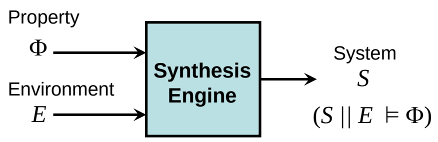

Fig. 2. A view of formal synthesis.

the essence of synthesis can be viewed as a *game solving* problem, where S and E represent the two players in a game; S is computed as a winning strategy ensuring that the composed system $S \| E$ satisfies Φ for all input sequences generated by the environment E. If such an S exists, we say that the specification (Φ and E) is *realizable*. Starting with the seminal work on automata-theoretic and deductive synthesis from specifications (e.g. [10], [11]), there has been steady progress on automatic synthesis. In particular, many recent techniques (e.g. [12], [13]) build upon the progress in formal verification in order to perform synthesis. However, there is a long way to go before automated synthesis is practical and widely applicable. One major challenge, shared with verification, is the difficulty of obtaining complete, formal specifications from the user. Even expert users find it difficult to write complete, formal specifications that are realizable. Often, when they do write complete specifications, the effort to write these is arguably more than that required to manually create the design in the first place. Additionally, the challenge of modeling the environment, as discussed above for verification, also remains for synthesis. Finally, synthesis problems typically have greater computational complexity than verification problems for the same class of specifications and models. For instance, equivalence checking of combinational circuits is NP-complete and routinely solved in industrial practice, whereas synthesizing a combinational circuit from a finite set of components is Σ_2-complete and only possible in very limited settings in practice. For some domains, both verification and synthesis are undecidable, but there are still compelling reasons to have efficient procedures in practice; a good example is the domain of hybrid systems — systems with both discrete and continuous state — whose continuous dynamics is non-linear, which arise commonly in cyber-physical systems and analog/mixed-signal circuits.

Let us summarize the observations made thus far. First, the main challenges facing formal verification and synthesis include system and environment modeling, creating good specifications, and the hardness of underlying decision problems. Second, the key to efficient verification is often in the synthesis of artifacts such as inductive invariants or abstractions — thus, verification is effectively solved by reduction to synthesis. Some of these challenges — such as dealing with computational hardness — can be partially addressed by advances in *computational engines* such as Binary Decision Diagrams (BDDs) [14], Boolean satisfiability (SAT) [15], and satisfiability modulo theories (SMT) solvers [16]. However, these alone are not sufficient to extend the reach of formal methods for verification and synthesis. Formal methods is fundamentally about *proof*, and for further advances, new *proof methodologies* are also required.

The close relation between verification and synthesis is something that has been noted by others, particularly Robert Brayton and his group. In systems such as VIS (Verification Interacting with Synthesis) [17] and ABC (a system for sequential synthesis and verification) [18], Brayton and his colleagues showed how techniques used in logic synthesis and optimization can substantially improve the effectiveness of formal verification techniques (and conversely, also how verification can improve synthesis). In this paper, we seek to make a different theoretical connection between verification and synthesis, and characterize a family of proof methods that hold much promise.

A fundamental characteristic of this family of proof methods is that they integrate induction and deduction. *Induction* is the process of inferring a general law or principle from observation of particular instances.[2] Machine learning algorithms are typically inductive, generalizing from (labeled) examples to obtain a learned *concept* or *classifier* [19], [20]. *Deduction*, on the other hand, involves the use of general rules and axioms to infer conclusions about particular problem instances. Traditional automated formal methods, such as model checking or theorem proving, are deductive. This is no surprise, as formal verification and synthesis problems (see Figures 1 and 2) are, by their very nature, deductive processes: given a particular specification Φ, environment E, and system S, a verifier typically uses a rule-based decision procedure for that class of Φ, E, and S to deduce if $S \| E \models \Phi$. On the other hand, inductive reasoning may seem out of place here, since an inductive argument only ensures that the truth of its premises make it *likely or probable* that its conclusion is also true. However, observe that humans often employ a combination of inductive and deductive reasoning while performing verification or synthesis. For example, while proving a theorem, one often starts by working out examples and trying to find a pattern in the properties satisfied by those examples. The latter step is a process of inductive generalization. These patterns might take the form of lemmas or background facts that then guide a deductive process of proving the statement of the theorem from known facts and previously established theorems (rules). Similarly, while creating a new design, one often starts by enumerating sample behaviors that the design must satisfy and hypothesizing components that might be useful in the design process; one then systematically combines these components, using design rules, to obtain a candidate artifact. The process usually iterates between inductive and deductive reasoning until the final artifact is obtained.

In this paper, we present a methodology, *SID*, that formalizes such a combination of inductive and deductive

[1] E may sometimes be encoded into Φ, but we prefer to keep it separate to emphasize the challenge of environment modeling.

[2] The term "induction" is often used in the verification community to refer to *mathematical induction*, which is actually a deductive proof rule. Here we are employing "induction" in its more classic usage arising from the field of Philosophy.

reasoning.[3] This methodology is inspired by recent successes in automatic abstraction and invariant generation such as counterexample-guided abstraction refinement (CEGAR) [23], as well as the advances in machine learning over the past several years. A key element in this combination is the use of *structure hypotheses*. These are mathematical hypotheses used to define the class of artifacts to be synthesized within the overall verification or synthesis problem. They are usually described in structural or syntactic terms. Structure hypotheses define a common language and associated constraints for inductive and deductive engines. Under these constraints, SID actively combines inductive and deductive reasoning: for instance, deductive techniques generate examples for learning, and inductive reasoning is used to guide the deductive engines.

To summarize, the core intellectual contributions of this paper are as follows: (i) the idea of performing verification by reduction to synthesis; (ii) SID, a formal methodology to combine inductive inference with deductive reasoning for solving synthesis problems; (iii) a theoretical framework showing how one can combine inductive and deductive reasoning to obtain the kinds of soundness and completeness guarantees needed in formal methods, and (iv) several examples of the methodology.

The rest of this paper is organized as follows. We describe the methodology in detail, with comparison to related work, in Section II. The methodology was first presented in earlier articles [24], [21], and some of the first novel synthesis efforts described in a Ph.D. thesis [22]. Since then, several new applications have been demonstrated, and a deeper understanding of the methodology has emerged, which is described in this article. We describe two new instances of this methodology in Section III. Future applications and further directions are explored in Section IV.

II. SID: Formalization and Related Work

We present a formalization of the SID methodology, and a discussion of related work. Section II-A defines basic notation for verification and synthesis problems. We discuss how verification can be reduced to synthesis in Section II-B. Given such reductions, one can simply focus on solving synthesis problems. The SID methodology for synthesis is formalized in Section II-C with a discussion of soundness and completeness in Section II-D. Related work is discussed in Section II-E. This section assumes some familiarity with basic terminology in formal verification and machine learning — see the relevant books by Clarke et al. [5], Manna and Pnueli [25], and Mitchell [19] for an introduction.

A. Verification and Synthesis Problems

As discussed in Section I, an instance of a verification problem is defined by a triple $\langle S, E, \Phi \rangle$, where S denotes the system, E is the environment, and Φ is the property to be verified. Here we assume that S, E, and Φ are described formally, in mathematical notation. For example, S and E can be finite-state systems, represented as Kripke structures, and Φ a linear temporal logic formula. The environment model E is sometimes included as part of the specification Φ. Similarly, an instance of a synthesis problem is defined by the pair $\langle \mathcal{C}_S, E, \Phi \rangle$, where the symbols S and E have the same meaning, and $\mathcal{C}_S$ defines a class of systems. As noted earlier, it is possible *in practice* for the descriptions of S, E, or Φ to be missing or incomplete; in such cases, the problem must be redefined as a synthesis problem in which the missing components must be synthesized so as to meet suitably modified objectives — e.g., if Φ is incomplete, one may synthesize the strongest requirement that is realizable along with a system that satisfies it.

A *family of verification or synthesis problems* is a triple $\langle \mathcal{C}_S, \mathcal{C}_E, \mathcal{C}_\Phi \rangle$ where $\mathcal{C}_S$ is a formal description of a class of systems, $\mathcal{C}_E$ is a formal description of a class of environment models, and $\mathcal{C}_\Phi$ is a formal description of a class of specifications.

B. Verification by Reduction to Synthesis

In this section, we formalize the notion of performing verification by reduction to synthesis. We begin by illustrating this notion with two examples.

Consider a common verification problem: proving that a certain property is an *invariant* of a system — i.e., that it holds in all states of that system. Let us first set up some notation. Additional background material may be found in a recent book chapter [26].

