Sciduction: 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 verification, certain major challenges remain. Chief amongst these are environment modeling, incompleteness in specifications, and the complexity of underlying decision problems.

In this position paper, we contend that these challenges can be tackled by integrating traditional, deductive methods with inductive inference (learning from examples) using hypotheses about system structure. We present *sciduction*, a formalization of such an integration, show how it can tackle hard problems in verification and synthesis, and outline directions for future work.

Categories and Subject Descriptors

B.5.2 [**Design Aids**]: Verification; I.2.6 [**Learning**]: Concept Learning

General Terms

Algorithms, Design, Theory, Verification

Keywords

Formal verification, synthesis, learning, deduction, induction

1. INTRODUCTION

> *"Formal methods research is about mechanizing creativity" — J. Strother Moore, FMCAD 2011 keynote.*
>
> *"To be creative requires divergent thinking (generating many unique ideas) and then convergent thinking (combining those ideas into the best result)." – Bronson & Merryman, Newsweek, July 2010.*

Formal verification has made enormous strides over the last few decades. Verification techniques such as model checking [12, 38, 15] and theorem proving (see, e.g. [25]) are used routinely in computer-aided design of integrated circuits and have been widely applied to find bugs in software and embedded systems. However, certain problems in system verification remain very challenging, stymied by computational hardness or requiring significant human input into the verification process. This position paper seeks to outline these challenges, discuss recent promising trends, and generalize these trends into an approach for tackling these 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. Some formal verification tools also include a proof or certificate of correctness with a YES answer. The first point to note

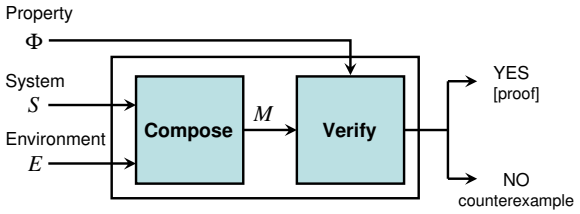

Figure 1: Formal verification procedure.

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 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 intellectual property (IP), 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., [6]). 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 key 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.[1] This sub-task involves the synthesis of *verification artifacts* such as inductive invariants, auxiliary lemmas, abstractions, environment assumptions or input constraints for compositional reasoning, etc. One often needs human insight into at least the form of these artifacts, if not the artifacts themselves, to succeed in verifying the design.

[1]Here we use the term "*synthesis*" in a different way from what is standard in the EDA community. We do not mean "logic synthesis". Rather, we mean the synthesis of formal artifacts from very high-level specifications or as part of the verification procedure.

DAC 2012, Jun 3-7 2012, San Francisco, CA, USA.

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. The EDA community has been in the forefront of work towards this goal. 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 [12] 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 synthesis, one starts with a specification Φ of the system to be synthesized, along with a model of its environment E. The goal of synthesis is to generate a system S that satisfies Φ when composed with E. Figure 2 depicts the synthesis process. Modeled thus, the essence of synthesis can be viewed as a *game*

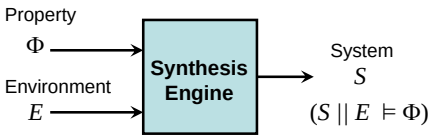

Figure 2: Formal synthesis procedure.

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. [31, 37]), there has been steady progress on automatic synthesis. In particular, many recent techniques (e.g. [49, 50]) 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. In some cases, both verification and synthesis are undecidable, but there are still compelling reasons to have efficient procedures in practice; a good example is hybrid systems — systems with both discrete and continuous state — whose continuous dynamics is non-linear, which arise commonly in embedded systems and analog/mixed-signal circuits.

To summarize, there are two main points. First, the main challenges facing formal verification and synthesis include system and environment modeling, creating good specifications, and the complexity of underlying decision problems. Second, the key to efficient verification is often in the synthesis of artifacts such as inductive invariants or abstractions, and thus, at the core, the challenges facing automatic verification are very similar to those facing automatic synthesis. Some of these challenges — such as dealing with computational complexity — can be partially addressed by advances in *computational engines* such as Binary Decision Diagrams (BDDs) [9], Boolean satisfiability (SAT) [29], and satisfiability modulo theories (SMT) solvers [5]. However, these alone are not sufficient to extend the reach of formal methods for verification and synthesis. New *methodologies* are also required.

As J. Strother Moore observed in his FMCAD 2011 keynote talk, formal methods involve the mechanization of human creativity in design. Arguably, the design of circuits and programs ranks amongst the most creative activities performed by humankind, alongside finding proofs in mathematics, conducting basic scientific research, composing music, creating artistic paintings, and more. The question naturally arises: do current automated formal methods follow the same approach as these other creative endeavors? Upon reflection, we find that this is not entirely the case. We contend that the "creativity gap" arises from an imbalance between the use of *deduction* and *induction* in formal methods, that recent trends are closing the gap, and that one can build upon those trends to develop a new paradigm for formal verification and synthesis.

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* [34, 3]. *Deduction*, on the other hand, involves the use of general rules and axioms to infer conclusions about particular problem instances. Traditional formal verification and synthesis techniques, 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 typically an inductive argument only ensures that the truth of its premises make it *likely or probable* that its conclusion is also true. However, one observes 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, *sciduction*, that formalizes such a combination of inductive and deductive reasoning.[3] This methodology is inspired by recent successes in automatic abstraction and invariant generation such as counterexample-guided abstraction refinement (CEGAR), as well as the enormous advances in machine learning over the past two decades. The key in integrating induction and deduction 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. Sciduction constrains inductive and deductive reasoning

[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.

