Formal Methods in System Design (2022) 60:426–451
https://doi.org/10.1007/s10703-023-00430-1

Machine learning and logic: a new frontier in artificial intelligence

Vijay Ganesh[1] · **Sanjit A. Seshia**[2] · **Somesh Jha**[3]

Received: 17 October 2022 / Accepted: 5 May 2023 / Published online: 14 June 2023

Abstract
Machine learning and logical reasoning have been the two foundational pillars of Artificial Intelligence (AI) since its inception, and yet, until recently the interactions between these two fields have been relatively limited. Despite their individual success and largely independent development, there are new problems on the horizon that seem solvable only via a combination of ideas from these two fields of AI. These problems can be broadly characterized as follows: how can learning be used to make logical reasoning and synthesis/verification engines more efficient and powerful, and in the reverse direction, how can we use reasoning to improve the accuracy, generalizability, and trustworthiness of learning. In this perspective paper, we address the above-mentioned questions with an emphasis on certain paradigmatic trends at the intersection of learning and reasoning. Our intent here is not to be a comprehensive survey of all the ways in which learning and reasoning have been combined in the past. Rather we focus on certain recent paradigms where *corrective feedback loops* between learning and reasoning seem to play a particularly important role. Specifically, we observe the following three trends: first, the use of learning techniques (especially, reinforcement learning) in sequencing, selecting, and initializing proof rules in solvers/provers; second, combinations of inductive learning and deductive reasoning in the context of program synthesis and verification; and third, the use of solver layers in providing corrective feedback to machine learning models in order to help improve their accuracy, generalizability, and robustness with respect to partial specifications or domain knowledge. We believe that these paradigms are likely to have significant and dramatic impact on AI and its applications for a long time to come.

Keywords Combinations of learning and reasoning · Learning for solvers · Learning for verification and synthesis · Solver layers in deep neural networks

1 Introduction

Artificial Intelligence (AI), since its inception, has had two major sub-fields, namely, logical reasoning and machine learning (ML) [31]. Each of these topics has had transformative positive impact on many fields of science, engineering, and business [9]. For example, in

Vijay Ganesh, Sanjit A. Seshia, Somesh Jha have contributed equally to this work.

Extended author information available on the last page of the article recent years learning techniques have effectively addressed longstanding problems within the context of computer vision, natural language processing, and game-playing at an elite level [12, 41, 75]. Similarly, we have witnessed a revolution in software engineering, program analysis, and verification thanks to logical reasoning tools such as SAT/SMT/CP solvers [10].

Despite remarkable and independent progress in both reasoning and learning, it is fair to say that the interactions between the two fields have been rather limited over the years [9, 31]. This is not to say that there are no interactions at all. Many methodologies for combining inductive learning and deductive reasoning have been developed in recent years [80]. For example, techniques have been proposed for injecting knowledge in learning [26]. Another example is Inductive Logic Programming (ILP) that aims to learn models or hypotheses from positive/negative examples and background knowledge, expressed in first-order logic [18, 74]. Yet another approach is neuro-symbolic AI, a broad area whose goal is to “integrate ML and reasoning” [89]. However, despite these attempts to bring reasoning and learning closer together over the years, we still are at an early stage of development in topics at the intersection of reasoning and learning.

Fortunately, over the last decade there is a recognition among longtime AI observers and researchers that there are new problems on the horizon that seem solvable only via combinations of learning and reasoning [9]. As a consequence, we are now witnessing an intensification of interactions between these two areas.

These include emerging fields such as trustworthy, secure, privacy-preserving, explainable, interpretable and reliable ML, where logical specification languages and automated reasoning tools such as SAT, SMT, CP, and MILP solvers will likely play an important role [87, 93]. For example, a completely new class of solvers has recently been developed with the goal of verifying ML models and there is now an annual competition called VNNCOMP to benchmark these solvers on large real-world neural networks [13], exemplifying the use of reasoning for learning.

In the opposite direction, it is now apparent that ML has a key role to play in formal reasoning of programs (i.e., use of ML in program synthesis, analysis, and verification), as well as in enabling logical reasoning tools become more efficient and powerful (i.e., the use of ML in solvers and provers). For example, there is an increasing use of Reinforcement Learning (RL) techniques alongside logical reasoning methods in the context of proving mathematical conjectures [15], as well as the continued use of ML in SAT/SMT/CP solvers in enabling them to become more efficient and effective for a variety of applications in software engineering, security, cryptanalysis, combinatorial mathematics, and AI [56, 59, 60].

Additionally, a new class of ML models are being developed that use a solver, as a logic or reasoning layer, during training and/or inference. Here we are referring to recent work such as SATnet, that combines ML models with differentiable solvers [92], the use of combinatorial blackbox solvers in Deep Neural Networks (DNNs) [72], as well as Logic Guided Machine Learning (LGML) [84], and its cousin Logic Guided Genetic Algorithms (LGGA) [7].

1.1 Goal and scope of this paper

Given these interactions between learning and reasoning over the last few decades, both shallow and deep, and the recent intensification of such interactions, it is but natural to ask “are there any identifiable common themes or lessons that have the potential for long-term impact on AI and applications?" In an attempt to answer this question, it is tempting to write a deep and comprehensive survey of the field with the aim of developing a well-articulated perspective. Unfortunately, this is a near impossible task given how quickly the field is evolving and the diversity of interactions between learning and reasoning. Hence, in this perspective paper ,[1] we narrowly focus on a few topics at the intersection of learning and reasoning, where we can identify certain long-term trends.

Specifically, in this paper we focus on three areas, namely, ML techniques for solvers [29, 56, 59, 60], the use of ML for synthesis/verification of programs and proofs [80], and the very recent trend of combinations of ML and solvers for training and/or inference [7, 72, 84, 92]. In a nutshell, we observe that many, if not all, techniques in the above-mentioned three areas at the intersection of learning and reasoning have the following in common, namely, “corrective feedback loops between ML and reasoning”.

A quick recap of abstraction-refinement techniques, dominant in formal verification, synthesis, analysis, and SAT and SMT solvers, can help contextualize what we mean by corrective feedback loops. Briefly, abstraction-refinement techniques, such as Counter-Example Guided Abstraction Refinement (CEGAR) [22], Counter-Example Guided Inductive Synthesis (CEGIS) [82], and Conflict-Driven Clause Learning (CDCL) [85], take as input a formal object (e.g., a mathematical formula) and perform reasoning with it. Typically, these techniques iteratively perform an over-approximation [2] as in the case of CEGAR or an under-approximation as in the case of CDCL, and then check whether the resulting abstraction is sufficient to get the correct answer. If not, they provide a corrective feedback, referred to as a “refinement" and iterate.

These abstractions and refinements can take many forms, and there is a vast literature on them in verification and synthesis contexts [10, 88]. For example, in the case of the CDCL SAT solver the abstraction is an under-approximation in the form of a reduced formula R under a partial assignment A applied to the input formula F. If A satisfies F, the method terminates. Else, it refines by constructing a proof P of why A does not satisfy F, and this proof is then fed back to the abstraction, thus correcting it and preventing future mistakes similar to the under-approximation R. More precisely, proofs P guide the Boolean Constraint Propagation (BCP) and the branching heuristic in *refining* the successive under-approximations that the solver constructs in a corrective-feedback loop. This process repeats until the correct answer is determined.

Naturally, one can view many verification algorithms such as CEGAR or CDCL as consisting of two processes, an abstractor and a refiner .[3] Now, the obvious next question is the following: what has all this got to do with learning or its interaction with reasoning? A quick leap of imagination leads one to a connection between abstraction-refinement with reinforcement learning, where the abstractor corresponds to an agent and a refiner corresponds to the environment, and the refiner provides corrective-feedback to the agent. Of course, there are important differences. However, the connection is strong enough that many recent ML-based CDCL algorithms have been viewed this way, enabling a lifting of techniques from the RL setting to SAT solving [29, 56, 59]. Another example is the

[1] Given that the focus of this paper is to provide a perspective on certain emerging trends in AI, there is no experimental or scientific data associated with it.

[2] Conceptually, the terms abstraction and approximation are very similar in their technical meaning in the broad context of program analysis, synthesis, and verification. Hence, we use them interchangeably.

[3] It is possible to imagine more complex configurations with multiple kinds of abstractors and refiners, but we won’t discuss them for the sake of brevity.

above-mentioned use of solver layers in DNNs, a la SATNet, where the solver is used to provide corrective feedback to a DNN [7, 72, 84, 92]. Yet another example is the use of ML techniques to synthesize programs [80]. These techniques initially construct over-approximation of the requirements and then use verification techniques to refine, until a satisfactory program is synthesized.