Let $M = (I, \delta)$ be a transition system where I is a logical formula encoding the set of initial states, and δ is a formula representing the transition relation. For simplicity, assume that M is finite-state, so that I and δ are Boolean formulas. Suppose we want to verify that M satisfies a temporal logic property $\Phi \doteq \mathbf{G}\, \phi$ where ϕ is a logical formula involving no temporal operators. We now consider two methods to perform such verification.

1) Invariant Inference: Consider first an approach to prove this property by (mathematical) induction. In this case, we seek to prove the validity of the following two logical statements:

$$\text{Base Case:} \quad I(s) \Rightarrow \phi(s) \tag{1}$$

$$\text{Induction Step:} \quad \phi(s) \wedge \delta(s, s') \Rightarrow \phi(s') \tag{2}$$

where, in the usual way, $\phi(s)$ denotes that the logical formula ϕ is expressed over variables encoding a state s.

In practice, when one attempts verification by induction as above for a system that is correct, one fails to prove the validity of the second statement, Formula 2. This failure is rarely due to any limitation in the underlying validity checkers for Formula 2. Instead, it is usually because the hypothesized invariant ϕ is "not strong enough." More precisely, ϕ needs to be conjoined (strengthened) with another formula, known as the *auxiliary inductive invariant*.

Put another way, the problem of verifying whether system satisfies an invariant property *reduces* to the problem of synthesizing an auxiliary invariant ψ such that the following two formulas are valid:

$$I(s) \Rightarrow \phi(s) \wedge \psi(s) \tag{3}$$

$$\phi(s) \wedge \psi(s) \wedge \delta(s, s') \Rightarrow \phi(s') \wedge \psi(s') \tag{4}$$

[3]SID stands for "Structure, Induction, and Deduction." In previous articles [21], [22], this methodology was termed "sciduction" which stood for the phrase "structure-constrained induction and deduction" and was chosen to draw an analogy with the *scientific method* of formulating hypotheses, validating them, drawing inferences, and iterating thus until a suitably validated hypothesis remains.

If no such ψ exists, then it means that the property ϕ is not an invariant of M, since otherwise, at a minimum, a ψ characterizing all reachable states of M should satisfy Formulas 3 and 4 above.

2) Abstraction-based Model Checking: Another common approach to solving the invariant verification problem is based on *sound and complete* abstraction. Given the original system M, one seeks to compute an abstract transition system $\alpha(M) = (I_\alpha, \delta_\alpha)$ such that $\alpha(M)$ satisfies Φ if and only if M satisfies Φ. This approach is computationally advantageous when the process of computing $\alpha(M)$ and then verifying whether it satisfies Φ is significantly more efficient than the process of directly verifying M in the first place. We do not seek to describe in detail what abstractions are used, or how they are computed. The only point we emphasize here is that the process of computing the abstraction is a synthesis task.

In other words, instead of directly verifying whether M satisfies Φ, we seek to synthesize an abstraction function α such that $\alpha(M)$ satisfies Φ if and only if M satisfies Φ, and then we verify whether $\alpha(M)$ satisfies Φ.

3) Reduction to Synthesis: Given the two examples above, let us now step back and formalize the general notion of performing verification by reduction to synthesis.

If the original verification problem is described by the tuple (S, E, Φ), the approach in the two examples above is to reduce it to a synthesis problem $(\mathcal{C}_T, D, \Omega)$ such that there exists a solution to $(\mathcal{C}_T, D, \Omega)$ if and only if there exists a solution to (S, E, Φ). To make things concrete, we instantiate the symbols above in the two examples.

Invariant inference. In this case, Φ is of the form $\mathbf{G}\,\phi$. The class $\mathcal{C}_T$ is the set of all Boolean formulae over the (propositional) state variables of M. The environment model D imposes no constraints; one can think of it as the logical formula $\mathtt{true}$. Ω comprises Formulas 3 and 4 above.

Abstraction-based verification. In this case, again, $\Phi = \mathbf{G}\phi$. $\mathcal{C}_T$ is the set of all abstraction functions α corresponding to a particular abstract domain [27]; for example, all possible localization abstractions [28]. The environment model D, once again, is $\mathtt{true}$. Ω is the statement "$\alpha(M)$ satisfies Φ if and only if M satisfies Φ".

Note that this is a reduction in the standard complexity-theoretic sense: the verification problem has a solution if and only if the synthesis problem has one. The synthesis problem is not any easier to solve, in the theoretical sense, than the original verification problem. However, the synthesis version may be easier to solve *in practice*, and may also be easier in the theoretical sense if a structure hypothesis imposes additional constraints. SID, which we describe in the next section, is a formalization of a family of particularly effective synthesis techniques.

We make one final remark about the reduction of verification to synthesis. Many such reductions involve the synthesis of specifications of various kinds: inductive invariants, pre-conditions, post-conditions, environment assumptions, interpolants, etc. Typically the size of these specifications is significantly smaller than that of the system being verified, but their presence can speed up verification by orders of magnitude. This yields some intuition into why a reduction to synthesis can be effective for verification.

C. Elements of the SID Methodology

SID is a family of algorithms for solving synthesis problems of the form $\langle \mathcal{C}_S, \mathcal{C}_E, \mathcal{C}_\Phi \rangle$ defined in Sec. II-A. An instance of SID can be described using a triple $\langle \mathcal{H}, \mathcal{I}, \mathcal{D} \rangle$, where the three elements are as follows:

1. *A structure hypothesis*, $\mathcal{H}$, encodes our hypothesis about the form of the *artifact to be synthesized*, whether it be an abstract system model, an environment model, an inductive invariant, a program, or a control algorithm (or any portion thereof);
2. *An inductive inference engine* or *Learner*, $\mathcal{I}$, is an algorithm for *learning from examples* an artifact h defined by $\mathcal{H}$, and
3. A *deductive engine* or *Teacher*, $\mathcal{D}$, is a *lightweight decision procedure* that applies deductive reasoning to answer queries made by $\mathcal{I}$.

Fig. 3 depicts the the above three elements in the most common mode of interaction between $\mathcal{I}$ and $\mathcal{D}$. Although the interface between $\mathcal{I}$ and $\mathcal{D}$ is usually captured by their definitions, it can be useful to separate out the description of the interface as a fourth element $\mathcal{Q}$ which comprises all types of queries that $\mathcal{I}$ can make to $\mathcal{D}$ along with the types of responses received. The rest of this section further elaborates on each of these elements.

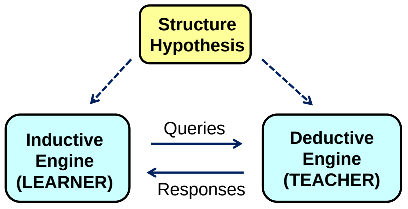

Fig. 3. Three Main Elements of the SID Approach

1) Structure Hypothesis: The structure hypothesis, $\mathcal{H}$, encodes our hypothesis about the structural or syntactic form of the artifact to be synthesized.

Formally, $\mathcal{H}$ is a (possibly infinite) set of *artifacts*. $\mathcal{H}$ represents a hypothesis that the system to be synthesized falls in a subclass $\mathcal{C}_\mathcal{H}$ of $\mathcal{C}_S$; i.e., $\mathcal{C}_\mathcal{H} \subseteq \mathcal{C}_S$. Note that $\mathcal{H}$ needs not be the same as $\mathcal{C}_\mathcal{H}$, since the artifact being synthesized might just be a portion of the full system description, such as the guard on transitions of a state machine, or the assignments to certain variables in a program. Each artifact $h \in \mathcal{H}$, in turn, is a unique set of *primitive elements* that defines its semantics. The form of the primitive element depends on the artifact to be synthesized.

More concretely, here are two examples of a structure hypothesis $\mathcal{H}$:

1. Suppose that $\mathcal{C}_S$ is the set of all finite automata over a set of input variables V and output variables U satisfying a specification Φ. Consider the structure hypothesis $\mathcal{H}$ that restricts the finite automata to be synchronous compositions of automata from a finite library L. The artifact to be synthesized is the entire finite automaton, and so, in this case, $\mathcal{H} = \mathcal{C}_\mathcal{H} \subset \mathcal{C}_S$. Each element $h \in \mathcal{H}$ is one such composition of automata from L. A primitive element is an input-output trace in the language of the automaton h.

2. Suppose that $\mathcal{C}_S$ is the set of all hybrid automata [29], where the guards on transitions between modes can be any region in $\mathbb{R}^n$ but where the modes of the automaton are fixed. A structure hypothesis $\mathcal{H}$ can restrict the guards to be hyperboxes in $\mathbb{R}^n$ — i.e., conjunctions of upper and lower bounds on continuous state variables. Each $h \in \mathcal{H}$ is one such hyperbox, and a primitive element is a point in h. Notice that $\mathcal{H}$ defines a subclass of hybrid automata $\mathcal{C}_{\mathcal{H}} \subset \mathcal{C}_S$ where the guards are n-dimensional hyperboxes. Note also that $\mathcal{H} \neq \mathcal{C}_{\mathcal{H}}$ in this case.