[3] sciduction stands for structure-constrained induction and deduction. An early version of this position paper appeared as a UC Berkeley technical report [43].

using structure hypotheses, and actively combines inductive and deductive reasoning: for instance, deductive techniques generate examples for learning, and inductive reasoning is used to guide the deductive engines. We explain how one can combine inductive and deductive reasoning to obtain the kinds of soundness/completeness guarantees one desires in formal verification and synthesis.

The rest of this paper is organized as follows. We describe the methodology in detail, with comparison to related work, in Section 2. Some new instances of this methodology are presented in Section 3. Future applications and further directions are explored in Section 4.

2. SCIDUCTION: FORMALIZATION AND RELATED WORK

We begin with a formalization of the sciduction methodology, and then compare it to related work. This section assumes some familiarity with basic terminology in formal verification and machine learning — see the relevant books by Clarke et al. [15], Manna and Pnueli [30], and Mitchell [34] for an introduction.

2.1 Verification and Synthesis Problems

As discussed in Section 1, 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. Similarly, an instance of a synthesis problem is defined by the pair $\langle E, \Phi \rangle$, where the symbols have the same meaning. In both cases, 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 missing components must be synthesized as part of the overall verification or synthesis process.

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. In the case of synthesis, $\mathcal{C}_S$ defines the class of systems to be synthesized.

2.2 Elements of sciduction

An instance of sciduction 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*, $\mathcal{I}$, is an algorithm for *learning from examples* an artifact h defined by $\mathcal{H}$, and
3. A *deductive engine*, $\mathcal{D}$, is a *lightweight decision procedure* that applies deductive reasoning to answer queries generated in the synthesis or verification process.

We elaborate on these elements below. For concreteness, the context of synthesis is used to explain the central ideas in the approach. Note however that all of the discussion applies also to verification, in the context of synthesis of *verification artifacts*. We will note points specific to verification or synthesis as they arise.

2.2.1 Structure Hypothesis

The structure hypothesis, $\mathcal{H}$, encodes our hypothesis about the form of the artifact to be synthesized.

Formally $\mathcal{H}$ is a (possibly infinite) set of *artifacts*. $\mathcal{H}$ encodes 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, corresponds to 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, $\mathcal{H} = \mathcal{C}_\mathcal{H}$. 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 finite automaton h.
2. Suppose that $\mathcal{C}_S$ is the set of all hybrid automata [1], 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 set-theoretically 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.

2.2.2 Inductive Inference

The inductive inference procedure, $\mathcal{I}$, is an algorithm for learning from examples an artifact $h \in \mathcal{H}$.

While any inductive inference engine can be used, in the context of verification and synthesis we expect that the learning algorithms $\mathcal{I}$ have one or more of the following characteristics:

- $\mathcal{I}$ performs *active learning*, selecting 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 be a human user.
- A deductive procedure is invoked in order to synthesize a concept (artifact) that is consistent with a set of labeled examples. The idea is to formulate this synthesis problem as a decision problem where the concept to be output is generated from the satisfying assignment.

2.2.3 Deductive Reasoning

The deductive engine, $\mathcal{D}$, is a lightweight decision procedure that applies deductive reasoning to answer queries generated in the synthesis or verification process.

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 overall synthesis or verification problem.

In theoretical terms, "lightweight" means that at least one of the following conditions must hold:

1. $\mathcal{D}$ must solve a problem that is a strict special case of the original.
2. One of the following two cases must hold:
   (i) If the original (synthesis or verification) problem is decidable, and the worst-case running time of the best known procedure for the original problem is $O(T(n))$, then $\mathcal{D}$ must run in time $o(T(n))$.

(ii) If the original (synthesis or verification) problem is undecidable, $\mathcal{D}$ must solve a decidable problem.

In practical terms, the notion of "lightweight" is fuzzier: intuitively, the class of problems addressed by $\mathcal{D}$ must be "more easily solved in practice" than the original problem class. For example, $\mathcal{D}$ could be a finite-state model checker that is invoked only on abstractions of the original system produced by, say, localization abstraction [26] — it is lightweight if the abstractions are solved faster in practice than the original concrete instance. Due to this fuzziness, it is preferable to define "lightweight" in theoretical terms whenever possible.

$\mathcal{D}$ can be used to answer queries generated by $\mathcal{I}$, where the query is typically formulated as a decision problem to be solved by $\mathcal{D}$. Here are some examples of tasks $\mathcal{D}$ can perform and the corresponding decision problems:

- Generating examples for the learning algorithm.
  Decision problem: "does there exist an example satisfying the criterion of the learning algorithm?"
- Generating labels for examples selected by the learning algorithm.
  Decision problem: "is L the label of this example?"
- Synthesizing candidate artifacts.
  Decision problem: "does there exists an artifact consistent with the observed behaviors/examples?"

2.2.4 Discussion

We now make a few remarks on the above formalism.

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 [34]. 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 [34] 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. 3 two applications of sciduction that have this feature.

Another point to note is 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.

Next, although we have defined sciduction 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.

Finally, in our definition of sciduction, we do not advocate any particular technique of combining inductive and deductive reasoning. Indeed, we envisage that there are many ways to "configure" the combination of $\mathcal{H}$, $\mathcal{D}$, and $\mathcal{I}$, perhaps using inductive procedures within deductive engines and vice-versa. Any mode of integrating $\mathcal{H}$, $\mathcal{I}$, and $\mathcal{D}$ that satisfies the requirements stated above on each of those three elements is admissible. We expect that the particular requirements of each application will define the mode of integration that works best for that application. We present illustrative examples in Section 3.

2.3 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 sciduction.