1.1.1 Corrective feedback loops in learning and reasoning

We do not aim to formally define corrective feedback loops between abstractors and refiners here, but rather provide a high-level informal description. Informally, corrective feedback loops can be viewed as consisting of two processes (or sub-routines) wherein one of them can be termed an agent or abstractor (aka, a student or a synthesizer/prover) and the other an environment or refiner (sometimes referred to as a teacher/verifier), communicating with each other iteratively. The precise definitions of an agent and environment depend on context. Fortunately, all these terms are well defined in many contexts including machine learning or formal methods. At a high level, we can think of an abstractor as a function whose aim is to construct a feasible abstraction (as in an over-approximation of a formula/code or an ML model that approximates some function), while the goal of the refiner is to correct the abstraction (as in providing learnt clauses or refinement formulas or reward signal) until the abstraction is in some meaningful contextual sense indistinguishable from the ”actual” function or system-under-analysis. Put differently, the goal of the interaction between the abstractor and refiner is to minimize some objective function that makes it infeasible to distinguish the abstraction from system-under-analysis or learn a policy by maximizing rewards.

As mentioned above, corrective feedback methods are common in formal methods. Further, corrective feedback loops are widely used in machine learning, even though one may not explicitly identify them as such in the literature. For example, all of RL can be viewed as corrective feedback from the environment to the learning agent(s) via a reward mechanism with the goal of learning an optimal policy [77]. Other areas where corrective feedback loops have been studied include control theory and program synthesis. Obviously, our goal is not survey all these notions of corrective feedback loops. Rather, the goal is identify this theme of corrective feedback in the algorithms being developed at the intersection of reasoning and learning.

In a nutshell, the theme of the paper then is the following: “Many algorithms being developed at the intersection of reasoning and learning can be viewed as consisting of two processes, one of which is an ML technique and the other a reasoning engine. The ML technique is often used to construct abstractions, while reasoning techniques can be leveraged to verify and correct them in a corrective feedback loop. This class of iterative learning+reasoning algorithms enable us to solve classes of AI problems that otherwise seem very difficult or infeasible to solve.”

1.1.2 Related work

For a deep and comprehensive overview of topics at the intersection of learning and reasoning, we refer the reader to an excellent paper by a group of authors who use the rather artful pseudonym K.R. Amel [9]. The K.R. Amel paper is an invaluable resource that touches on almost every conceivable research topic at the intersection of ML and reasoning (with a strong focus on Knowledge Representation and Reasoning or KRR). The stated goal of the K.R. Amel paper is to "construct an inventory of common concerns in KRR and ML, of methodologies combining reasoning principles and learning, of examples of KRR/ML synergies". By contrast, our work is a deep dive into specific topics at the intersection of ML and reasoning, and identifying the general principle of corrective feedback loops between ML and reasoning.

For a comprehensive and recent survey on the use of ML techniques for reasoning, we strongly recommend the excellent book by Sean Holden [46]. This reference book surveys the entire field of ML for solvers starting from the late 1980s to the present, classifying various types of ML techniques used in solvers based on how they are used. For an overview of ML techniques used in solvers with a sharp focus on Constraint Programming (CP), we recommend the comprehensive survey paper by Popescu et al. [70].

Another topic at the intersection of ML and logic is verification of ML models, especially Deep Neural Networks (DNNs) [13]. Many *DNN solvers*, based on MILP solvers have been developed to verify whether DNNs are vulnerable to certain kinds of adversarial attacks. We avoid surveying this topic because it is still quite new and it is too early to tell what paradigms are likely to dominate this field in the long term. We refer the reader to the VNNCOMP and VNNLIB websites [13, 91] for additional information on this topic.

2 ML for solvers

Logical reasoners (LR) or solvers ,[4] computer programs that take as input mathematical formulas and decide whether they have solutions (i.e., are satisfiable), have long been a dominant area of AI research since the 1950s. Solvers have found application in planning [53], knowledge representation and reasoning (KRR) [90], and software engineering (broadly construed to include testing, analysis, and verification) [17, 24]. For example, SAT solvers such as MapleSAT [56] and SMT solvers such as Z3 [28] are integral to many software testing, program analysis and verification techniques.

The field of logical reasoning by itself is quite broad with many different roots, resulting in disparate communities of researchers working on a variety of mathematical software. For example, the field of Constraint Programming (CP) developed wholly as a sub-field of AI, while the fields of SAT/SMT solvers and first/higher-order theorem provers developed largely as part of formal methods and programming languages, Mixed Integer Linear Programming (MILP) solvers as part of optimization research, and Computer Algebra Systems (CAS) at the intersection of mathematics and computer science. Yet, increasingly these fields are coming closer together, where the common thread is the introduction of similar ML methods in all these symbolic reasoning systems, a theme we highlight below.

Concretely, we primarily focus on ML techniques as applied to SAT solvers, in particular how the CDCL algorithm can be viewed as an RL system, where the agent is the branching heuristic, while the corrective feedback is provided by its symbolic reasoning engine (i.e., conflict analysis). Having said that, we also very briefly survey the use of ML to predict satisfiability directly, as well as its use in Stochastic Local Search (SLS). The goal is to provide some contrast between the use of corrective feedback loops in CDCL and other uses of ML in solvers.

[4] In this paper, we interchangeably use the terms logical reasoners, symbolic reasoners, decision procedures, and solvers (dually, provers), since all such systems fundamentally reason about mathematical formulas and decide whether they are satisfiable (dually, valid).

2.1 Brief history of ML for solvers

For most of its history, logical reasoning as a field has been dominated by algorithms that implement proof systems (e.g., resolution) or are search-based (e.g., stochastic local search) or some combinations thereof (e.g., Conflict-Driven Clause-Learning). Proof systems are a very natural way to approach the question of how best to implement a solver, since proof construction and deduction are central to logical reasoning. At the same time, search methods such as stochastic local search (SLS) have also been developed and shown to be effective in some settings, especially when input formulas are randomly-generated and satisfiable. However, interestingly, until a couple of decades ago, most solver algorithms or heuristics were not ML based.

Part of the reason for a lack of early enthusiasm towards using ML methods for logical reasoning was driven by the belief that ML is unsuitable for handling symbolic data or that ML methods cannot generalize well in the context of formulas and proofs, since even a minor change to these symbolic objects completely changes their properties and thus making learning generalizations difficult. Another reason is that scalable and practical implementations and specialized hardware for ML methods were not widely available until recently, further limiting their adoption in the early days of logical reasoning.

Fortunately, over the last 30 years we have witnessed a sea change in the use of ML in logical reasoning systems, with the trend beginning as early as 1989 in the pioneering works of Ertel et al. [33] and Johnson [50]. Additionally, nearly two decades ago, the effectiveness of ML for algorithm and parameter selection for solvers was demonstrated both for SMT [78] and SAT [44]. Part of the reason for this dramatic change is the availability of a significant amount of data relating to formal objects as formulas and proofs. Solvers generate copious amounts of data as they solve formulas. Significant portions of such data have been collected and curated over the years. Another reason is the maturity of ML software and hardware infrastructural support, such as PyTorch or Tensorflow. However, arguably the most important reason for the adoption of ML in logical reasoners is due to paradigmatic shifts in our understanding of the value of ML methods in a constraint solving setting [56, 59]. Specifically, today we have a wide variety of ML methods for logical reasoning ranging from directly predicting satisfiability, algorithm selection or parameter tuning in solvers, or as optimization heuristics in implementations of proof systems. In this Section, we cover these different ways of applying ML to logical reasoning and contrast their strengths and weaknesses. We conclude this section with our view of the long term trends in the field of ML for solvers.

2.2 Background on the SAT/SMT problem and solvers

Boolean satisfiability (SAT) is one of the central problems in computer science and mathematics, at the heart of the famous P vs. NP question and is believed to be intractable in general [10]. This problem has been studied intensively both by theorists and practitioners since it was shown to be NP-complete by Stephen Cook, and independently by Leonid Levin, in 1971 [25]. The problem can be stated as follows:

Problem 1 (**The Boolean Satisfiability Problem**) Given a Boolean formula $\phi(x_1, x_2, \ldots, x_n)$ in conjunctive normal form (CNF) over Boolean variables $x_1, x_2, \ldots, x_n$, determine whether it is satisfiable. We say that a formula $\phi(x_1, x_2, \ldots, x_n)$ is satisfiable if there exists an assignment to the variables of $\phi(x_1, x_2, \ldots, x_n)$ such that the formula evaluates to true under that assignment. Otherwise, we say the formula is unsatisfiable. This problem is also sometimes referred to as CNF-SAT.

There are many variations of the SAT problem (e.g., k-CNF, CircuitSAT, etc.) that are all equivalent from a worst-case complexity perspective. By SAT, we always refer to the CNF-SAT problem, unless otherwise stated. A SAT solver is a computer program aimed at solving the Boolean satisfiability problem. The term SMT stands for Satisfiability Modulo Theories, and the SMT problem refers to the satisfiability problem for first-order theories that are particular relevant in software engineering. We refer the reader to the Handbook of Satisfiability for a more thorough treatment of SAT, SMT, and constraint programming (CP) solvers and first-order provers [10].

It goes without saying that the two properties users want from solvers are correctness (i.e., the solver declares an input formula F to be satisfiable if and only if F is indeed satisfiable) and scalability (i.e., as the size of the input formula grows, hopefully the running time of the solver does not grow exponentially). Unfortunately, due to the fact that most solvers aim to solve NP-hard problems, believed to be intractable, we do not expect any solver to scale well beyond a certain finite formula size for most formula classes. Even so, we can and do empirically compare solvers over a large and growing set of open-source benchmarks obtained from a variety of applications that includes program analysis, verification, security, AI, as well as randomly-generated k-SAT instances.