A structure hypothesis $\mathcal{H}$ can be syntactically described in several ways. For instance, in the second example above, $\mathcal{H}$ can define a guard either as a hyperbox in $\mathbb{R}^n$ or using mathematical logic as a conjunction of atomic formulas, each of which is an interval constraint over a real-valued variable.

2) Inductive Inference: The inductive inference procedure, $\mathcal{I}$, is an algorithm for learning from examples an artifact $h \in \mathcal{H}$. Here we use the term "examples" in its broadest sense, to include any relevant partial information about the specification for the synthesis problem that is provided in response to a query made by $\mathcal{I}$.

While any inductive learning procedure can be used, in the context of verification and synthesis the learning algorithms $\mathcal{I}$ tend to have one or more of the following characteristics:

- $\mathcal{I}$ performs *active learning*, selecting or generating the examples that it learns from.
- Examples and/or labels for examples are generated by one or more *oracles*. The oracles could be implemented using deductive procedures or by evaluation/execution of a model on a concrete input. In some cases, an oracle could even be a human user.
- $\mathcal{I}$ can invoke general-purpose deductive procedures, such as SAT or SMT solvers, to synthesize an artifact that is consistent with a set of labeled examples. This is in contrast with traditional machine learning algorithms that tend to be specially designed for each concept class.

An example of $\mathcal{I}$ is learning finite automata based on membership and equivalence queries, as originally formulated by Angluin [30], but using algorithmic techniques such as sequential equivalence checking as the oracle answering the queries. A contrast with mainstream machine learning is that examples are actively *generated* during the process of inductive inference, as opposed to being sampled from some (known or unknown) distribution. Such generated examples can be valuable beyond the synthesis process as test cases or simulation test patterns that can reveal corner-case bugs, as shown in Sec. III-B.

3) Deductive Reasoning: The deductive engine, $\mathcal{D}$, is a lightweight decision procedure that applies deductive reasoning to answer queries generated by $\mathcal{I}$.

Examples of $\mathcal{D}$ used in practice include SAT solvers, SMT solvers, model checkers, testing procedures, and numerical simulation procedures. The word "lightweight" refers to the fact that the decision problem being solved by $\mathcal{D}$ must be easier, in theoretical or practical terms, than that corresponding to the original synthesis or verification problem. Formally, at least one of the following conditions must hold:

1. *Easier in Theory:* If the original (synthesis or verification) problem is decidable, so must be the decision problem solved by $\mathcal{D}$, and further, the latter problem must have lower computational complexity. Otherwise, if the original (synthesis or verification) problem is undecidable, $\mathcal{D}$ must solve a decidable problem.
   An example of this approach is the component-based synthesis of circuits (loop-free programs) by iteratively invoking a SAT solver (NP) rather than solving the original problem (Σ_2) [12], [31].
2. *Easier in Practice:* One of the following two conditions must hold: (i) $\mathcal{D}$ solves a problem that is a strict special case of the original, or (ii) if $\mathcal{D}$ is also a decision procedure for the original problem, then it must be used on inputs of smaller size.
   An example of (i) is in the area of hybrid systems, where verifying safety properties of a general class of hybrid automata is replaced by verification of such properties for a single mode (purely continuous system) [32], [21] – note that both problems are undecidable in general. An example of (ii) is the use of finite-state model checking on abstractions of the original system that typically have a much smaller state space than the original [23].

4) $\mathcal{Q}$: Interface between $\mathcal{I}$ and $\mathcal{D}$: The definition of an instance $\langle \mathcal{H}, \mathcal{I}, \mathcal{D} \rangle$ is completed by fixing the interface between $\mathcal{I}$ and $\mathcal{D}$. This interface is given in terms of a finite set $\mathcal{Q}$ comprising each *type of query* $\mathcal{I}$ can make to $\mathcal{D}$, along with the *type of response* from $\mathcal{D}$. Thus $\mathcal{D}$ can be viewed as an oracle that guides $\mathcal{I}$. Example queries include witness queries (asking for positive or negative examples/primitive elements), membership queries (asking for the label for a specified example), equivalence/verification queries (asking if a candidate artifact is correct), etc. Such interaction is also termed as *query-based learning* [30] or *oracle-guided inductive synthesis* [33].

In some cases, it may be also be the case that queries are made by $\mathcal{D}$ to $\mathcal{I}$: e.g., consider a theorem prover that queries an inductive learner to conjecture lemmas consistent with provided examples. This mode of interaction is less-studied, but compatible with the framework given here. We note that this can be an interesting topic for future work.

Let Q denote the (possibly infinite) set of all query-response pairs encoding communication between $\mathcal{I}$ and $\mathcal{D}$. We can view Q as a subset of $Q_{\mathcal{I}} \times Q_{\mathcal{D}}$, where $Q_{\mathcal{I}}$ are the queries from $\mathcal{I}$ to $\mathcal{D}$ and $Q_{\mathcal{D}}$ are responses from $\mathcal{D}$ to $\mathcal{I}$. Then, we can formally define $\mathcal{I}$ as a mapping from Q^* to $\mathcal{H} \times Q_{\mathcal{I}}$; i.e., $\mathcal{I}$ maps a stream of query-reponse pairs to a conjectured artifact $h \in \mathcal{H}$ along with the next query. Similarly, $\mathcal{D}$ maps Q^* to $Q_{\mathcal{D}}$. A more detailed presentation of this interface may be found in [33].

Each query $q \in Q_{\mathcal{I}}$ can be formulated as a decision problem to be solved by $\mathcal{D}$. Here are some common examples of queries for $\mathcal{D}$ and the corresponding decision problems:

- Witness query: "Does there exist a positive/negative example for specification Φ?"
- Membership query: "Is L the label of this example?"
- Verification query: "Does the candidate artifact h satisfy specification Φ?"

For brevity, we omit a detailed exposition of the types of queries from this paper, making only two remarks: (i) the query types are similar to those studied in the machine learning literature on query-based learning [30], but not limited to them, and (ii) a more detailed theoretical study of the interface and associated problems can be found in [33].

5) Discussion: We now make a few remarks on the above formalism.

Inductive Bias from Structure Hypotheses. In the above description of the structure hypothesis, $\mathcal{H}$ only "loosely" restricts the class of systems to be synthesized, allowing the possibility that $\mathcal{C}_{\mathcal{H}} = \mathcal{C}_S$. We argue that a tighter restriction is often desirable. One important role of the structure hypothesis is to reduce the search space for synthesis, by restricting the class of artifacts $\mathcal{C}_S$. For example, a structure hypothesis could be a way of codifying the form of human insight to be provided to the synthesis process. Additionally, restricting $\mathcal{C}_{\mathcal{H}}$ also aids in inductive inference. Fundamentally, the effectiveness of inductive inference (i.e., of $\mathcal{I}$) is limited by the examples presented to it as input; therefore, it is important not only to select examples carefully, but also for the inference to generalize well beyond the presented examples. For this purpose, the structure hypothesis should place a strict restriction on the search space, by which we mean that $\mathcal{C}_{\mathcal{H}} \subsetneq \mathcal{C}_S$. The justification for this stricter restriction comes from the importance of *inductive bias* in machine learning. Inductive bias is the set of assumptions required to *deductively* infer a concept from the inputs to the learning algorithm [19]. If one places no restriction on the type of systems to be synthesized, the inductive inference engine $\mathcal{I}$ is unbiased; however, an unbiased learner will learn an artifact that is consistent only with the provided examples, with no generalization to unseen examples. As Mitchell [19] writes: "a learner that makes no a priori assumptions regarding the identity of the target concept has no rational basis for classifying any unseen instances." Given all these reasons, it is highly desirable for the structure hypothesis $\mathcal{H}$ to be such that $\mathcal{C}_{\mathcal{H}} \subsetneq \mathcal{C}_S$. We present in Sec. III applications of SID that have this feature.

Randomization. We also note that it is possible to use randomization in implementing $\mathcal{I}$ and $\mathcal{D}$. For example, a deductive decision procedure that uses randomization can generate a YES/NO answer with high probability.

Multiple Engines. Although we have defined SID as combining a single inductive engine with a single deductive engine, this is only for simplicity of the definition and poses no fundamental restriction. One can always view multiple inductive (deductive) engines as a being contained in a single inductive (deductive) procedure where this outer procedure passes its input to the appropriate "sub-engine" based on the type of input query.

D. Soundness and Completeness Guarantees

It is highly desirable for verification or synthesis procedures to provide *soundness* and *completeness* guarantees. In this section, we discuss the form these guarantees take for a procedure based on SID.