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 $\langle E, \Phi \rangle$, if it outputs S, then $S \| E \models \Phi$. A synthesis technique is *complete* if, when there exists 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." We leave a more precise discussion of this point to a later stage in this paper when it becomes relevant.

2.3.1 Validity of the Structure Hypothesis

In sciduction, the existence of soundness and completeness guarantees depends on the validity of the structure hypothesis. Informally, we say that the structure hypothesis $\mathcal{H}$ is *valid* if the set of correct artifacts to be synthesized, if any exist, is guaranteed to include one in $\mathcal{C}_\mathcal{H}$.

Let us elaborate on what we mean by the phrase "correct artifacts to be synthesized":

- In the context of a synthesis problem, this is relatively easy: one seeks an element c of $\mathcal{C}_S$ that satisfies a specification Φ. If Φ is available as a formal specification, this phrase 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 verification, there can be many artifacts to be synthesized, such as inductive invariants, abstractions, or environment assumptions. Each such artifact is an element of a set $\mathcal{C}_S$. The specification for each such "synthesis sub-task", generating a different kind of artifact, is different. For invariant generation, $\mathcal{C}_S$ is the set of candidate invariants, and the specification is that the artifact $c \in \mathcal{C}_S$ be an inductive invariant of the system S. For abstractions, $\mathcal{C}_S$ defines the set of abstractions, and the specification is that $c \in \mathcal{C}_S$ must be a sound *and* precise abstraction with respect to the property to be verified, Φ; here "precise" means that no spurious counterexamples will be generated. We will use Ψ to denote the cumulative specification for all synthesis sub-tasks in the verification problem.

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

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

where, for synthesis, $\Psi = \Phi$, and, for verification, Ψ denotes the cumulative specification for the synthesis sub-tasks, as discussed

above.

Similarly, the existence of an artifact to be synthesized that additionally 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 formula $\texttt{valid}(\mathcal{H})$ given below:

$$\texttt{valid}(\mathcal{H}) \triangleq (\exists c \in \mathcal{C}_S \,.\, c \models \Psi) \implies (\exists c \in \mathcal{C}_{\mathcal{H}} \,.\, c \models \Psi) \quad (1)$$

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. Indeed, one extremely effective technique in verification, counterexample-guided abstraction refinement (CEGAR), can be seen as a form of sciduction where the latter case holds (Sec. 2.4.1 has a more detailed discussion of the link between CEGAR and sciduction.) However, in some cases, $\texttt{valid}(\mathcal{H})$ can be proved valid even without these cases; see Sec. 3.1 for an example.

2.3.2 Conditional Soundness

A verification/synthesis procedure following the sciduction 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 soundess requirement for $\mathcal{P}$ can be expressed as the following logical expression:

$$\texttt{valid}(\mathcal{H}) \implies \texttt{sound}(\mathcal{P}) \quad (2)$$

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.

2.4 Context and Previous Work

In both ancient and modern philosophy, there is a long history of arguments about the distinction between induction and deduction and their relationship and relative importance. This literature, although very interesting, is not directly relevant to the discussion in this paper.

Within computer science and engineering, the field of artificial intelligence (AI) has studied inductive and deductive reasoning and their connections (see, e.g., [41]). As mentioned earlier, Mitchell [34] 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 [19] 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 sciduction approach. Below, we highlight a representative sample of related work; this sample is intended to be illustrative, not exhaustive.

2.4.1 Instances of Sciduction

We first survey prior work in verification and synthesis that has provided inspiration for formulating the sciductive approach. Sciduction 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). In CEGAR [14], depicted in Fig. 3, the key problem is to synthesize an abstract model so as to eliminate spurious counterexamples. CEGAR solves a synthesis sub-task of generating abstract models

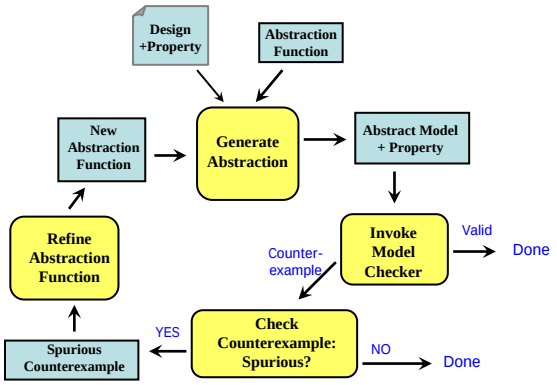

Figure 3: Counterexample-guided abstraction refinement (CEGAR).

that are *sound* (they contain all behaviors of the original system) and *precise* (any counterexample for the abstract model is also a counterexample for the original system). The synthesized artifact is thus the abstract model. CEGAR has been successfully applied to hardware [14], software [4], and hybrid systems [13]. One can view CEGAR as an instance of sciduction 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 [26], 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 [34], although the traditional CEGAR refinement algorithms are somewhat different from the learning algorithms proposed in the version spaces framework. Gupta, Clarke, et al. [21] have previously observed the link to inductive learning and have proposed version 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.

In CEGAR, usually the original system is in $\mathcal{C}_{\mathcal{H}}$, and since it is a sound and precise 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.

There are other counterexample-guided techniques that are also instances of sciduction. Programming by sketching is a novel approach to synthesizing software by encoding programmer insight in the form of a *partial program*, or "sketch" [49]. An algorithmic approach central to this work is counterexample-guided inductive synthesis (CEGIS) [49, 48], which is analogous to CEGAR. The structure hypothesis is the sketch, and the inductive and deductive procedures are similar to those in CEGAR.

Invariant Generation. One of the important steps in verification based on model checking or theorem proving is the construction of *inductive invariants*. (Here "inductive" refers to the use of mathematical induction.) One often needs to strengthen the main safety property with auxiliary inductive invariants so as to succeed at proving/disproving the property.