Solvers are compared annually at the SAT competition that witnesses dramatic progress each and every year [35]. Some of these solvers, such as MapleSAT and variants [56], Glucose [5] etc. routinely solve formulas with hundreds of millions of variables and clauses in them. Further, with increasing use of ML methods, solvers are able to scale new heights not witnessed even a few years ago.

2.3 Classification of ML methods for logic solvers

A rich and diverse set of ML techniques for solvers have been explored over the past few decades. For example, one could use the type of ML method used such as supervised learning or reinforcement learning. However, we choose to classify ML methods into four categories based on how they are used, i.e., to directly predict satisfiability, solution search heuristics, parameter tuning, and optimization heuristics aimed at selection, sequencing, and initialization of proof rules, tactics, and algorithms.

2.3.1 Satisfiability prediction: from formulas to SAT/UNSAT

A natural way to view the satisfiability problem for any logic L is as a classification problem, i.e., given an L-formula F, classify it as satisfiable or not. The earliest attempt in this regard was by Johnson [50]. Building on previous work by Hopfield and Tank [47] in the context of the Traveling Salesman Problem (TSP), Johnson's method defined recurrent neural network such that the associated objective function attains globally optimum values only at satisfying assignments in the associated Boolean polytope. While this was a very novel and pioneering approach at the time, the method was not particularly effective. Even for small formulas, the loss landscape proved to be hard to navigate to due to the numerous local minima it contained. Further, the method did not guarantee convergence.

In 2008, Devlin and O'Sullivan [30] used many standard supervised learning techniques to go from structural features of an input formula, such as number of variables and clauses, to directly predicting satisfiability. The motivation for such approaches is quite straightforward, namely, that supervised learning techniques have been very successful in a plethora of domains, and so it is only natural to try and predict satisfiability from large representative feature sets. A significant amount of effort goes into feature selection and engineering, since formulas possess a near endless number of features and it can be quite challenging to select a representative predictive subset.

Perhaps the most interesting new class of ML-based solvers that predict satisfiability directly are those based on Graph Neural Networks (GNNs) [11]. The most well-known among them is the NeuroSAT solver by Selsam et al. [81]. GNNs are an exciting new class of neural networks that are able to take arbitrarily large graphs as inputs, unlike Multi Layer Perceptrons (MLPs) that only accept as input fixed-sized vectors or Recurrent Neural Networks (RNNs) that accept arbitrarily long sequences of fixed-size vectors. This feature of GNNs is particularly useful in the context of solvers because formulas are easily represented as graphs. Further, due to the fact that GNN-based solvers directly accept formulas (as graphs), they completely circumvent the feature engineering problem faced by above-described methods that aim to predict satisfiability directly. Finally, GNN-based solvers have another important feature is that they respect many invariances, such as permutation invariance, that Boolean formulas possess.

Briefly, GNN-based solvers work via a message passing mechanism, similar to well-known satisfiability methods such as survey and belief propagation. The GNN-based solver, NeuroSAT, takes a bipartite graph of a Boolean formula as input, wherein the nodes of graph correspond to literals and clauses of the input formula, and there is an edge between a literal node and clause node if the literal occurs in the corresponding clause in the input formula (there are also special edges between the nodes corresponding to a variable x and the node corresponding to its complement).

The solver iteratively computes an embedding vector for each node (i.e., for both literal and clause nodes) in the graph via a series of messages passed back and forth along the edges. Every literal and clause of the input formula has an embedding at the start of an iteration. The iterations perform two consecutive message passing updates. First, each clause node receives messages from its neighbouring literal nodes and updates its embedding appropriately. Second, each literal receives messages from its neighbouring clauses, as well as its complement, and updates its embedding accordingly. Eventually, the embeddings of the variable nodes converge to a representation in $\mathbb{R}$ for each of the $2v$ literals, that corresponds to a satisfying assignment if the formula is predicted to be satisfiable or denotes that the formula is unsatisfiable.

Employing ML methods to predict satisfiability directly has indeed garnered a fair amount of attention in recent years. Unfortunately, these methods suffer from poor scalability, accuracy, and generalizability. The poor scalability of such solvers is largely due to the computational effort required for feature computation and engineering, since those features that are often the best predictor of satisfiability are also the most difficult to compute.

Even in the case of GNN-based solvers, that do not compute any features and process formulas directly as graphs, the scalability is poor due to performance issues with embedding computations in GNNs. Further, the general lack of 100% accuracy of such solvers means that they are inherently incomplete or unsound. Hence, they are not particularly useful by themselves and have to be used in conjunction with some kind of correction mechanism if one's goal is to decide satisfiability. Finally, formulas from different classes of application tend to have very different structural features, making it difficult to train satisfiability predictors that generalize well.

2.3.2 ML heuristics for stochastic local search

Stochastic Local Search (SLS) methods are a class of search-based solvers with an extensive history and have been found to be remarkably good at solving randomly-generated formulas [76]. Unlike solvers based on proof systems such as CDCL, search-based methods typically search through the space of assignments corresponding to an input formula looking for a satisfying one and are often incomplete.

Briefly, the SLS algorithm works as follows: it initially selects a candidate assignment (usually randomly) and checks if the input formula is satisfiable under this candidate assignment. If yes, the method returns that assignment, else, at least one variable's value in the assignment is flipped (i.e., true to false, or vice-versa) to obtain a modified assignment. Subsequent to that a check is performed to see whether the input formula is satisfiable under this modified assignment. This process continues until either the input formula has been decided as satisfiable or the maximum number of flips allowed by the method has been reached.

Given the simplicity of the algorithm, there are very few places in the SLS algorithm that are particularly suited for ML-based heuristics. Having said that, there are two subroutines that are ripe for ML-based heuristics and they are the selection heuristics for flipping variables and the heuristics for choosing the initial assignment.

Over the last several decades considerable research has been gone into heuristic for selecting variables to be flipped (flip heuristic). Most flip heuristics use statistics of different kinds to determine which variable to flip next. That is, they dynamically compute a variety of metrics per variable and then flip the variable that is highest in some order defined over these metrics. For example, a popular metric is the number of additional clauses satisfied by flipping a variable, instead of not flipping it, referred to as *net gain* of a variable v. Similarly, *negative gain* (resp. *positive gain*) is the number of clauses rendered unsatisfied (resp. satisfied), that were previously satisfied (resp. unsatisfied), by flipping the value of a variable v.

In a series of papers, Fukunaga [38–40], proposes the following idea: a flip heuristic can be viewed as computing some combination of a set of the above-mentioned metrics (which he refers to as primitives), such as negative or positive gain. Once viewed in this way, combinations of these primitives can be explored via Genetic Algorithms (GAs) that aim to minimize some loss in order to evolutionarily come up with a combination of primitives. The advantage of such methods is that they are online, dynamic, and adapt the heuristic to the given formula. Other researchers have also come up with similar GA approaches for adaptive heuristics [16].

A few observations are in order here. In general, SLS methods tend not to exploit the logical structure of input formulas, unlike CDCL solvers, and hence are unlikely to perform well on industrial instances. Having said that, the research on SLS methods, especially ML-based flip heuristics, is fertile ground for metrics that can be adapted to CDCL solvers. In recent years, flip statistics have been applied in the context of CDCL-based parallel portfolio and divide-and-conquer solvers [65], as well as newer SLS+CDCL solvers have been proposed [61]. A long-term trend that we see here is the leveraging of ML-based heuristics and metrics being adapted to a newer class of combinations of SLS and CDCL solvers.

2.3.3 Selecting and sequencing proof rules, tactics, and algorithms

Arguably, one of the most impactful SAT algorithm to-date is the Conflict-Driven Clause Learning method [63], which in turn is built on top of the DPLL SAT algorithm [27]. It is no exaggeration to state that the CDCL-based solvers, such as MiniSAT [32], Glucose [5], MapleSAT and variants [56], etc., have had a transformative impact on many fields including software engineering, security, and AI [10]. Not surprisingly, these solvers are a jumble of complex and intricate heuristics, making it difficult to describe them in great detail in a few short pages here. Fortunately, it is possible to describe CDCL solvers as *implementations of proof systems*. This abstraction not only dramatically simplifies our presentation, but more importantly enables us to powerfully articulate why ML methods can be effectively integrated into CDCL-based solvers.

Proof systems are a very natural way of thinking about solvers abstractly, since proof construction and deduction are central to logical reasoning. Hence, it is not surprising that many solver algorithms are based on proof systems, unlike solvers that are based on ML methods aimed at directly predict satisfiability (e.g., NeuroSAT) or stochastic local search solvers. For example, it is easy to show that the DPLL SAT algorithm, introduced in series of papers in early 1960s [27], implements the well-known tree-like resolution proof system. Also, recently Atserias et al. [3] and Pipatsrisawat et al. [69] showed that the CDCL SAT algorithm, viewed as a proof system, is *polynomially equivalent* to the resolution proof system.