A verifier is said to be *sound* if, given an arbitrary problem instance $\langle S, E, \Phi \rangle$, the verifier outputs "YES" only if $S \| E \models \Phi$. The verifier is said to be *complete* if it outputs "NO" when $S \| E \not\models \Phi$.

The definitions for synthesis are similar. A synthesis technique is *sound* if, given an arbitrary problem instance $\langle \mathcal{C}_S, E, \Phi \rangle$, if it outputs S, then $S \in \mathcal{C}_S$ and $S \| E \models \Phi$. A synthesis technique is *complete* if, when there exists $S \in \mathcal{C}_S$ such that $S \| E \models \Phi$, it outputs at least one such S.

Formally, for a verification/synthesis procedure $\mathcal{P}$, we denote the statement "$\mathcal{P}$ is sound" by $\texttt{sound}(\mathcal{P})$.

Note that we can have probabilistic analogs of soundness and completeness. Informally, a verifier is *probabilistically sound* if it is sound with "high probability." For brevity, we will limit ourselves to non-probabilistic scenarios here.

1) Validity of the Structure Hypothesis: In SID, the existence of soundness and completeness guarantees depends on the validity of the structure hypothesis. For a synthesis problem, we say that the structure hypothesis $\mathcal{H}$ is *valid* if the subset of $\mathcal{C}_S$ satisfying the specification Φ contains at least one element in $\mathcal{C}_{\mathcal{H}}$. If Φ is available as a formal specification, this condition is precisely defined. However, as noted earlier, one of the challenges with synthesis can be the absence of good formal specifications. In such cases, we use Φ to denote a "golden" specification that one would have in the ideal scenario. For a verification problem, validity of the structure hypothesis is defined in terms of the synthesis problem it is reduced to (as in Sec. II-B).

Thus, for both verification and synthesis, the existence of an artifact satisfying the specification can be expressed as the following logical formula:

$$\exists c \in \mathcal{C}_S \,.\, c \models \Psi$$

where, for synthesis, Ψ is the original specification Φ, and, for verification, Ψ denotes the cumulative specification for the synthesis problem it is reduced to.

Similarly, the existence of an artifact satisfying Ψ that also satisfies the structure hypothesis $\mathcal{H}$ is written as:

$$\exists c \in \mathcal{C}_{\mathcal{H}} \,.\, c \models \Psi$$

Given the above logical formulas, we define the statement "the structure hypothesis is *valid*" as the validity of the logical statement $\texttt{valid}(\mathcal{H})$ given below:

$$\texttt{valid}(\mathcal{H}) \;\triangleq\; (\exists c \in \mathcal{C}_S \,.\, c \models \Psi) \Rightarrow (\exists c \in \mathcal{C}_{\mathcal{H}} \,.\, c \models \Psi) \tag{5}$$

In other words, if there exists an artifact to be synthesized (that satisfies the corresponding specification Ψ), then there exists one satisfying the structure hypothesis.

Note that $\texttt{valid}(\mathcal{H})$ is trivially valid if $\mathcal{C}_{\mathcal{H}} = \mathcal{C}_S$, or if the consequent $\exists c \in \mathcal{C}_{\mathcal{H}} \,.\, c \models \Psi$ is valid. An example of the former case is counterexample-guided abstraction refinement (CEGAR), an effective technique in verification that can be seen as a form of SID that synthesizes abstractions (Sec. II-E1 has a more detailed discussion of the link between CEGAR and SID.) However, in some cases, $\texttt{valid}(\mathcal{H})$ can be proved valid even without these cases simply by exploiting properties of Ψ and $\mathcal{C}_{\mathcal{H}}$; see [21] for an example.

2) Conditional Soundness: A verification/synthesis procedure following the SID paradigm must satisfy a conditional soundness guarantee: procedure $\mathcal{P}$ must be *sound if the structure hypothesis is valid.*

Without such a requirement, $\mathcal{P}$ is a heuristic, best-effort verification or synthesis procedure. (It could be extremely useful, nonetheless.) With this requirement, we have a mechanism to formalize the assumptions under which we obtain soundness — namely, the structure hypothesis.

More formally, the soundness requirement for $\mathcal{P}$ can be expressed as the following logical expression:

$$\texttt{valid}(\mathcal{H}) \Rightarrow \texttt{sound}(\mathcal{P}) \tag{6}$$

Note that one must prove $\texttt{sound}(\mathcal{P})$ under the assumption $\texttt{valid}(\mathcal{H})$, just like one proves unconditional soundness. The point is that making a structure hypothesis can allow one to devise procedures and prove soundness where previously this was difficult or impossible.

Where completeness is also desirable, one can formulate a similar notion of conditional completeness. We will mainly focus on soundness in this paper.

E. Context and Previous Work

Within computer science and engineering, the field of artificial intelligence (AI) has studied inductive and deductive reasoning and their connections (see, e.g., [34]). As mentioned earlier, Mitchell [19] describes how inductive inference can be formulated as a deduction problem where inductive bias is provided as an additional input to the deductive engine. *Inductive logic programming* [35], an approach to machine learning, blends induction and deduction by performing inference in first-order theories using examples and background knowledge. Combinations of inductive and deductive reasoning have also been explored for synthesizing programs (plans) in AI; for example, the SSGP approach [36] generates plans by sampling examples, generalizing from those samples, and then proving correctness of the generalization.

Our focus is on the use of combined inductive and deductive reasoning in *formal verification and synthesis*. While several techniques for verification and synthesis combine subsets of induction, deduction, and structure hypotheses, there are important distinctions between many of these and the SID approach. Below, we highlight a representative sample of related work; this sample is intended to be illustrative, not exhaustive.

1) Instances of SID: We first survey prior work in verification and synthesis that has provided inspiration for formulating the notion of SID. Specifically, SID can be seen as a "lens" through which one can view the common ideas amongst these techniques so as to extend and apply them to new problem domains.

Counterexample-Guided Abstraction-Refinement (CEGAR). Counterexample-guided abstraction refinement (CEGAR) [23] is an algorithmic approach to perform abstraction-based model checking. CEGAR has been successfully applied to hardware [23], software [37], and hybrid systems [38]. As depicted in Fig. 4, CEGAR solves the synthesis sub-task described in Sec. II-B2 of generating abstract models that are *sound* (they contain all behaviors of the original system) and *precise* (a counterexample generated for the abstract model is also a counterexample for the original system). One can view CEGAR as an instance of SID as follows:

- The *abstract domain*, which defines the form of the abstraction function, is the structure hypothesis. For example, in verifying digital circuits, one might use localization abstraction [28], in which abstract states are cubes over the state variables.
- The inductive engine $\mathcal{I}$ is an algorithm to learn a new abstraction function from a spurious counterexample. Consider the case of localization abstraction. One approach in CEGAR is to walk the lattice of abstraction functions, from most abstract (hide all variables) to least abstract (the original system). This problem can be viewed as a form of learning based on version spaces [19], although the traditional CEGAR refinement algorithms are somewhat different from the learning algorithms proposed in the version spaces framework. Gupta, Clarke, et al. [39] have previously observed the link to inductive learning and have proposed versions of CEGAR based on alternative learning algorithms (such as induction on decision trees).
- The deductive engine $\mathcal{D}$, for finite-state model checking, comprises the model checker and a SAT solver. The model checker is invoked on the abstract model to check the property of interest, while the SAT solver is used to check if a counterexample is spurious. One can view $\mathcal{D}$ as answering verification queries posed by $\mathcal{I}$.

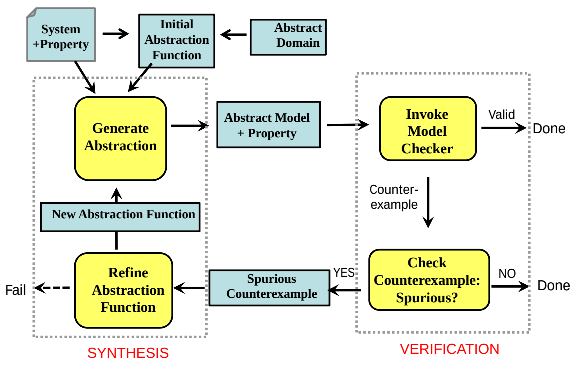

Fig. 4. Counterexample-guided abstraction refinement (CEGAR) as inductive synthesis.

In CEGAR, usually the original system is in $\mathcal{C}_{\mathcal{H}}$, and since it is a sound and precise (though trivial) abstract model, the consequent $\exists c \in \mathcal{C}_{\mathcal{H}} \,.\, c \models \Psi$ is valid. Thus, the structure hypothesis is valid, and the notion of soundness reduces to the traditional (unconditional) notion.