In recent years, an effective approach to generating inductive invariants is to assume that they have a particular structural form, use simulation to prune out candidates, and then use a SAT solver or model checker to prove those candidates that remain. This is an instance of the sciduction approach, and is very effective. For example, these strategies are implemented in the ABC verification and synthesis system [8] and described in part in Michael Case's PhD thesis [11]. The structure hypothesis $\mathcal{H}$ defines the space of candidate invariants as being either constants (literals), equivalences, implications, or in some cases, random clauses or based on k-cuts in the and-inverter graph. The inductive inference engine is very rudimentary: it just keeps all instances of invariants that match $\mathcal{H}$ and are consistent with simulation traces. The deductive engine is a SAT solver. Clearly, in this case, the structure hypothesis is restrictive in that the procedure does not seek to find arbitrary forms of invariants. However, the verification procedure is still sound, because if a suitable inductive invariant is not found, one may fail to prove the property, but a buggy system will not be deemed correct.

This idea has also been explored in software verification, by combining the Daikon system [17] for generating likely program invariants from traces with deductive verification systems such as ESC/Java [18].

Learning for Compositional Verification. The use of learning algorithms has been investigated extensively in the context of synthesizing environment assumptions for compositional verification. Most of these techniques are based on Angluin's L^* algorithm [2] and its variants; see [16] for a recent collection of papers on this topic. These techniques are an instance of sciduction in the following sense. The artifact being synthesized is an environment model in the form of a state machine. For finite-state model checking, one typically assumes that the environment is also a finite-state transducer whose inputs and outputs are defined by those of the system. The structure hypothesis is thus not restrictive, i.e., $\mathcal{C}_\mathcal{H} = \mathcal{C}_E$. The L^* algorithm is a learning algorithm based on queries and counterexamples. The counterexamples are generated by the model checker, which forms the deductive procedure in this case.

2.4.2 Other Related Work

Program Analysis Using Relevance Heuristics. McMillan [33] describes the idea of verification based on "relevance heuristics", which is the notion that facts useful in proving special cases of the verification problem are likely to be useful in general. This idea is motivated by the similar approach taken in (CDCL) SAT solvers. A concrete instance of this approach is interpolation-based model checking [32], where a proof of a special case (e.g., the lack of an assertion failure down a certain program path) is used to generate facts relevant to solving the general verification problem (e.g., correctness along all program paths). Although this work generalizes from special cases, the generalization traditionally is not inductive, and no structure hypothesis is involved.

However, one might note the possibility of using inductive inference in the generalization step as follows: since an interpolant is a specific type of logical formula that is consistent with states reachable at a particular program point (positive examples), but inconsistent with the particular path extension from that point generated by the model checker (negative examples). Indeed, this is the approach taken very recently by Sharma et al [47]. This variant, based on inductive inference, can be viewed as an instance of sciduction, where the structure hypothesis is a particular assumption on the syntactic form of the interpolant (which is the artifact being synthesized).

Automata-Theoretic Synthesis from Linear Temporal Logic (LTL). One of the classic approaches to synthesis is the automata-theoretic approach for synthesizing a finite-state transducer (FST) from an LTL specification, pioneered by Pnueli and Rosner [37]. The approach is a purely deductive one, with a final step that involves solving an emptiness problem for tree automata. No structure hypothesis is made on the FST being synthesized. Although advances have been made in the area of synthesis from LTL, for example in special cases [36], some major challenges remain: (i) writing complete specifications is tedious and error-prone, and (ii) the computational complexity for general LTL is doubly-exponential in the size of the specification. It would be interesting to explore if inductive techniques can be combined with existing deductive automata-theoretic procedures to form an effective sciduction-based approach to some class of systems or specifications.

Verification-Driven Software Synthesis. Srivastava et al. [50] have proposed a verification-driven approach to synthesis (called VS3), where programs with loops can be synthesized from a scaffold comprising of a logical specification of program functionality, and domain constraints and templates restricting the space of synthesizable programs. The latter is a structure hypothesis. However, the approach is not sciduction since the synthesis techniques employed are purely deductive in nature.

3. NEW INSTANCES OF SCIDUCTION

In this section, we discuss, in somewhat more depth, two newer applications of sciduction. The first of these, controller synthesis for hybrid systems (Sec. 3.1), tackles the problem of high complexity of the underlying decision problem. The second, timing analysis of embedded software (Sec. 3.2), tackles the difficulty of environment modeling.

3.1 Controller Synthesis

We present here a new approach to the synthesis of *switching logic* in hybrid systems [23]. It differs from CEGAR in that $\mathcal{C}_\mathcal{H} \subset \mathcal{C}_S$; yet, there are reasonable conditions under which the structure hypothesis is valid. Since the problem area might be a unfamiliar to many readers in the EDA community, we provide more background for this problem.

Many embedded and control systems are conveniently modeled as multi-modal dynamical systems (MDSs). An MDS is a physical system (also known as a "plant") that can operate in different modes. The dynamics of the plant in each mode is known, and is usually specified using a continuous-time model such as a system of ordinary differential equations (ODEs). However, to achieve safe and efficient operation, it is typically necessary to switch between the different operating modes using carefully construced *switching logic*: guards on transitions between modes. The MDS along with its switching logic consitutes a *hybrid system*. Manually designing switching logic so as to ensure that the hybrid system satisfies its specification can be tricky and tedious.