More generally, proof complexity theorists have long argued for the view that solvers of all kinds, whether they be SAT, SMT, MAXSAT, QBF, CP solvers or first-order provers, are best abstracted as proof systems. This view obviously makes a lot of sense when a solver is used to establish unsatisfiability, given that one powerful way to do so is by constructing a refutational proof. Further, even when solvers determine that an input is satisfiable, they do so by constructing a proof of unsatisfiability of those parts of the search space of the input formula that are empty (i.e., do not contain any solutions).

Additionally, this view that “solvers implement proof systems” is helpful from a practical design perspective as well, particularly as an argument in favor of using ML heuristics in solvers. Starting in 2016, Liang and Ganesh along with their collaborators [56, 57, 59] articulated this idea into a coherent thesis, i.e., solver algorithms that implement proof system (e.g., CDCL) can be viewed as consisting of two disjoint sets of sub-routines, wherein, one set implements inference/proof rules (e.g., Boolean Constraint Propagation implements unit resolution and conflict analysis implements general resolution), while the other set implement heuristics that are aimed at sequencing (e.g., branching), selecting (e.g., tactic or algorithm selection), and initializing (e.g., value initialization) these proof rules with the goal of constructing optimal proofs. Further, given the abundant availability of data regarding solver behavior and input formulas, one can implement these heuristics via ML methods, leveraging the wealth of knowledge in ML-based optimization methods.

This view has led to the recent rapid development of a variety of ML-based branching [60], restart [59], initialization [29], and algorithm and tactic selection methods [71], transforming solver research into a unique field at the intersection of ML and symbolic reasoning.

We can capture the above-stated view as the following pithy slogan:

Solvers implement proof systems via a combination of symbolic reasoning rule-based methods and ML heuristics aimed at optimally sequencing, selecting and initializing

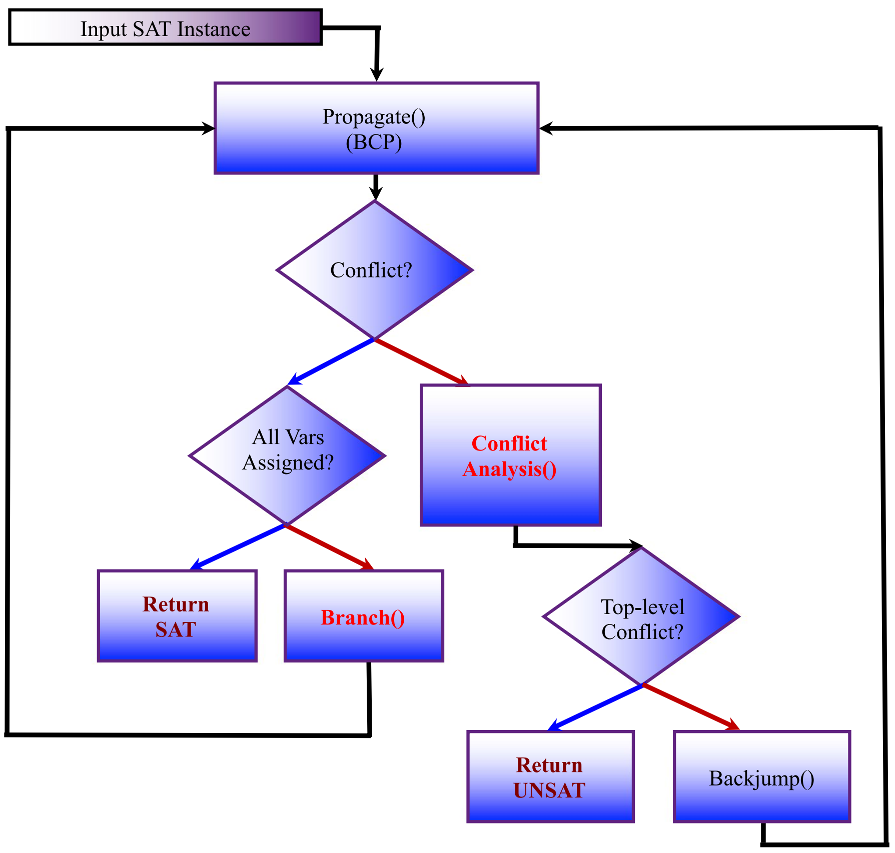

Fig. 1 A Schematic view of the Conflict-Driven Clause-Learning (CDCL) SAT Solving Algorithm. One can see that above algorithm has two distinct loops in it, one where BCP is followed by branching, and other where BCP is followed by conflict analysis. The first loop can be viewed as an agent performing actions (branching on variables), while the second can be viewed as a deductive environment that analyzes and determines why the partial assignment does not satisfy the input formula and returns a reward to the agent in terms of a conflict clause

proof rules in order to construct optimal proofs for a given input formula. A particularly effective way to do this is to view a solver's sequencing and selection heuristics as reinforcement learning agents that sequence/select rules based on the rewards they obtain by interacting with a deductive environment.

2.3.4 Conflict-driven clause learning solvers and reinforcement learning

The algorithm that best exemplifies the above-stated design principle is the modern ML-based CDCL solver (See Figs. 1 and 2) [56]. While the modern version of the CDCL algorithm is quite complex, it has been empirically established that the three sub-routines that are most important from an efficiency point of view are conflict analysis, Boolean Constraint Propagator (BCP), and the branching heuristic. The conflict analysis method, which essentially implements the general resolution proof rule, is aimed at finding the *root cause* for why assignments do not satisfy the input formula. The BCP subroutine, which

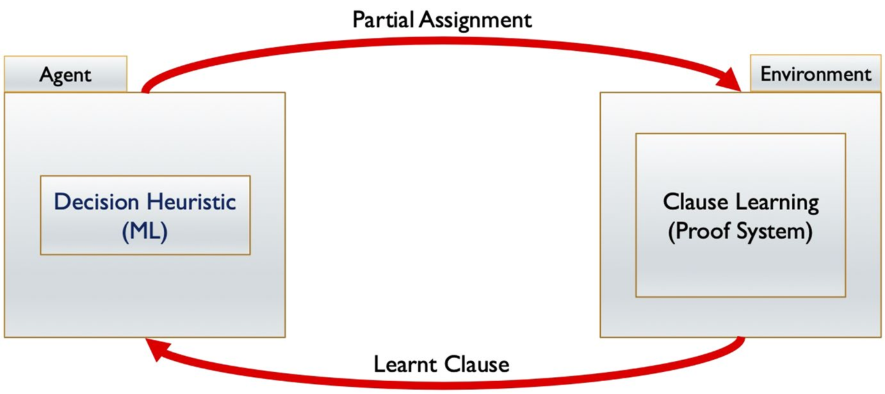

Fig. 2 Reinforcement learning view of conflict-driven clause-learning SAT solvers. In this view, the branching or decision heuristic chooses variables (agents choosing actions) resulting in unsatisfying partial assignments that trigger interactions with a clause-learning system (deductive environment) that provides corrective feedback in terms of conflict clauses (rewards)

abstractly implements unit resolution proof rule, simplifies the formula with respect to *unit clauses* and choices made by the branching heuristic. Finally, the branching heuristic is a method aimed at choosing variables from the input formula and assigning them a value. When viewed from an RL lens, the branching heuristic can be seen as a student/agent, while the conflict analysis and BCP together can be viewed as a deductive environment/teacher, with both the student and teacher interacting with each other iteratively with the goal of creating efficient proofs.

Briefly, the CDCL algorithm (Fig. 1) works as follows: upon receiving an input Boolean formula F, the CDCL algorithm first simplifies the formula F with respect to any unit clauses in F using the BCP sub-routine. For example, if (x) is a unit clause in the formula, then x is set to true and the entire formula is simplified accordingly. After this step, the solver could be in one of three possible states: a satisfying assignment has been found, at which point the solver terminates and returns SAT, or the solver has reached a state where it cannot determine whether the input formula is satisfiable or not, at which point the branching heuristic (the student agent) chooses a variable and adds it to a stack of chosen/decision variables and consequent implications, or a “conflict state” has been reached. A conflict state is defined as when either the formula has been determined to be unsatisfiable, at which point the solver returns UNSAT, or an unsatisfying assignment I has been found contingent on some decision variables. The second of these two scenarios triggers “conflict analysis” (i.e., the teacher/environment is executed to determine a reward), which causes the teacher to analyze the root cause of why the assignment I was unsatisfying, learn a “blocking or learnt clause” C. By blocking we mean that the clause C prevents the solver from ever again exploring this unsatisfying assignment I and potentially exponentially many other assignments that are unsatisfying due to the same root cause. The solver backtracks to undo the decisions that led to the conflict, adds learnt clause C to its database of facts implied by the input formula F, and then continues its search until it converges to the correct SAT/UNSAT answer.