Other Instances. Several other common paradigms in verification and synthesis can also be seen as instances of SID:

- *Data-Driven Invariant Generation:* In recent years, an effective approach to generating inductive invariants is to assume that they have a particular structural form (e.g., conjunctions of affine constraints or equivalences), learn candidates that are consistent with simulation or test data, and then use a deductive engine (e.g., SAT solver, SMT solver, or model checker) to establish that the remaining candidates are indeed inductive invariants. Examples include inference of equivalences and implications in the ABC system [18], [40] and invariant inference for programs [41].
- *Counterexample-Guided Inductive Sythesis (CEGIS):* This is a novel approach to program synthesis [12] that starts by encoding programmer insight in the form of a *partial program*, or "sketch" (which forms the structure hypothesis), and then performs counterexample-guided learning similar to CEGAR to obtain an instantiation of the sketch that satisfies a specification.
- *Learning for Compositional Verification:* These techniques perform compositional verification by reduction to synthesis of environment models. The inductive learning algorithms used in these approaches are mostly based on

Angluin's L^* algorithm [30] and its variants; see [42] for a recent collection of papers on this topic. The structure hypothesis constrains the environment to be of a certain form (e.g. finite-state, with specific inputs and outputs). The deductive engine is typically a model checker.

- *Lazy SMT Solving:* SMT solvers can be seen as synthesizing a Boolean formula that is equisatisfiable to the original SMT formula. Lazy SMT solvers [16] perform inductive synthesis similar to CEGAR, by iteratively synthesizing lemmas over literals in the original SMT formula that rule out conjunctions of literals inconsistent with the underlying logical theories.

2) Techniques that are not SID: To contrast with the methods described above, we provide a few examples of verification and synthesis methods that are not instances of SID:

- *Cone-of-Influence Reduction:* This is a deductive approach to computing a sound and complete abstract model [5], where one computes the closure of all variables that may determine the value of the property to be verified. For many problems, the abstract model computed via cone-of-influence is too detailed to provide any efficiency benefits.
- *Automata-Theoretic Synthesis from Linear Temporal Logic (LTL):* This is a classic approach for synthesizing a finite-state transducer (FST) from an LTL specification, pioneered by Pnueli and Rosner [11]. The approach is a purely deductive one, with a final step that involves solving an emptiness problem for tree automata.
- *Deductive Invariant Generation:* Several techniques for generating inductive invariants, such as a method of Tiwari et al. [43] are deductive, using fixpoint computations and rule-based quantifier elimination.
- *Eager SMT Solving:* The eager approach to SMT solving [16], [44] is a deductive approach to synthesizing an equisatisfiable Boolean formula from the original SMT formula, based on applying rewrite rules, small-model theorems, and other satisfiability-preserving encoding strategies.

3) Recent Theoretical Results: Since the publication of the conference version of this article, the author has been involved in two complementary efforts that are closely related. The first is a class of synthesis problems called *syntax-guided synthesis* (SyGuS) [45], where the structure hypothesis is encoded into the problem definition in the form of grammars rather than be part of the solver. This approach circumvents the challenge of ensuring validity of the structure hypotheses by making it part of the problem definition. The second is a theoretical framework for *oracle-guided inductive synthesis* with an analysis of how variations in the counterexample-guided inductive synthesis paradigm can impact the convergence to a correct artifact [46], [33].

III. SID for Specification Synthesis

In this section, we present, in more depth, two recent applications of the SID approach. Both instances involve the synthesis of specifications in the context of an overall verification or synthesis problem. The first application concerns the generation of temporal logic specifications, particularly environment assumptions, for use in reactive controller synthesis [47], [48]. The second addresses the problem of verifying temporal logic properties of industrial closed-loop cyber-physical systems [49], where the properties are not readily available and hence the problem is reduced to one of synthesizing requirements. We also summarize other applications of SID in recent years.

A. Synthesis of Environment Assumptions

The synthesis of controllers from linear temporal logic (LTL) has gained increasing practical application in the fields of control systems and robotics (e.g., see [50], [51], [52], [53]). Recent algorithmic advances in synthesis from LTL (e.g., GR(1) synthesis [54]) have made it practical to generate controllers from large specifications in real-world settings. However, as discussed in Sec. I, a significant challenge to the wider adoption of this methodology is the need to write complete temporal logic requirements.

The most challenging part of writing a temporal logic specification for synthesis is writing the constraints (assumptions) on the environment's behavior. In the context of robotics, if the constraints are too strict, one risks generating a controller that is incapable of operating in the real world. On the other hand, if the constraints are too weak, the specification is likely to be *unrealizable*, i.e., there is no system that can meet the specification for such an unconstrained, adversarial environment.

The *environment assumption synthesis* (EAS) problem can be formalized as follows:

> $\langle$EAS$\rangle$: Given a satisfiable but unrealizable LTL formula Φ, generate another LTL formula Ψ such that $\Psi \neq \textbf{false}$, $\Psi \Rightarrow \Phi$ is realizable, and Ψ is the weakest such formula.

Here "weakest" means that there is no other LTL formula Ψ' such that $\Psi \Rightarrow \Psi'$ and $\Psi' \Rightarrow \Phi$ is realizable. For example, if formulas $\mathbf{G}\,\mathbf{F}\,p$ and $\mathbf{G}\,p$ are the only two assumptions that make the specification realizable, $\mathbf{G}\,\mathbf{F}\,p$ is weaker than $\mathbf{G}\,p$ (and hence the weakest).

There is usually one other requirement in practice: Ψ must be understandable by human designers. Ultimately, whether a specification is complete and environment assumptions are reasonable can only be determined by the designers. We interpret this requirement to mean that syntactically simpler LTL formulas are favored over large, complicated ones.

A purely deductive approach to this problem was given by Chatterjee et al. [55] where the authors compute the weakest environment assumption as a monolithic Büchi automaton. While this approach does solve the problem, the generated Ψ is usually not easily understandable by even an expert in formal methods.

In a recent paper [47], we proposed an alternate approach based on SID. At the top level, the approach is based on *counter-strategy-guided learning*, similar to CEGAR [23] and CEGIS [12], but using a *version space learning* algorithm in the synthesis phase. Fig. 5 illustrates the main ideas. We begin with three inputs: (i) an LTL formula Φ in the GR(1) fragment, (ii) a structure hypothesis $\mathcal{H}$ about the environment assumptions to be synthesized (provided in the form of templates for GR(1) properties), and (iii) (optionally) a set of input-output traces indicating environment behavior that must be allowed by any synthesized assumptions. Typically,

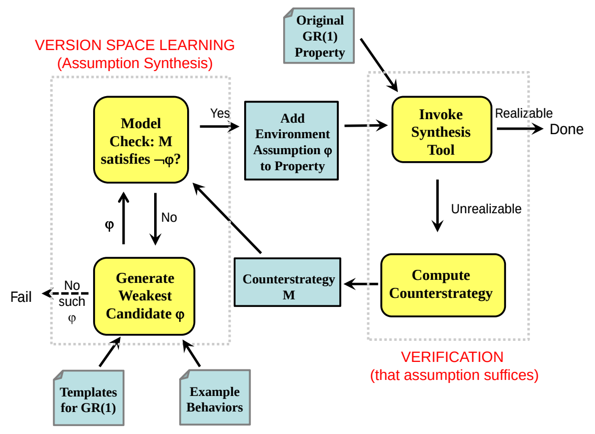

Fig. 5. Counterstrategy-guided synthesis of environment assumptions

the GR(1) formula Φ comprises just the requirements on the system, and no environment assumptions (so that the environment is a completely unconstrained adversary).

Then we invoke the GR(1) synthesis algorithm of Piterman et al. [54]. If Φ is realizable, then the process terminates with a synthesized finite-state transducer. If not, we compute the (finite-state) winning strategy for the environment, denoted M. The synthesis algorithm and the counterstrategy extraction together form a deductive procedure.

The structure hypothesis $\mathcal{H}$ defines a finite set of GR(1) properties using syntactic templates, as described in more detail by Li et al. [47], [48]. This set forms a lattice, with logical implication defining the partial order between formulas on the lattice. The weakest environment assumption, **true**, is the bottom element, and the strongest assumption, **false**, is the top element. The templates restrict the Boolean structure of the GR(1) properties, and therefore $\mathcal{C}_{\mathcal{H}} \subsetneq \mathcal{C}_S$.