While several techniques for switching logic synthesis have been proposed (see [23] for a survey), it remains quite challenging to handle systems with a combination of rich discrete structure (in the form of multiple modes) and complex non-linear dynamics within modes. One representative switching logic synthesis problem is for

ensuring that specified *safety* properties are satisfied. More precisely, the problem is stated as follows:

⟨SLS⟩ Given a safety property, a multimodal dynamical system (MDS), and a set of initial states, synthesize switching logic for the MDS so that the resulting hybrid system is safe.

There are no constraints on the intra-mode continuous dynamics in the MDS, other than it be deterministic and locally Lipschitz at all points [23].

An example of such a switching logic synthesis problem is the 3-gear automatic transmission system [28] depicted in Figure 4 as a hybrid automaton [1]. This example has seven modes. The transitions between modes are labeled with guard variables: g_{ij} labels the transition from Mode i to Mode j. Such a guard is termed an *entry guard* for Mode j and an *exit guard* for Mode i.

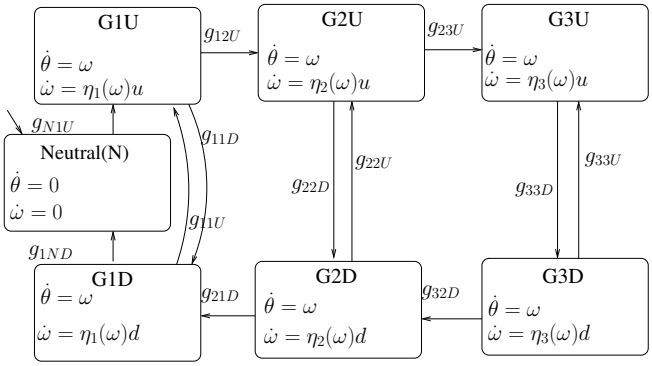

Figure 4: Automatic Transmission System

Note that for this example, the dynamics in each mode are *nonlinear differential equations*. u and d denote the throttle in accelerating and deaccelerating mode. The transmission efficiency η is η_i when the system is in the ith gear, given by:

$$\eta_i = 0.99e^{-(\omega - a_i)^2/64} + 0.01$$

where $a_1 = 10, a_2 = 20, a_3 = 30$ and ω is the speed. The distance covered is denoted by θ. The acceleration in mode i is given by the product of the throttle and transmission efficiency.

The synthesis problem is to find the guards between the modes such that the efficiency η is high for speeds greater than some threshold, that is, $\omega \geq 5 \Rightarrow \eta \geq 0.5$. Also, ω must be less than an upper limit of 60. So, the safety property ϕ_S to be enforced would be

$$(\omega \geq 5 \Rightarrow \eta \geq 0.5) \wedge (0 \leq \omega \leq 60)$$

Note that for the class of hybrid automata with nonlinear dynamics within modes, even reachability analysis is undecidable. Synthesizing safe switching logic is therefore undecidable too, unless additional assumptions are imposed. While one cannot expect to have a synthesis procedure that works in all cases, finding safe switching logic is a non-intuitive task that can be tricky and involve a lot of trial-and-error for human designers to get right.

Instance of sciduction. A new approach to the ⟨SLS⟩ problem has been presented by the author and colleagues (Jha et al. [23]). This approach is as an instance of sciduction, as follows:

- *Structure Hypothesis:* A particular syntactic form is imposed on the guards of the hybrid system: they are *hyperboxes*. More precisely, the structure hypothesis includes the following two properties:
  1. The safe switching logic, if one exists, has all guards as n-dimensional hyperboxes with vertices lying on a known discrete grid.[4]
  2. For each mode, if all exit guards and all but one entry guard are fixed as hyperboxes, then for the remaining entry transition to that mode, the safe switching states constitute a hyperbox on the above-mentioned discrete grid.

  While the above structure hypothesis may not be valid for general multi-modal dynamical systems, it can be proved valid under two additional properties: (i) the continuous dynamics within a mode is such that state variables vary *monotonically* within a mode [23], and (ii) the discrete grid reflects the finite-precision with which values of continuous system variables can be recorded. These are reasonable assumptions that hold in many embedded control systems.

  To summarize, $\mathcal{C}_{\mathcal{H}}$ is the set of all hybrid automata in which the guards satisfy the above structure hypothesis.
- *Inductive Inference:* This routine is an algorithm to learn hyperboxes in $\mathbb{R}^n$ from labeled examples. An example is a point in $\mathbb{R}^n$. Its label is positive if the point is inside the box, and negative otherwise.

  The main idea is to view safe switching states as positive examples and unsafe switching states as negative examples. Jha et al. [23] show how, given such labels, one can learn hyperboxes using the results of Goldman and Kearns [20]. The positive/negative labels on states, required by the inductive routine, are generated by a deductive engine, as described below.
- *Deductive Reasoning:* In order to label a switching state s for a mode m as safe or unsafe, we need a procedure to answer the following question: if we enter m in state s and follow its dynamics, will the trajectory visit only safe states until some exit guard becomes true?

  This is a reachability analysis problem for purely continuous systems modeled as a system of ordinary differential equations (ODEs) with a single initial state. This problem is known to be undecidable in general [40]. However, in practice, this reachability problem can be solved for many kinds of continuous dynamical systems (including the intra-mode dynamics for the example shown in Fig. 4) using state-of-the-art techniques for *numerical simulation*. Thus, the deductive engine in this approach is a numerical simulator that can handle the dynamics in each mode of the multi-modal dynamical system. The numerical simulator must be *ideal*, in that it must always return the correct YES/NO answer to the above reachability question. Since this reachability problem is a strict special case of reachability for the entire hybrid systems model, the deductive engine is "lightweight" as per the requirement in Sec. 2.