When viewed from an online RL point of view (See Fig. 2), the student agent (i.e., the branching heuristic) performs actions that correspond to choosing decision variables. There are n possible actions that the student can take corresponding to the n variables in the input formula F. The natural question that arises is the following: what is the optimal online policy for the student?. That is, when the heuristic is ready to choose a variable, which one should be chosen. Ideally, the student(s) should take those actions that maximize their reward. In their paper on this RL view of CDCL, Liang et al. [56] modeled branching heuristic or agent as in the stateless Multi-Armed Bandit problem. They then proposed a reward function based on learnt clauses, i.e., those decision variables or actions that were involved in the implications that led to a conflict and the consequent learnt clauses where rewarded by increasing their "activity", while all other variables were not rewarded. The idea is that those decision variables that caused the solver to deduce a valuable learnt clause in the near past would also likely pay off in the near future, during the run of the solver, with more learnt clauses.

The above-mentioned RL-based branching heuristic is by no means the first such method. Perhaps the earliest ML heuristic in the context of the DPLL solver was by Lagoudakis and Littman [58], who used reinforcement learning (RL) as part of a #SAT solver implemented via a DPLL procedure. In brief, their solver works as follows: as mentioned previously, branching heuristics are perhaps one of the most important determinant of a solver's performance. In their solver, Lagoudakis and Littman implemented seven different branching heuristics and the goal of their RL agent is to choose the best among these techniques, immediately prior to a variable being selected to be branched upon. The agent in this case chooses between heuristics (as opposed to variables in the case of LRB, as discussed above) and uses a history-based reward metric.

In their work, Duan et al. [29] used Bayesian Moment Matching technique to learn an initial value to variable assignments that would be more efficient than the default heuristic of setting all variables to the value FALSE, as is often done in most SAT solvers. Flint and Blaschko [34] describe a method where they draw a correspondence between the search performed by CDCL SAT solver and heuristic search, a la the A* search algorithm [45], and use a perceptron to predict which variable, if branched upon, is most likely to lead to a satisfying assignment. As features they use simple variable counts in unary and binary learnt clauses as well as the *activity* of variables.

There are other interesting lines of research at the intersection of the ML and reasoning, such as algorithm selection of solvers [83, 94]. We do not cover these here primarily because these techniques treat solvers as blackboxes, and the ML techniques used are not tied intimately into the inner workings of the solver and hence fall outside the scope of the discussion here. Having said that, these are powerful techniques that have been extensively studied elsewhere [10].

In conclusion, this line of research where ML heuristics are used to optimally select, sequence, and initialize proof rules seems to have had the most impact of all the ML methods that have been researched in the context of ML for solvers. Such ML heuristics are already part of leading sequential and parallel solvers, whether they be SAT, SMT or CP.

3 Inductive learning in synthesis and verification

Inductive machine learning has a close connection with some of the most effective methods for formal verification and synthesis. In this section, we describe how these connections go back a few decades, and outline some directions for the future. A longer exposition of the ideas described here may be found in [80].

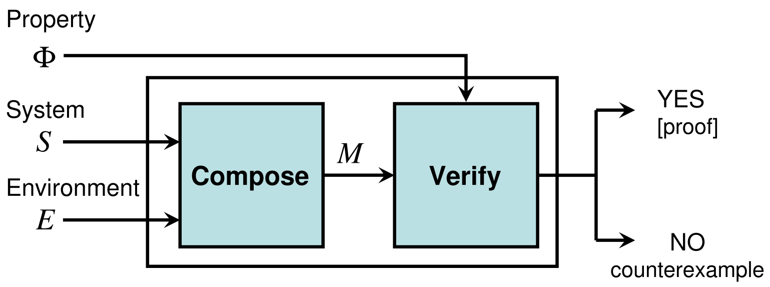

Fig. 3 The traditional view of formal verification

We begin by revisiting the traditional view of formal verification, as a decision problem with three inputs (see Fig. 3): (i) A model of the system to be verified, S; (ii) A model of the environment, E, and (iii) 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 one or more counterexamples which indicate how Φ is violated. For a YES answer, some tools also include a proof or certificate of correctness, which can be checked by an independent procedure.

Similarly, in formal synthesis, one starts with the inputs E and Φ, and seeks to generate a system S such that $S||E \models \Phi$.

As we will see in this section, there is an important and close connection between formal verification and formal synthesis. In fact, the roots of model checking lie in the problem of synthesis: the seminal paper on model checking by Clarke and Emerson [19] 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."

Moreover, we will see how the bridge from formal verification to synthesis is facilitated through the use of machine learning.

3.1 Verification by reduction to synthesis

A key connection between formal verification and formal synthesis is that many approaches to verification operate by solving one or more synthesis tasks. In other words, verification is performed 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. First, we introduce suitable notation. Additional background material may be found in a book chapter on formal modeling for verification [88].

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.

3.1.1 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:

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

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

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 a system satisfies an invariant property *reduces* to the problem of synthesizing an auxiliary invariant ψ such that the following two formulas are valid:

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

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

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.

3.1.2 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.

3.1.3 Other examples

In the original papers outlining verification by reduction to synthesis, and accompanying presentations, Seshia [79, 80] listed several formal artifacts generated in verification that could benefit from the application of synthesis, including not just inductive invariants and

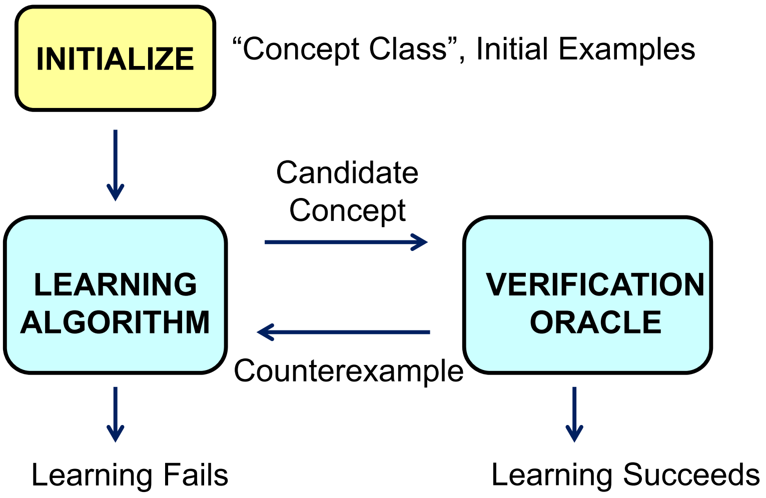

Fig. 4 Counterexample-guided inductive synthesis (CEGIS)

abstractions, but also auxiliary invariants, environment assumptions and models, interface specifications, pre- and post-conditions for procedures, ranking functions for proving termination, interpolants, etc. Moreover, Seshia also mentioned how the formal artifacts produced inside solvers, such as theory lemmas learned within SMT solvers and instantiations of quantifiers in SMT solvers, can also benefit from synthesis. In the decade since, we have seen several papers demonstrate this use of synthesis for verification, particularly applications of the paradigm of *syntax-guided synthesis* (SyGuS) [1].

3.2 SID and counterexample-guided inductive synthesis

If verification can be effectively solved by reduction to synthesis, how can one effectively solve synthesis problems?

There are a variety of techniques for solving formal synthesis, ranging from the classic deductive approaches such as the methods of Manna and Waldinger [64] to ML-based “inductive synthesis”. However, the dramatic advances in program synthesis achieved over the last two decades have come about as a result of a combination of inductive and deductive synthesis. This trend was formalized in [79, 80] as the Structure-Induction-Deduction (SID) methodology, an integration of inductive inference and deductive reasoning using structure hypotheses — hypotheses about the structural form of the artifacts being synthesized. Common approaches such as sketch-based synthesis [82], template-based synthesis [49] and component-based synthesis [48] are instances of the SID approach. For further details, we refer the reader to [80].

The most popular instantiation of SID is the technique known as *counterexample-guided inductive synthesis* (CEGIS) [82]. Figure 4 gives a high-level view of the CEGIS approach to synthesis.

The defining aspect of CEGIS is its learning strategy: *learning from counterexamples provided by a verification oracle and positive examples provided by an input–output oracle*. The learning algorithm, which is initialized with a particular choice of concept class L and possibly with an initial set of (positive) examples, proceeds by searching the space