Given this lattice of assumptions, we use inductive learning based on version spaces [19] to learn the weakest assumption consistent with our observations so far, including generated counterstrategies and the optional set of traces. The learning algorithm works as follows. First, using the traces, we can compute the "strongest frontier" of properties that are consistent with the traces; if there is no such set, we simply define the trivial frontier comprising all formulas such that the only stronger property is **false**. We maintain the "weakest frontier" of properties that are not implied by any previously added assumptions; initially this frontier comprises the single formula **true**. If the weakest frontier ever crosses the strongest frontier, then the synthesis step fails: there is no assumption satisfying the structure hypothesis $\mathcal{H}$ that makes the specification realizable. If not, given the counterstrategy M, we then select any property φ in the weakest frontier and invoke a model checker to verify whether $M \models \neg\varphi$. If so, adding φ as an environment assumption will rule out counterstrategy M. We further check that φ is consistent with all previously added environment assumptions. If either check fails, we update the frontier, pick another formula φ and iterate until we either fail or find an assumption φ that works.

Given the learned environment assumption φ, we add it to the set of environment assumptions generated so far. This set is then minimized to remove any redundant assumptions that are implied by the others. We then update the antecedent of the formula Φ accordingly, and iterate the loop. At any point, the conjunction of the current set of generated environment assumptions φ forms the environment assumption formula Ψ. When $\Psi \Rightarrow \Phi$ is realizable, the loop terminates successfully.

The guarantees of this SID approach are formalized in the theorem below.

Theorem *3.1:* The above algorithm is sound and complete for $\langle$EAS$\rangle$ when $\mathcal{H}$ is valid; i.e., when there exists a weakest environment assumption that is a conjunction of GR(1) formulas defined by the templates. □

Recall that soundness means that if the algorithm generates an environment assumption formula Ψ, then Ψ is the weakest environment assumption different from **false** such that $\Psi \Rightarrow \Phi$ is realizable. Soundness holds when $\mathcal{H}$ is valid due to the version space learning algorithm and the consistency and minimization steps. Completeness means that when a weakest environment assumption Ψ exists that yields realizability, the algorithm finds it. The finite search space imposed by $\mathcal{H}$ ensures that completeness holds when $\mathcal{H}$ is valid.

We have successfully applied this algorithm to infer environment assumptions almost identical to those written by human designers [47]. The approach has also been used to analyze an Federal Aviation Administration (FAA) specification, wherein the original specification was found unrealizable, and our algorithm was then used to suggest an additional environment assumption [48].

B. Synthesis of Requirements for Cyber-Physical Models

Cyber-physical systems (CPS) are those that tightly integrate computational processes with the physical world [62]. The model-based development (MBD) paradigm [63] is increasingly being employed for industrial cyber-physical systems, particularly in the avionics and automotive sectors. A model of a CPS typically comprises two logical parts: (i) the *plant*, which is usually the physical system being controlled such as the camshaft in an internal combustion automobile engine whose rotational dynamics must be modeled along with a thermodynamics model of the engine; and (ii) the *controller* that employs some specific control law to regulate the behavior of the physical system. The overall *closed-loop* model is then obtained as the composition of the controller with the plant. Such a model is usually expressed in industry-standard languages such as Simulink/Stateflow [64] or LabVIEW [65]. Given the closed-loop model, the verification problem is to determine whether the model satisfies a set of *requirements*, ideally expressed in a formal notation. Unfortunately, in current industrial practice, these requirements are high-level and often vague, and expressed in informal (natural) language. Examples of requirements in the automotive industry include "better fuel-efficiency", "signal should eventually settle", and "must exhibit resistance to turbulence". Verification is mainly done via simulation, where an engineer manually inspects simulation results to determine if any requirement was violated.

There is a thus a need for techniques and tools to help engineers in industry to write formal requirements while leveraging existing available information about designs. Specifically, the author and colleagues considered the problem of designing a tool that takes as input a closed-loop model in Simulink

Application	$\mathcal{H}$	$\mathcal{I}$	$\mathcal{D}$
Synthesis of bit-vector programs [31]	Loop-free programs from component library	Learning from distinguishing inputs	SMT solving for input/program generation
Timing analysis of embedded software [56], [57]	$w + \pi$ platform model & constraints	Game-theoretic online learning	SMT + ILP solving for basis path generation
Switching logic synthesis for safety [32]	Guards as hyperboxes	Hyperbox learning from labeled points	Numerical simulation as reachability oracle
Term-level verification [58] via abstraction [59]	Boolean formulas over restricted variable sets	Decision tree learning from simulations	SMT-based model checking
Switching logic synthesis for optimality [60]	Guards as halfspaces	Learning halfspaces from labeled points	Numerical optimization to find optimal switching points
Synthesis of environment assumptions in LTL [47], [48]	Restricted GR(1) templates	Version-space learning & Counterstrategy-guided learning	Automata-theoretic GR(1) synthesis & Finite-state model checking
Requirement synthesis for closed-loop control models [49]	Signal temporal logic (STL) templates	Counterexample-guided parameter learning	STL falsification based on numerical simulation
Model predictive control for temporal logic objectives [61]	Linearity for control and disturbances	Counterexample-guided learning of control	Mixed integer linear optimization

TABLE I
Summary of Selected Applications of SID. For each application, we briefly describe the structure hypothesis $\mathcal{H}$, the inductive inference technique(s) $\mathcal{I}$, and the deductive procedure(s) $\mathcal{D}$.

along with a suite of simulation traces describing permitted behavior, and outputs a requirement in a formal notation such as temporal logic [49]. With the increasing acceptance of temporal logics in practical domains such as automotive systems, it is reasonable to expect that libraries of commonly-used requirements will become available to control designers. Moreover, given the hybrid discrete-continuous nature of the models, one would need a flavor of temporal logic such as Metric Temporal Logic (MTL) [66], [67] or Signal Temporal Logic (STL) [68], [69]. In particular, STL has been found to be particularly well-suited to expressing common control-theoretic properties of signals involving overshoot, undershoot, settling-time, rise-time, dwell-time, etc. For example, the STL formula $\mathbf{F}_{[0,10]}(x > 3)$ expresses the property that the value of signal x exceeds 3 some time in the interval $[0, 10]$. Another useful feature of STL is that it has *quantitative satisfaction semantics*, i.e., given a trace, an STL formula is not just **true** or **false**, but has a numerical satisfaction value that is non-negative if and only if the Boolean satisfaction value is **true**.

Given the above, we now formally define the *CPS requirement synthesis* (CRS) problem as follows:

> $\langle$CRS$\rangle$: Given a closed-loop control model M (e.g., in Simulink), and a suite of simulation traces $\mathcal{T}$, generate a signal temporal logic (STL) formula Φ such that Φ is the strongest STL formula satisfied by M and consistent with $\mathcal{T}$.

As in the previous section, the generated STL formula Φ must be understandable and deemed to be reasonable by human designers. Additionally, there are no known purely deductive approaches to this problem due to the complexity of closed-loop Simulink models — non-linearity, switching, lookup tables, etc. — which do not fit in any known class of hybrid systems for which verification is decidable.

We have applied the SID paradigm to the $\langle$CRS$\rangle$ problem, with an approach that is a variation on the *counterexample-guided inductive synthesis* (CEGIS) [12] approach. Fig. 6 sketches the overall approach. We begin with the closed loop model M and a set of simulation traces $\mathcal{T}$. We first make a structure hypothesis to constrain the space of possible requirements. The structure hypothesis $\mathcal{H}$ is that the requirements are instances of a finite collection of templates expressed in *parametric signal temporal logic* (PSTL). A PSTL formula is obtained from an STL formula by replacing one or more numerical constants with parameters (variables). The templates are typically crafted based on input from the control engineers about the kinds of patterns they typically look for in simulation traces (e.g., settling time of a signal, maximum overshoot above a nominal upper bound, etc.). Clearly, the structure hypothesis severely restricts the STL properties we synthesize, so that $\mathcal{C}_{\mathcal{H}} \subsetneq \mathcal{C}_S$.