  The reader might wonder why a numerical simulator is termed as a deductive engine. Indeed, on the surface a numerical simulator seems quite different from a deductive theorem prover. However, on closer inspection one finds that both procedures employ similar deductive reasoning: they both solve systems of constraints using axioms about underlying theories and rules of inference, and they both involve the use of rewrite and simplification rules.

The overall approach of Jha et al. [23] operates within a fixpoint computation loop that initializes each guard with an overapproximate hyperbox, and then iteratively shrinks entry guards using the hyperbox learning algorithm that selects states, queries the simula-

[4] Recall that a hyperbox corresponds to a conjunction of interval constraints over the continuous variables. The requirement for the vertices of the hyperbox to lie on a discrete grid is equivalent to requiring the constant terms in the hyperbox to be rational numbers with known finite precision.

tor for labels, and then infers a smaller hyperbox from the resulting labeled states. They show that their approach can efficiently synthesize safe switching logic for many kinds of systems with monotonic continuous dynamics [23], including the automotive transmission system shown in Fig. 4.

3.2 Timing Analysis of Sofware

The analysis of quantitative properties, such as bounds on timing and power, is central to the design of reliable embedded software and systems. Fundamentally, such properties depend not only on program logic, but also on details of the program's environment. The environment includes several elements — the processor, characteristics of the memory hierarchy, the operating system, the network, etc. Moreover, in contrast with many other verification problems, the environment must be modeled with a relatively high degree of precision. Most state-of-the-art approaches to worst-case execution time (WCET) analysis employ significant manual modeling, which can be tedious, error-prone and time consuming, even taking several months to create a model of a relatively simple microcontroller. See [42] for a more detailed description of the challenges in quantitative analysis of software.

There are several variants of the timing analysis problem. We will consider the following representative timing analysis problem in this paper:

⟨TA⟩ Given a terminating program P, its execution platform (environment) E, and a fixed starting state of E, is the execution time of P on E always at most τ?

If the execution time can exceed τ, it is desirable to obtain a test case (a state of P) that shows how the bound of τ is exceeded.

The main challenge in solving this problem, as noted earlier, is the generation of a model of the environment E. While the problem statement ⟨TA⟩ includes E, complete details of the environment may not be available due to intellectual property issues. Even if a complete description of E is available, it would require substantial abstraction to facilitate timing analysis.

Additionally, the complexity of the timing analysis arises from two dimensions of the problem: the *path dimension*, where one must find the right computation path in the task, and the *state dimension*, where one must find the right (starting) environment state to run the task from. Moreover, these two dimensions interact closely; for example, the choice of path can affect the impact of the starting environment state.

We show in the next section that sciduction offers a promising approach to address this challenge of environment modeling.

Instance of sciduction: GAMETIME**.** Automatic inductive inference of models offers a way to mitigate the challenge of environment modeling. We have created a new approach, termed GAMETIME [46, 45, 44], in which a *program-specific timing model* of the platform is inferred from observations of the program's timing that are automatically and systematically generated. The program-specificity is an important difference from traditional approaches, which seek to manually construct a timing model that works for *all* programs one might run on the platform. GAMETIME only requires one to run end-to-end measurements on the target platform, making it easier to port to new platforms. The GAMETIME approach, along with an exposition of theoretical and experimental results, including comparisons with other methods, is described in existing papers [46, 45, 44]. We only give a brief summary here to describe how it is an instance of sciduction.

The central idea in GAMETIME is to view the platform as an adversary that controls the choice and evolution of the environment state, while the tool has control of the program path space. The problem is then a formulated as a game between the tool and the platform. GAMETIME uses a sciductive approach to solve this game based on the following elements:

- *Structure hypothesis:* The platform E is modeled as an adversarial process that selects weights on the edges of the control-flow graph of the program P in two steps: first, it selects the path-independent weights w, and then the path-dependent component π. The edge weights represent execution times of the basic blocks corresponding to those edges. Formally, w cannot depend on the program path being executed, whereas π is drawn from a distribution which is a function of that path. Both w and π are elements of $\mathbb{R}^m$, where m is the number of edges in the CFG after unrolling loops and inlining function calls. We term w as the *weight* and π as the *perturbation*, and the structure hypothesis as the *weight-perturbation model*.
  More specifically, the structure hypothesis $\mathcal{H}$ used by GAMETIME for problem ⟨TA⟩ is to define the space of environment models $\mathcal{C}_\mathcal{H}$ to be all processes that select a pair (w, π) every time the program P runs, where additionally the pair satisfies the following constraints (see [45] for details):
  C1: The mean perturbation along any path is bounded by a quantity $\mu_{\max}$.
  C2: The worst-case (longest) path is the unique longest path by a specified margin ρ.
  In general, this structure hypothesis is restrictive: i.e., $\mathcal{C}_\mathcal{H} \subset \mathcal{C}_E$, where $\mathcal{C}_E$ is the space of all environments. The restrictions come from conditions C1 and C2, since any environment can be viewed as picking times for basic blocks in terms of a path-independent component w and a path-dependent component π. The implications of C1 and C2 will be discussed later.
- *Inductive inference:* The inductive inference routine is a learning algorithm that operates in a *game-theoretic online setting*. The task of this algorithm is to learn the (w, π) model from measurements. The idea in GAMETIME is to measure execution times of P along so-called *basis paths*; these paths are those that form a basis for the set of all paths, in the standard linear algebra sense of a basis. GAMETIME chooses amongst these basis paths uniformly at random over a number of trials, recording the length (time) for each one. A (w, π) model is inferred from the end-to-end measurements of program timing along each of the basis paths.
- *Deductive reasoning:* Timing measurements for basis paths constitute the examples for the inductive learning algorithm. These examples are generated using SMT-based test generation. More precisely, an SMT solver combined with an integer linear programming (ILP) engine generates a set of feasible basis paths. The ILP engine generates candidate basis paths. From each candidate basis path, an SMT formula is generated such that the formula is satisfiable if and only if the path is feasible. Thus, the deductive procedure for GAMETIME is the SMT solver combined with the ILP solver. The procedure is lightweight, as it solves an NP-hard problem, whereas the overall timing analysis is PSPACE-hard, requiring reachability analysis for the composition of the program with an abstract timing model of the platform, typically a finite-state system.