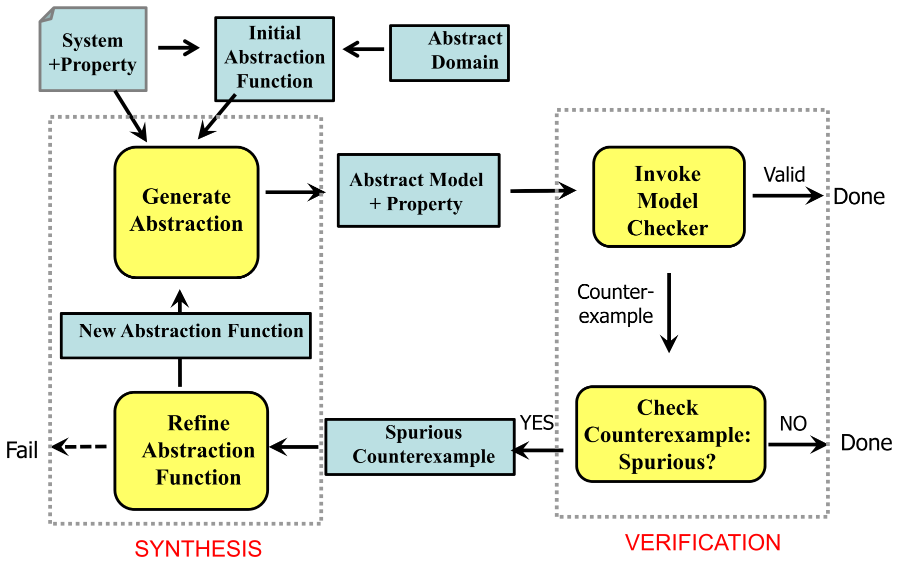

Fig. 5 Counterexample-guided abstraction refinement (CEGAR) as inductive synthesis

of candidate concepts for one that is consistent with the examples seen so far. There may be several such consistent concepts, and the search strategy determines the chosen candidate, an expression e. The concept e is then presented to the verification oracle, which checks the candidate against the correctness specification Φ. This verification oracle can be implemented as an SMT solver that checks whether Φ is satisfied by the candidate. If yes, the synthesizer terminates and outputs this candidate. Otherwise, the verification oracle generates a counterexample, which is returned to the learner. The learning algorithm adds the counterexample to its set of examples and repeats its search. Note that the precise encoding of a counterexample and its use can vary depending on the details of the learning algorithm employed. It is possible that, after some number of iterations of this loop, the learning algorithm may be unable to find a candidate concept consistent with its current set of (positive/negative) examples, in which case the learning step, and hence the overall CEGIS procedure, fails.

Several search strategies have been developed over the years for learning a candidate expression (program) in L, each with its pros and cons. See [1, 6] for an overview of the basic approaches for CEGIS.

3.3 Counterexample-guided abstraction refinement

Counterexample-guided abstraction refinement (CEGAR) [21] is an algorithmic approach to perform abstraction-based model checking. CEGAR has been successfully applied to hardware [21], software [14], and hybrid systems [20]. In the context of this section, CEGAR can be seen as a learning-based approach to synthesizing abstractions for formal verification.

Figure 5 gives an overview of the CEGAR approach. CEGAR solves the synthesis task described in Sec. 3.1.2 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 the SID approach 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 [54], in which abstract states are cubes over the state variables.
- The inductive learner 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 [62], although the traditional CEGAR refinement algorithms are somewhat different from the learning algorithms proposed in the version spaces framework. Gupta, Clarke, et al. [23, 43] have previously observed the link to inductive learning and have proposed versions of CEGAR based on alternative machine learning algorithms (such as induction on decision trees).
- The deductive engine 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 fact, CEGAR can be seen as an instance of CEGIS, where the deductive engine plays the role of the verification engine, the inductive learner plays the role of the inductive synthesizer, and the structure hypothesis defines the space of artifacts (abstractions) to be synthesized. We emphasize that the idea of CEGAR precedes that of CEGIS and the SID methodology, and CEGAR was undoubtedly influential in the formulation of CEGIS. Our aim, in this section, is to simply point out how the ideas generalize and connect to the effective use of machine learning for formal verification and synthesis.

3.4 Formal inductive synthesis and oracle-guided inductive synthesis

While CEGIS has been, by far, the leading algorithmic method for program synthesis over the last fifteen years, there is also a growing realization of its limitations. Foremost amongst these is a heavy reliance on having a verification oracle with two crucial properties: (i) it must be efficient and (ii) it must provide informative counterexamples. However, when verification itself is computationally expensive, high-quality formal specifications are unavailable, or when the verifier does not by default provide counterexamples that help the learner converge quickly to a correct program, CEGIS falls short. For this reason, researchers have developed alternatives to the basic CEGIS approach such as learning from distinguishing inputs [48], which uses an oracle that can provide inputs that distinguish seemingly-correct candidate programs, and CEGIS(T), which generalizes from concrete counterexamples to produce constraints that better direct the search for candidate programs [2].

Even with these alternatives to CEGIS, approaches to inductive learning-based synthesis share some common characteristics that have been captured in the SID methodology [80] and further formalized by Jha and Seshia [51] as *formal inductive synthesis* (FIS), a family of synthesis problems, and *oracle-guided inductive synthesis* (OGIS), a family of solution techniques for solving FIS. Central to FIS and OGIS is the concept of an *oracle interface*, which is a set of query-response pairs that define the interface between the learner and the oracle(s) that guide the learner in its search for a program.

An FIS problem has four inputs: (i) a space of candidate programs or formal artifacts, (ii) a domain of example behaviors or traces, (iii) a formal specification, and (iv) an oracle interface that defines the types of oracles available to the learner and the interface to them. Thus, in contrast to pure learning-based methods, it requires the presence of a formal specification defining correctness and also allows for the use of richer oracle interactions than possible in standard ML paradigms such as supervised, unsupervised, and active learning. The only exception is the area of query-based learning (e.g. [4]), but even here there is an important distinction: in query-based learning, the oracles are part of the problem description and the learner must work with *any* oracle that satisfies the oracle interface, whereas in FIS, the oracles are part of the solution and *can be co-designed with the learner* so long as they adhere to the oracle interface. See [51] for more details and theoretical results about FIS and OGIS.

Oracle-guided inductive synthesis (OGIS) generalizes CEGIS and other variants of inductive synthesis by developing a general theory of interaction between learners and oracles. In OGIS, the process of synthesis is coordinated by a dialog between a learner and an oracle that adheres to a given oracle interface. A OGIS procedure comprises a pair of learner and oracle and solves a given FIS problem if there exists a dialog that either converges in the learner providing a candidate program that satisfies the formal specification or the learner concluding that no such program exists for the given FIS problem. Jha and Seshia [51] provide several instantiations of OGIS, including CEGIS and Angluin's L^* query-based learning algorithm for deterministic finite automata. For example, CEGIS is a special case of OGIS, where the oracle interface comprises two query types: a *positive witness query* where an oracle must provide a positive example (input–output example or trace) and a *counterexample query* where a verification oracle provides a counterexample showing how the candidate program violates the formal specification. Jha and Seshia [51] provide a theoretical analysis of CEGIS, showing how different choices of counterexample can influence whether CEGIS converges to a correct program or not, for infinite-sized program spaces where termination is not guaranteed. They also show how, for finite program spaces, the concept of *teaching dimension* [42] is useful to bound the query complexity of OGIS. More recently, other researchers have introduced newer query types such as the *Hoare query* introduced by Feldman et al. [36], where they show, for invariant synthesis, a class of finite-state transition systems and invariants such that every learner, even computationally-unbounded ones, will require $2^{\Omega(n)}$ Hoare queries, where n is the number of Boolean state variables.

While the afore-mentioned papers set out the initial theory of FIS and OGIS, much more remains to be done. One important goal is to develop efficient, general-purpose solvers to solve OGIS problems. A step towards this goal has been recently taken by Polgreen et al. [73] who introduce the frameworks of *satisfiability modulo theories and oracles* (SMTO) and *synthesis modulo oracles* (SyMO). SMTO extends SMT solving with reasoning about oracle functions. SyMO extends synthesis solving, particularly SyGuS solving, to admit a more general class of oracle interfaces, and provides a generic approach to solving OGIS problems via SyGuS by translating oracle responses back into logical constraints. SMTO and SyMO is now being integrated into verification tools such as the UCLID5 system [68, 86].

In summary, formal inductive synthesis and oracle-guided inductive synthesis provide a firm foundation for learning-driven synthesis and its integration with formal verification, building on the pioneering contributions of CEGAR and CEGIS. We believe the development of the theory, algorithms, and applications of FIS and OGIS will be a productive topic of research for the coming decade.

4 Deep neural networks with feedback from solvers

In recent years, another very interesting trend of incorporating solvers in DNNs is gaining momentum that promises to change AI in a fundamental way. The solvers in these DNN-solver NeuroSymbolic architectures are used during training as part of the back-propagation step [92] and/or during inference, or as an active learning corrective feedback mechanism from solver to the DNN post-inference [92]. We cover three prominent lines of research in this context, namely, SATnet [92], integration of blackbox solvers in DNNs [72], and provide a brief overview of additional attempts at integrating DNNs and solvers.

In their paper on the SATNet tool, Wang et al. present a DNN architecture with a differentiable approximate MAXSAT solver layer. Their differentiable approximate solver is based on a coordinate descent approach to solving the semidefinite program (SDP) relaxation of the MAXSAT problem. During the backward pass of its training, the SATNet tool leverages the fact that the solver is differentiable, in order to differentiate through it and appropriately update the weights of DNN. Unlike many other attempts at integrating solvers into DNNs, SATNet does not assume that the logical structure of the problem is given, but rather is learnt. Prominently, the authors use SATNet to learn how to play “Visual Sudoku" solely from examples. The Visual Sudoku problem asks for a system to map images of incomplete Sudoku puzzles to completed puzzles. SATNet uses its differentiable solver to learn the logical structure between the variables representing a Sudoku puzzle, while at the same time attempts to solve the given incomplete input Sudoku puzzle.