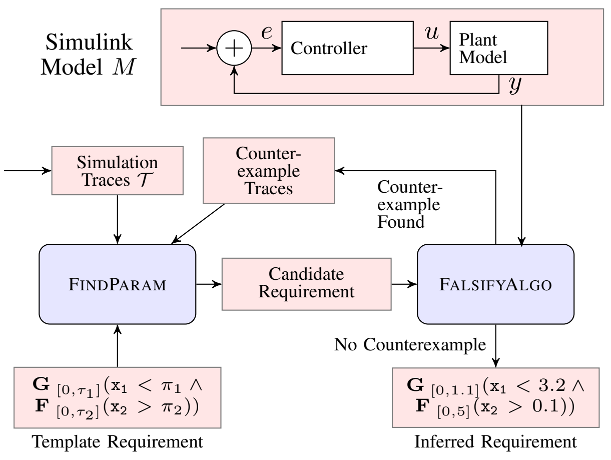

Fig. 6. Counterexample-guided synthesis of requirements for cyber-physical models

Next, using the templates and the set of traces $\mathcal{T}$, a parameter synthesis algorithm FINDPARAM is invoked to compute the strongest instantiation of the templates that are consistent with $\mathcal{T}$. For example, if in every trace in $\mathcal{T}$ the value of a particular signal x is always at most 42, then one can instantiate a template of the form $\mathbf{G}\,(x \leq \pi)$ with parameter π as $\mathbf{G}\,(x \leq 42)$. In general, finding such a strongest instantiation of a PSTL formula over a set of traces is computationally expensive, requiring exponential time in the worst to search over the parameter space. Fortunately, in all practical PSTL templates that we have encountered in the automotive domain, the PSTL formula has a special *monotonicity* property: the quantitative satisfaction value of the PSTL formula varies monotonically with the parameter valuation. Moreover, this monotonicity can be determined automatically by an encoding to a satisfiability modulo theories (SMT) solver. See [49] for further details.

Given such a strongest instantiation, i.e., a collection of STL properties, we then invoke FALSIFYALGO, an algorithm which tries to find a violation of one of these properties on the given model M. Ideally, we would like FALSIFYALGO to be a sound and complete verification tool that, given a Simulink models M and an STL property Φ, can decide whether or not M satisfies Φ. However, this problem is undecidable if the class of models M is not severely restricted. Thus, in general, one cannot prove that M satisfies Φ, one can only try to find violations of Φ, a process termed as *falsification*. In recent years, there has been substantial progress on state-of-the-art falsification algorithms for MTL and STL, based on numerical minimization of the quantitative satisfaction function, implemented in tools such as S-TALIRO [70] and BREACH [71]. If FALSIFYALGO finds a violation of the instantiated properties, then the resulting trace is added back into $\mathcal{T}$ and the loop repeated. Otherwise, the CEGIS loop terminates with the instantiated STL properties generated as output to the control engineer.

The guarantees of this SID approach are formalized in the theorem below.

Theorem *3.2:* The above algorithm for the $\langle$CRS$\rangle$ problem is sound and complete when $\mathcal{H}$ is valid and when FALSIFYALGO is a sound verifier for the model M. □

For this problem, soundness means that if the algorithm generates an STL formula Φ, then Φ is the strongest requirement satisfied by M and consistent with $\mathcal{T}$. Completeness means that when such a strongest requirement exists, the algorithm will find it. The soundness of the verifier FALSIFYALGO is key to giving this guarantee, since without it one cannot claim that M satisfies Φ, and one might also miss counterexamples from which to synthesize the strongest Φ.

In spite of the strong condition under which soundness and completeness is guaranteed, which may not hold in practice, the approach has proved extremely effective on industrial models [49]. In particular, the approach has been used on industrial automotive models supplied by Toyota engineers, including a large model of an airpath controller for a diesel engine. In these experiments, not only was the approach useful in synthesizing requirements, it also found a corner-case bug in the afore-mentioned model that was confirmed by a designer [49]. This work highlights another, more practical, connection between verification and synthesis: synthesizing the strongest requirement satisfied by the system can be an effective method for finding tricky corner-case bugs in the system, since the generated counterexamples and the final synthesized requirement provides information about parts of the model where bugs may lie.

C. Summary of Other Instances

The SID methodology has been applied by the author and colleagues to several other problems in specification, verification, and synthesis, as summarized in Table I. In addition, other researchers have very recently applied this approach for problems as diverse as Lyapunov analysis for control [72], and data-driven invariant inference for programs [41].

IV. CONCLUSION AND FUTURE DIRECTIONS

This paper posits that SID, a tight integration of induction and deduction with a structure hypothesis, is a promising approach to addressing challenging problems in formal methods and its applications. The crux of formal methods lies in algorithmic techniques for proof, and our proposal is inspired by the approaches human mathematicians typically employ, by combining inductive reasoning with systematic deductive processes. We show how some of the recent successes in formal methods, such as counterexample-guided abstraction refinement, can be seen as instances of SID. In particular, the structure hypothesis provides a way for a human to provide creative input into the verification process without getting mired in tedious details.

Given a new problem in verification or synthesis, how can we apply the ideas in this paper to tackle it? Here is a general prescription for applying SID:

1. Identify the hard synthesis sub-task(s) within the overall synthesis or verification problem. For example, in verification, one may need to synthesize inductive invariants or abstractions. For synthesis, one may need to synthesize parts of the specification or the implementation that are tricky or tedious to manually generate.
2. Formalize suitable structure hypotheses for each sub-task. We envision that, for a new problem, the structure hypothesis will be manually provided by the user. This step would require the user to have expertise both in formal methods and in the application domain. However, with experience, one might be able to build a "set of templates" into the verification or synthesis tool that encode reasonable structure hypotheses for non-expert users.
3. Formalize the reduction of the overall problem to synthesis. From a complexity-theoretic viewpoint, each synthesis sub-task can be viewed as a problem solvable by an *oracle* procedure. This oracle synthesis procedure must be designed to be sound/complete when the structure hypothesis is valid.
   Note that the SID paradigm may be recursively applied to the synthesis problem being solved by the oracle procedure.
4. Prove the validity of the structure hypotheses or derive reasonable assumptions that entail its validity.

Since the publication of the conference version of this paper [21], the author and colleagues have successfully applied this paradigm to a variety of problems, including synthesizing requirements for cyber-physical models [49], model predictive control in a receding horizon framework [61], and synthesizing strategies for probabilistic models with applications to risk-limiting renewable energy pricing [73].

In closing, we consider several directions for future work.

First, recall that the soundness and completeness guarantees of SID only hold when the structure hypothesis is valid. If unconditional guarantees are needed for a particular application, one needs to prove the validity of a candidate hypothesis $\mathcal{H}$. It would be useful to develop a general, systematic approach for checking the validity of $\mathcal{H}$.

Second, it is important for the structure hypothesis to be flexible and programmable, since it may need to be changed if the original synthesis problem is unrealizable, or if the specification changes. Consequently, it is also important for the inductive learning algorithms to be flexible to accommodate such changes in the structure hypotheses. Such flexibility is not typical of traditional machine learning algorithms.

Third, we note that SID offers ways to integrate inductive reasoning into deductive engines, and vice-versa. It is intruiging to consider if SAT and SMT solvers can benefit from this approach — for example, using inductive reasoning to guide the solver for specific families of SAT/SMT formulas, or to learn how to instantiate quantifiers in a theorem prover. Similarly, can one effectively use deductive engines as oracles in learning at scale? Are there new concept learning problems that can be effectively solved using this approach?

Finally, the landscape of applications is yet to be fully explored. An interesting direction is to take problems that have classically been addressed by purely deductive methods and apply the SID approach to them. As an example, consider again the problem of synthesis from LTL specifications. One challenge for this problem is to deal with the doubly-exponential computational complexity. It would be interesting to see if the synthesis algorithms themselves can be made more scalable using SID, e.g., by combining machine learning algorithms with traditional deductive methods. An initial step towards this objective has been taken by the author and colleagues for strategy synthesis of Markov Decision Processes (MDPs) for LTL objectives [74], but much more remains to be done.

Acknowledgments

This article is a result of ideas synthesized and verified (!) over many years in collaboration with several students and colleagues. The contributions of Susmit Jha, in particular, are gratefully acknowledged. This work has been supported in part by several sponsors including the National Science Foundation (CNS-0644436, CNS-0627734, CNS-1035672, and CCF-1139138), Semiconductor Research Corporation (SRC) contracts 1355.001 and 2045.001, an Alfred P. Sloan Research Fellowship, the Hellman Family Faculty Fund, the Toyota Motor Corporation under the CHESS center, and the Gigascale Systems Research Center (GSRC), MultiScale Systems Center (MuSyC), and TerraSwarm Research Center, three centers funded by Semiconductor Research Corporation programs sponsored by MARCO and DARPA.

REFERENCES

[1] J. M. Wing, "A specifier's introduction to formal methods," *IEEE Computer*, vol. 23, no. 9, pp. 8–24, September 1990.

[2] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[3] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs*, 1981, pp. 52–71.

[4] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *Symposium on Programming*, ser. LNCS, no. 137, 1982, pp. 337–351.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2000.

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

[7] M. J. C. Gordon and T. F. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[8] M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[9] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient detection of vacuity in ACTL formulas," *Formal Methods in System Design*, vol. 18, no. 2, pp. 141–162, 2001.

[10] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM TOPLAS*, vol. 2, no. 1, pp. 90–121, 1980.

[11] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *ACM Symposium on Principles of Programming Languages (POPL)*, 1989, pp. 179–190.

[12] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, 2006.

[13] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2010, pp. 313–326.