The operation of GAMETIME is sketched in Figure 5. As shown in the top-left corner, the process begins with the generation of the control-flow graph (CFG) corresponding to the program (possibly at the binary level), where all loops have been unrolled to a maximum iteration bound, and all function calls have been inlined into the top-level function. The CFG is assumed to have a single source node (entry point) and a single sink node (exit point); if not, dummy source and sink nodes are added. A feasible set of basis paths (along with corresponding test cases) are then extracted using a combination of ILP and SMT solving. This is the main deductive step of GAMETIME. The program is then compiled for the target platform, and executed on these test cases. In the basic GAMETIME algo-

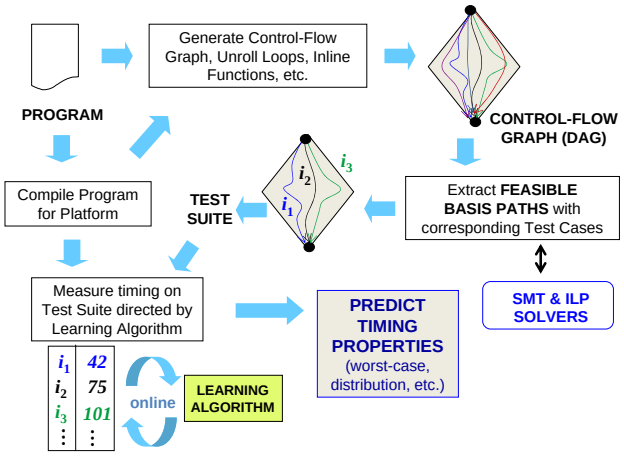

Figure 5: GAMETIME **overview**

rithm (described in [46, 45]), the sequence of tests is randomized, with basis paths being chosen uniformly at random to be executed. The overall execution time of the program is recorded for each test case. From these end-to-end execution time measurements, GAMETIME's learning algorithm — the inductive inference engine — generates the (w, π) model that can then be used for timing analysis. In contrast with most existing tools for timing analysis (see, e.g., [39]), GAMETIME can not only be used for WCET estimation, it can also be used to predict execution time of arbitrary program paths, and certain execution time statistics (e.g., the distribution of times). For example, to answer the problem $\langle \text{TA} \rangle$ presented in the preceding section, GAMETIME would predict the longest path, execute it to compute the corresponding timing τ^*, and compare that time with τ: if $\tau^* \leq \tau$, then GAMETIME returns "YES", otherwise it returns "NO" along with the corresponding test case.

Theoretical Guarantees and Empirical Results. Assuming the structure hypothesis holds, GAMETIME answers the timing analysis question $\langle \text{TA} \rangle$ with high probability. In other words, if the structure hypothesis is valid, GAMETIME is *probabilistically sound and complete* in the following sense:

> Given any $\delta > 0$, if one runs a number of tests that is polynomial in $\ln \frac{1}{\delta}$, $\mu_{\max}$, and the program parameters, GAMETIME will report the correct YES/NO answer to Problem $\langle \text{TA} \rangle$ with probability at least $1 - \delta$.

See the theorems in [46, 45] for details. Every part of the structure hypothesis, with the possible exception of condition C2, is valid in practice. Condition C1 holds since the mean pertubation cannot be unbounded for physical platforms. However, C2 must be validated; this has been done experimentally.

Experimental results indicate that in practice GAMETIME can accurately predict not only the worst-case path (and thus the WCET) but also the distribution of execution times of a task from various starting environment states. These results have been obtained on pipelined processors with instruction and data caches as well as branch prediction [45].

4. DISCUSSION AND FUTURE DIRECTIONS

This paper posits that sciduction, a tight integration of induction and deduction with structure hypothesis, is a promising approach to addressing challenging problems in formal verification and synthesis. Our proposal seeks to mirror the approach a human might take to a verification or design problem, by combining inductive reasoning with systematic deductive processes. Some of the recent successes in the formal methods area can be seen as instances of sciduction. The structure hypothesis provides a way for a human to provide creative input into the verification process without getting mired in tedious details.

We conclude with a discussion of how the ideas herein can be applied to other problems, and outline directions for future work.

4.1 Insights for Verification and Synthesis

How can one apply sciduction to a new problem in verification or synthesis?

At present, we feel that more experience is needed before particular combinations of induction and deduction are deemed more useful than others. However, one can use a general prescription for applying sciduction in cases where a purely deductive approach falls short, as follows:

1. Identify the hard synthesis sub-task(s) within the overall synthesis or verification problem.
2. Formalize suitable structure hypotheses, if needed, for each sub-task.
3. For each sub-task, devise a top-level inductive or deductive procedure to solve it under the structure hypothesis. This procedure may in turn invoke other inductive or deductive procedures: to generate examples for learning, synthesize candidate artifacts, verify properties of synthesized artifacts, etc.
4. Prove the validity of the structure hypotheses, ideally theoretically, otherwise empirically.