By contrast, Vlastelica et al. [72] use a combinatorial blackbox solver as a layer in a DNN with the goal of ensuring global consistency in a multi-object tracking or route planning on map images. Consider the problem of a robot trying to find the shortest path given a picture of a map. This problem requires the robot to solve two sub-problems, first image and object recognition that recognizes a map and paths in it, and a second sub-problem of solving for the shortest path through the graph corresponding to the input map. Further, the authors assume that the only labels available are the outputs of solvers, and hence their tool has to discover the label for the output of the DNN itself. Finally, the authors insist on using a combinatorial solver as a blackbox and not require it to be differentiable. The requirement that solvers be differentiable is a big weakness of methods like SATNet because such solvers are sub-optimal in terms of runtime, performance and optimality guarantees. Further, this requirement limits the choices one has for the solver layer, limiting ones ability to leverage the transformational progress made in combinatorial solver algorithms.

The primary technical challenge for such a design boils down to providing meaningful gradient information to the DNN from the solver during the backward pass. To accomplish, the authors are able to leverage the minimization structure of the underlying combinatorial problem and efficiently compute a gradient of a continuous interpolation of the linearization of the loss computed over output of the solver and the actual label. They successfully apply their method to a variety of problems including the problem of route planning over map images in the context of robotics. We find this approach to be very versatile and can be applied in a variety of contexts where one needs to bring learning and reasoning together in corrective feedback loop.

Another approach that combines ML with solvers is Logic Guided Machine Learning (LGML) [84] and its cousin Logic Guided Genetic Algorithms (LGGA) [7]. The context of these approaches is symbolic regression (SR), where the goal is to convert data into symbolic formulas via ML [67]. Symbolic regression (SR) is a broad sub-field of ML where the goal is to learn a model that takes as input data and a mathematical language L, and outputs an L-formula that fits the data [52, 67].

The primary application of SR is in automated knowledge discovery from data, and it can be helpful in interpreting models [8]. A problem common to many SR techniques is that the equations produced by these systems often overfit their input data, i.e., relying on input data alone is sometimes a poor guide to discovering new knowledge. A more effective approach is to leverage known facts or domain knowledge to debug equations learnt by SR systems.

In their paper, Scott et al. [84] propose a new class of methods that combines SR systems with SMT solvers. The SMT solvers are used to cross-verify whether the equations produced by the SR are consistent with domain knowledge, which are expressed in the language of the SMT solver. If not, a counterexample is generated which is then fed back to the SR model, in a manner similar to Counter-Example Guided Abstraction Refinement. Kim et al. [52] recently demonstrated a DNN-based symbolic regression technique, wherein equation learning is performed via gradient-based optimization, and can be combined with other DNN layers.

One of the most important recent developments in ML research is the rise of large language models (LLMs) such as OpenAI's GPT4 [66]. These models seem to perform spectacular feats of generating text and other media that match, and often exceed, expert human-level capabilities. While such models are known to perform logical and mathematical reasoning (e.g., Minerva [55] from Google Research), they also fail spectacularly on many instances of such tasks. It is but natural to ask whether combining LLMs with solvers and provers can improve their performance at reasoning tasks. One direction of research in this context that we find noteworthy is the tight integration of LLMs with proof assistants such as HOL Lite and Coq, often referred to as Neural Theorem Proving (NTP). Due to space limitations, we don't expand on this topic and refer the reader to recent work on NTPs [37].

In most of these above-mentioned approaches, logical reasoning tools such as solvers/provers are used to provide corrective feedback to a DNN or an ML model. In some of these techniques, such as SATnet, solvers are used both during training as well as inference.

5 Conclusion

In this perspective paper, we discuss topics at the intersection of ML and logical reasoning, which historically have been the two key pillars of AI. While the advances in these two sub-fields have been impressive, they have largely been pursued by different communities. In recent years, researchers have identified the need to bring these communities together, and some work has started at the intersection of these two sub-areas. This paper highlights the need for tighter integration between ML and logical reasoning. Within this context, we cover three topics highlighting the synergy between ML and logical reasoning: (1) use of ML in solvers, (2) use of ML in synthesis and verification of programs and proofs, and (3) combinations of ML and symbolic solvers with the goal of enabling logical reasoning in ML. Rather than give an exhaustive survey of these topics, we have tried to provide enough background to highlight interesting directions for future research. One direction that is particularly noteworthy is the use of corrective feedback loops between an ML models and reasoning engines. We are inspired by our belief that research at the intersection of learning and reasoning can be deeply and fundamentally transformative for both the field of AI and society more broadly. We very much hope that the directions and problems that we have identified will stimulate further research in this new and emerging field of AI.

References

1. Alur R, Bodik R, Juniwal G, Martin Milo MK, Raghothaman M, Seshia SA, Singh R, Solar-Lezama A, Torlak E, Udupa A (2013) Syntax-guided synthesis. In: Proceedings of the IEEE international conference on formal methods in computer-aided design (FMCAD), pp 1–17
2. Abate A, David C, Kesseli P, Kroening D, Polgreen E (2018) Counterexample guided inductive synthesis modulo theories. In: Computer aided verification-30th international conference (CAV), volume 10981 of Lecture Notes in Computer Science, pp 270–288. Springer
3. Atserias A, Fichte JK, Thurley M (2009) Clause-learning algorithms with many restarts and bounded-width resolution. In: Kullmann O (ed) Theory and Applications of Satisfiability Testing - SAT. Springer, Berlin Heidelberg, Berlin, Heidelberg, pp 114–127
4. Angluin D (1988) Queries and concept learning. Mach Learn 2(4):319–342
5. Audemard Gilles, Simon Laurent (2013) Glucose 2.3 in the SAT 2013 Competition. In: Proceedings of SAT competition 2013, pp 42–43
6. Alur R, Singh R, Fisman D, Solar-Lezama A (2018) Search-based program synthesis. Commun ACM 61(12):84–93
7. Ashok D, Scott J, Wetzel SJ, Panju M, Ganesh V (2021) Logic guided genetic algorithms. In: 35th AAAI conference on artificial intelligence, AAAI 2021, 33rd Conference on innovative applications of artificial intelligence, IAAI 2021, The 11th symposium on educational advances in artificial intelligence, EAAI 2021, virtual event, Feb 2–9, 2021, pp 15753–15754. AAAI Press
8. Alaa AM, Schaar M van der (2019) Demystifying black-box models with symbolic metamodels. Adv Neural Inf Process Syst 32
9. Bouraoui Z, Cornuéjols A, Denoeux T, Destercke S, Dubois D, Guillaume R, Marques-Silva J, Mengin J, Prade H, Schockaert S, Serrurier M, Vrain C (2019) From shallow to deep interactions between knowledge representation, reasoning and machine learning (kay r. amel group). CoRR, abs/1912.06612
10. Biere A, Heule M, van Maaren H, Walsh T (2009) (eds) Handbook of Satisfiability, vol. 185 of Frontiers in Artificial Intelligence and Applications. IOS Press
11. Bünz B, Lamm M (2017) Graph neural networks and boolean satisfiability. CoRR, abs/1702.03592
12. Bengio Y, LeCun Y, Hinton GE (2021) Deep learning for AI. Commun ACM 64(7):58–65
13. Bak S, Liu C, Johnson TT (2021) The second international verification of neural networks competition (VNN-COMP 2021): Summary and results. CoRR, abs/2109.00498
14. Ball T, Levin V, Rajamani SK (2011) A decade of software model checking with SLAM. Commun ACM 54(7):68–76
15. Bansal K, Loos SM, Rabe MN, Szegedy C, Wilcox S (2019) Holist: an environment for machine learning of higher order logic theorem proving. In: Kamalika C and Ruslan S, (eds.) Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA, volume 97 of Proceedings of Machine Learning Research, pp 454–463. PMLR
16. Bader-El-Den MB, Poli R (2007) Generating SAT local-search heuristics using a GP hyper-heuristic framework. In: Nicolas M, El-Ghazali T, Pierre C, Marc S, Evelyne L, (eds), Artificial Evolution, 8th International Conference, Evolution Artificielle, EA 2007, Tours, France, October 29-31, 2007, revised selected papers, volume 4926 of Lecture Notes in Computer Science, Springer, pp 37–49
17. Clarke E, Biere A, Raimi R, Zhu Y (2001) Bounded model checking using satisfiability solving. Formal Methods Syst Des 19(1):7–34
18. Cropper A, Dumancic S, Evans R, Muggleton SH (2022) Inductive logic programming at 30. Mach Learn 111(1):147–172
19. Clarke EM, Emerson EA (1981) Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of programs, pp 52–71
20. Clarke EM, Fehnker A, Han Z, Krogh BH, Stursberg O, Theobald M (2003) Verification of hybrid systems based on counterexample-guided abstraction refinement. In: TACAS, pp 192–207
21. Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: 12th international conference on computer aided verification (CAV), vol. 1855 of Lecture Notes in Computer Science, Springer, pp 154–169