[14] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, August 1986.

[15] S. Malik and L. Zhang, "Boolean satisfiability: From theoretical hardness to practical success," *Communications of the ACM (CACM)*, vol. 52, no. 8, pp. 76–82, 2009.

[16] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, A. Biere, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 4, ch. 8.

[17] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo *et al.*, "VIS: A system for verification and synthesis," in *Computer Aided Verification (CAV)*. Springer, 1996, pp. 428–432.

[18] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *Computer Aided Verification (CAV)*, 2010.

[19] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[20] D. Angluin and C. H. Smith, "Inductive inference: Theory and methods," *ACM Computing Surveys*, vol. 15, pp. 237–269, Sep. 1983.

[21] S. A. Seshia, "Sciduction: Combining induction, deduction, and structure for verification and synthesis," in *Proceedings of the Design Automation Conference (DAC)*, June 2012, pp. 356–365.

[22] S. K. Jha, "Towards automated system synthesis using sciduction," Ph.D. dissertation, EECS Department, University of California, Berkeley, Nov 2011.

[23] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *12th International Conference on Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, vol. 1855. Springer, 2000, pp. 154–169.

[24] S. A. Seshia, "Sciduction: Combining induction, deduction, and structure for verification and synthesis," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-68, May 2011.

[25] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems:Specification*. Springer-Verlag, 1992.

[26] S. A. Seshia, N. Sharygina, and S. Tripakis, "Modeling for verification," in *Handbook of Model Checking*, E. M. Clarke, T. Henzinger, and H. Veith, Eds. Springer, 2014, ch. 3.

[27] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.

[28] R. Kurshan, "Automata-theoretic verification of coordinating processes," in *11th International Conference on Analysis and Optimization of Systems – Discrete Event Systems*, ser. LNCS. Springer, 1994, vol. 199, pp. 16–28.

[29] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical Computer Science*, vol. 138, no. 1, pp. 3–34, February 1995.

[30] D. Angluin, "Queries and concept learning," *Machine Learning*, vol. 2, pp. 319–342, 1988.

[31] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, 2010, pp. 215–224.

[32] ——, "Synthesizing switching logic for safety and dwell-time requirements," in *Proceedings of the International Conference on Cyber-Physical Systems (ICCPS)*, April 2010, pp. 22–31.

[33] S. Jha and S. A. Seshia, "A Theory of Formal Synthesis via Inductive Learning," *ArXiv e-prints*, May 2015.

[34] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[35] S. Muggleton and L. de Raedt, "Inductive logic programming: Theory and methods," *The Journal of Logic Programming*, vol. 19-20, no. 1, pp. 629–679, 1994.

[36] H. Fox, "Agent problem solving by inductive and deductive program synthesis," Ph.D. dissertation, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2008.

[37] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Proc. ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, June 2001, pp. 203–213.

[38] E. M. Clarke, A. Fehnker, Z. Han, B. H. Krogh, O. Stursberg, and M. Theobald, "Verification of hybrid systems based on counterexample-guided abstraction refinement," in *TACAS*, 2003, pp. 192–207.

[39] A. Gupta, "Learning abstractions for model checking," Ph.D. dissertation, Computer Science Department, Carnegie Mellon University, 2006.

[40] M. Case, "On invariants to characterize the state space for sequential logic synthesis and formal verification," Ph.D. dissertation, EECS Department, UC Berkeley, Apr 2009.

[41] R. Sharma and A. Aiken, "From invariant checking to invariant inference using randomized search," in *26th International Conference on Computer Aided Verification (CAV)*, 2014, pp. 88–105.

[42] Dimitra Giannakopoulou and Corina S. Pasareanu, eds., "Special issue on learning techniques for compositional reasoning," *Formal Methods in System Design*, vol. 32, no. 3, pp. 173–174, 2008.

[43] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar, "A technique for invariant generation," in *In 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2001, pp. 113–127.

[44] S. A. Seshia, "Adaptive eager boolean encoding for arithmetic reasoning in verification," Ph.D. dissertation, Carnegie Mellon University, 2005.

[45] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, October 2013.

[46] S. Jha and S. A. Seshia, "Are there good mistakes? a theoretical analysis of cegis," in *3rd Workshop on Synthesis (SYNT)*, July 2014, pp. 84–99.

[47] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *Proceedings of the Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, July 2011, pp. 43–50.

[48] W. Li, "Specification mining: New formalisms, algorithms and applications," Ph.D. dissertation, EECS Department, University of California, Berkeley, Mar 2014.

[49] X. Jin, A. Donzé, J. Deshmukh, and S. A. Seshia, "Mining requirements from closed-loop control models," in *Proceedings of the International Conference on Hybrid Systems: Computation and Control (HSCC)*, April 2013, pp. 43–52.

[50] T. Wongpiromsarn, "Formal methods for design and verification of embedded control systems: application to an autonomous vehicle," Ph.D. dissertation, California Institute of Technology, 2010.

[51] T. Wongpiromsarn, U. Topcu, and R. Murray, "Receding horizon temporal logic planning for dynamical systems," in *48th IEEE Conference on Decision and Control (CDC)*, Dec. 2009, pp. 5997 –6004.

[52] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Temporal-logic-based reactive mission and motion planning," *IEEE Transactions on Robotics*, vol. 25, no. 6, pp. 1370–1381, 2009.

[53] P. Nuzzo, H. Xu, N. Ozay, J. B. Finn, A. L. Sangiovanni-Vincentelli, R. M. Murray, A. Donzé, and S. A. Seshia, "A contract-based methodology for aircraft electric power system design," *IEEE Access*, vol. 2, pp. 1–25, 2014.

[54] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," in *7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. Lecture Notes in Computer Science, vol. 3855. Springer, 2006, pp. 364–380.

[55] K. Chatterjee, T. A. Henzinger, and B. Jobstmann, "Environment assumptions for synthesis," in *Proceedings of the 19th international conference on Concurrency Theory (CONCUR)*. Springer-Verlag, 2008, pp. 147–161.

[56] S. A. Seshia and A. Rakhlin, "Game-theoretic timing analysis," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE Press, 2008, pp. 575–582.

[57] ——, "Quantitative analysis of systems using game-theoretic learning," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 11, no. S2, pp. 55:1–55:27, 2012.

[58] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions," in *Proc. Computer-Aided Verification (CAV'02)*, ser. LNCS 2404, E. Brinksma and K. G. Larsen, Eds., July 2002, pp. 78–92.

[59] B. Brady, R. E. Bryant, and S. A. Seshia, "Learning conditional abstractions," in *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, October 2011, pp. 116–124.

[60] S. Jha, S. A. Seshia, and A. Tiwari, "Synthesis of optimal switching logic for hybrid systems," in *Proceedings of the International Conference on Embedded Software (EMSOFT)*, October 2011, pp. 107–116.

[61] V. Raman, A. Donzé, D. Sadigh, R. M. Murray, and S. A. Seshia, "Reactive synthesis from signal temporal logic specifications," in *Proceedings of the 8th International Conference on Hybrid Systems: Computation and Control (HSCC 2015)*, April 2015, pp. 239–248.

[62] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*, first edition ed. http://leeseshia.org, 2011.

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

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

[65] National Instruments, Inc., "LabVIEW," 2015. [Online]. Available: http://www.ni.com/labview/

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

[67] R. Alur, T. Feder, and T. A. Henzinger, "The Benefits of Relaxing Punctuality," *J. ACM*, vol. 43, no. 1, pp. 116–146, Jan. 1996.

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

[69] E. Asarin, A. Donzé, O. Maler, and D. Nickovic, "Parametric identification of temporal properties," in *Proc. of Runtime Verification*, 2011, pp. 147–160.

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

[71] A. Donzé, "Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems," in *Proc. of Computer Aided Verification*, 2010, pp. 167–170.

[72] J. Kapinski, J. V. Deshmukh, S. Sankaranarayanan, and N. Arechiga, "Simulation-guided Lyapunov analysis for hybrid dynamical systems," in *17th International Conference on Hybrid Systems: Computation and Control (HSCC)*, 2014, pp. 133–142.

[73] A. Puggelli, A. Sangiovanni-Vincentelli, and S. A. Seshia, "Robust strategy synthesis for probabilistic systems applied to risk-limiting renewable-energy pricing," in *Proceedings of the 14th International Conference on Embedded Software (EMSOFT)*, October 2014.

[74] D. Sadigh, E. S. Kim, S. Coogan, S. Sastry, and S. A. Seshia, "A learning based approach to control synthesis of markov decision processes for linear temporal logic specifications," in *Proceedings of the 53rd IEEE Conference on Decision and Control (CDC)*, December 2014, pp. 1091–1096.

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