In addition to the work described in Sec. 3, we have applied sciduction in other settings. For example, a major challenge for automatic synthesis from linear temporal logic (LTL) is in writing complete and consistent specifications, of which the environment assumptions are a large part. In recent work [27], we have demonstrated that environment assumptions can be mined from traces and counter-strategies. We have also recently used a combination of induction on decision trees (see [34]) and SMT-based ("term-level") model checking using UCLID [10] to perform conditional term-level abstraction of register-transfer level (RTL) designs [7]. We have used sciduction in the automatic synthesis of loop-free programs, with applications to synthesizing high-performance code and deobfuscating malware [22]. Finally, in the area of controller synthesis for hybrid systems, initial results have been obtained on synthesizing switching logic for optimality, rather than just safety [24].

4.2 Future Directions

There are several directions for future work.

First, recall that the soundness guarantees of sciduction only hold when the structure hypothesis is valid. When this is not trivially true (e.g., $\mathcal{C}_\mathcal{H} = \mathcal{C}_S$), one has to use other properties of the problem, such as monotonicity of dynamics in the hybrid systems problem of Sec. 3.1. It would be useful to develop a general, systematic approach for checking the validity of the hypothesis $\mathcal{H}$.

Second, we note that sciduction 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. Similarly, how can one effectively use deductive engines as oracles in learning algorithms? 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 sciductive approach to them. As an example, consider again the problem of synthesis from LTL specifications. Another challenge for this problem is to deal with the doubly-exponential computational com-

plexity. It would be interesting to see if the synthesis algorithms themselves can be made more scalable using sciduction. Another direction is to generalize the ideas used for timing analysis to other quantitative properties of cyber-physical systems, and also for verification problems at the hardware-software interface ("hardware-software verification"). In both settings, generating environment models can be quite challenging, and, from our experience with timing analysis, it appears that sciduction can be effectively brought to bear on these problems.

Acknowledgments

This article is a result of ideas synthesized and verified (!) over the last few years in collaboration with several students and colleagues. The contributions of Susmit Jha, in particular, are gratefully acknowledged. This paper has benefited from feedback on talks on this work given by the author at several venues in 2009-11. This work has been supported in part by several sponsors including the National Science Foundation (CNS-0644436, CNS-0627734, and CNS-1035672), 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) and MultiScale Systems Center (MuSyC), two of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

5. REFERENCES

[1] 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*, 138(1):3–34, February 1995.

[2] D. Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.

[3] D. Angluin and C. H. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys*, 15:237–269, Sept. 1983.

[4] 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)*, pages 203–213, June 2001.

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

[6] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. *Formal Methods in System Design*, 18(2):141–162, 2001.

[7] 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)*, pages 116–124, October 2011.

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

[9] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[10] 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 E. Brinksma and K. G. Larsen, editors, *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.

[11] M. Case. *On Invariants to Characterize the State Space for Sequential Logic Synthesis and Formal Verification*. PhD thesis, EECS Department, UC Berkeley, Apr 2009.

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

[13] 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*, pages 192–207, 2003.

[14] 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)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

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

[16] Dimitra Giannakopoulou and Corina S. Pasareanu, eds. Special issue on learning techniques for compositional reasoning. *Formal Methods in System Design*, 32(3):173–174, 2008.

[17] M. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, Seattle, 2000.

[18] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *SIGPLAN Notices*, 37:234–245, May 2002.

[19] H. Fox. *Agent problem solving by inductive and deductive program synthesis*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2008.

[20] S. A. Goldman and M. J. Kearns. On the complexity of teaching. *Journal of Computer and System Sciences*, 50:20–31, 1995.

[21] A. Gupta. *Learning Abstractions for Model Checking*. PhD thesis, Computer Science Department, Carnegie Mellon University, 2006.

[22] 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)*, pages 215–224, 2010.

[23] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Synthesizing switching logic for safety and dwell-time requirements. In *Proceedings of the International Conference on Cyber-Physical Systems (ICCPS)*, pages 22–31, April 2010.

[24] 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)*, pages 107–116, October 2011.

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

[26] R. Kurshan. Automata-theoretic verification of coordinating processes. In *11th International Conference on Analysis and Optimization of Systems – Discrete Event Systems*, volume 199 of *LNCS*, pages 16–28. Springer, 1994.

[27] 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.

[28] J. Lygeros. Lecture notes on hybrid systems. 2004.

[29] S. Malik and L. Zhang. Boolean satisfiability: From theoretical hardness to practical success. *Communications of the ACM (CACM)*, 52(8):76–82, 2009.

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

[31] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM TOPLAS*, 2(1):90–121, 1980.

[32] K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. 15th International Conference on Computer-Aided Verification (CAV)*, pages 1–13, July 2003.

[33] K. L. McMillan. Relevance heuristics for program analysis. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 145–146. ACM Press, 2008.

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

[35] S. Muggleton and L. de Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19-20(1):629–679, 1994.

[36] 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)*, volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2006.

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

[38] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, number 137 in LNCS, pages 337–351, 1982.

[39] Reinhard Wilhelm et al. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 2007.

[40] K. Ruohonen. Undecidable event detection problems for ODEs of dimension one and two. *Informatique Théorique et Applications*, 31(1):67–79, 1997.

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

[42] S. A. Seshia. Quantitative analysis of software: Challenges and recent advances. In *Proc. Formal Aspects of Component Software (FACS)*, 2010.

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

[44] S. A. Seshia and J. Kotker. GameTime: A toolkit for timing analysis of software. In *Proc. Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, 2011.

[45] S. A. Seshia and A. Rakhlin. Quantitative analysis of systems using game-theoretic learning. *ACM Transactions on Embedded Computing Systems (TECS)*. To appear.

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

[47] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. Technical Report MSR-TR-2012-13, Microsoft Research, January 2012.

[48] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

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

[50] 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)*, pages 313–326, 2010.