22. Clarke EM, Grumberg O, Jha S, Yuan L, Veith H (2003) Counterexample-guided abstraction refinement for symbolic model checking. J ACM 50(5):752–794
23. Clarke EM, Gupta A, Kukula JH, Strichman O (2002) SAT based abstraction-refinement using ILP and machine learning techniques. In: computer aided verification, 14th international conference (CAV), vol. 2404 of Lecture Notes in Computer Science, Springer, pp 265–279
24. Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR (2006) EXE: automatically generating inputs of death. In: Proceedings of the 13th ACM conference on computer and communications security, CCS '06, New York, NY, USA. ACM, pp 322–335
25. Cook S (1971) The complexity of theorem-proving procedures. In: proceedings of the third annual ACM symposium on theory of computing (STOC), ACM, pp 151–158
26. Dash T, Chitlangia S, Ahuja A, Srinivasan A (2021) How to tell deep neural networks what we know. CoRR, abs/2107.10295
27. Davis M, Logemann G, Loveland D (1962) A machine program for theorem-proving. Commun ACM 5(7):394–397
28. De Moura L, Bjørner N (2008) Z3: An efficient smt solver. In: international conference on tools and algorithms for the construction and analysis of systems, Springer, pp 337–340
29. Duan H, Nejati S, Trimponias G, Poupart P, Ganesh V (2020) Online bayesian moment matching based SAT solver heuristics. In: Proceedings of the 37th international conference on machine learning, ICML 2020, 13–18 July 2020, Virtual Event, vol. 119 of proceedings of machine learning research, PMLR, pp 2710–2719
30. Devlin D, O'Sullivan B (2008) Satisfiability as a classification problem. In: Proceedings of the 19th Irish Conference on artificial intelligence and cognitive science
31. Dubois D, Prade H (2019) Towards a reconciliation between reasoning and learning-a position paper. In: Nahla Ben A, Benjamin Q, Martin T, (eds) scalable uncertainty management-13th international conference, SUM 2019, Compiègne, France, Dec 16–18, 2019, Proceedings, vol. 11940 of lecture notes in computer science, Springer, pp 153–168
32. Eén N, Sörensson N (2004) Theory and applications of satisfiability testing: 6th international conference, SAT 2003, santa margherita ligure, Italy, May 5–8, 2003, selected revised papers, chapter an extensible SAT-solver, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 502–518
33. Ertel W, Schumann J, Suttner CB (1989) Learning heuristics for a theorem prover using back propagation. In: Johannes R, Karl L, (eds) 5. Österreichische Artificial Intelligence-Tagung, Igls, Tirol, 28. bis 30. Sept 1989, Proceedings, vol. 208 of Informatik-Fachberichte, Springer, pp 87–95
34. Flint A, Blaschko MB (2012) Perceptron learning of SAT. In: Bartlett PL, Pereira FNC, Burges CJC, Léon B, Weinberger KQ, (eds) Advances in neural information processing systems 25: 26th annual conference on neural information processing systems 2012. Proceedings of a meeting held Dec 3–6, 2012, Lake Tahoe, Nevada, United States, pp 2780–2788
35. Froleyks N, Heule M, Iser M, Järvisalo M, Suda M (2021) SAT competition 2020. Artif Intell 301:103572
36. Feldman YMY, Immerman N, Sagiv M, Shoham S (2020) Complexity and information in invariant inference. Proc ACM Program Lang 4(POPL):5:1-5:29
37. First E, Rabe MN, Ringer T, Brun Y (2023) Baldur: whole-proof generation and repair with large language models. CoRR, abs/2303.04910
38. Fukunaga AS (2002) Automated discovery of composite SAT variable-selection heuristics. In: Dechter R, Kearns MJ, Sutton RS, (eds) proceedings of the eighteenth national conference on artificial intelligence and fourteenth conference on innovative applications of artificial intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada, AAAI Press / The MIT Press, pp 641–648
39. Fukunaga AS (2004) Evolving local search heuristics for SAT using genetic programming. In: Kalyanmoy D, Riccardo P, Wolfgang B, Hans-Georg B, Burke EK, Darwen PJ, Dasgupta D, Floreano D, Foster JA, Mark H, Owen H, Pier Luca L, Lee S, Andrea T, Dirk T, Tyrrell AM, (eds) Genetic and evolutionary computation-GECCO 2004, genetic and evolutionary computation conference, Seattle, WA, USA, June 26–30, 2004, Proceedings, Part II, volume 3103 of Lecture Notes in Computer Science, Springer, pp 483–494
40. Fukunaga AS (2008) Automated discovery of local search heuristics for satisfiability testing. Evol Comput 16(1):31–61
41. Ian JG, Yoshua B, Courville AC (2016) Deep Learning. Adaptive computation and machine learning. MIT Press, Cambridge
42. Goldman SA, Kearns MJ (1992) On the complexity of teaching. J Comput Syst Sci 50:303–314
43. Gupta A (2006) Learning Abstractions for Model Checking. PhD thesis, Carnegie Mellon University, June

44. Hutter F, Babic D, Hoos HH, Hu AJ (2007) Boosting verification by automatic tuning of decision procedures. In: 7th international conference on formal methods in computer-aided design (FMCAD), IEEE Computer Society, pp 27–34
45. Hart PE, Nilsson NJ, Raphael B (1968) A formal basis for the heuristic determination of minimum cost paths. IEEE Trans Syst Sci Cybern 4(2):100–107
46. Holden SB (2021) Machine learning for automated theorem proving: learning to solve SAT and QSAT. Found Trends Mach Learn 14(6):807–989
47. Hopfield JJ, Tank DW (1985) Neural computation of decisions in optimization problems. Biol Cybern 52:141–152
48. Jha S, Gulwani S, Seshia SA, Tiwari A (2010) Oracle-guided component-based program synthesis. In: Proceedings of the 32nd international conference on software engineering (ICSE), pp 215–224
49. Jha S, Gulwani S, Seshia SA, Tiwari A (2010) Synthesizing switching logic for safety and dwell-time requirements. In: proceedings of the international conference on cyber-physical systems (ICCPS), pp 22–31
50. Johnson JL (1989) A neural network approach to the 3-satisfiability problem. J Parallel Distributed Comput 6:435–449
51. Jha S, Seshia SA (2017) A theory of formal synthesis via inductive learning. Acta Inform 54(7):693–726
52. Kim S, Lu PY, Mukherjee S, Gilbert M, Jing L, Čeperić V, Soljačić M (2021) Integration of neural network-based symbolic regression in deep learning for scientific discovery. IEEE Trans Neural Netw Learn Syst 32(9):4166–4177
53. Kautz HA, Selman B (1992) Planning as satisfiability. In: Bernd N, (ed) 10th European conference on artificial intelligence, ECAI 92, Vienna, Austria, August 3–7, 1992. Proceedings, John Wiley and Sons, pp 359–363
54. Kurshan R (1994) Automata-theoretic verification of coordinating processes. In: 11th international conference on analysis and optimization of systems–discrete event systems, vol. 199 of LNCS, Springer, pp 16–28
55. Lewkowycz A, Andreassen A, Dohan D, Dyer E, Michalewski H, Ramasesh VV, Slone A, Anil C, Schlag I, Gutman-Solo T, Wu Y, Neyshabur B, Gur-Ari G, Misra V (2022) Solving quantitative reasoning problems with language models. CoRR, abs/2206.14858
56. Liang JH, Ganesh V, Poupart P, Czarnecki K (2016) Learning rate based branching heuristic for SAT solvers. In: Nadia C, Daniel LB, (eds) Theory and applications of satisfiability testing–SAT 2016, Cham. Springer International Publishing, pp 123–140
57. Liang JH (2018) Machine learning for SAT solvers. PhD thesis, University of Waterloo, Canada
58. Lagoudakis Michail G, Littman Michael L (2001) Learning to select branching rules in the DPLL procedure for satisfiability. Electron Notes Discrete Math 9:344–359
59. Liang JH, Oh C, Mathew M, Thomas C, Li C, Ganesh V (2018) Machine learning-based restart policy for CDCL SAT solvers. In: theory and applications of satisfiability testing-SAT 2018 - 21st international conference, SAT 2018, held as part of the federated logic conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings, pp 94–110
60. Lederman G, Rabe MN, Seshia S, Lee EA (2020) Learning heuristics for quantified boolean formulas through reinforcement learning. In: 8th international conference on learning representations (ICLR), April
61. Lorenz JH, Wörz F (2020) On the effect of learned clauses on stochastic local search. In: Luca P, Martina S, (eds) Theory and applications of satisfiability testing-SAT 2020–23rd international conference, Alghero, Italy, July 3–10, 2020, Proceedings, vol. 12178 of lecture notes in computer science, Springer, pp 89–106
62. Tom M (1997) Mitchell. McGraw-Hill, Machine Learning
63. Marques-Silva JP, Sakallah KA (1996) GRASP-A new search algorithm for satisfiability. In: proceedings of the 1996 IEEE/ACM international conference on computer-aided design, ICCAD '96, Washington, DC, USA. IEEE Computer Society, pp 220–227
64. Manna Z, Waldinger R (1980) A deductive approach to program synthesis. ACM Trans Program Lang Syst (TOPLAS) 2(1):90–121
65. Nejati S, Frioux LL, Ganesh V (2020) A machine learning based splitting heuristic for divide-and-conquer solvers. In: Helmut S, (ed) Principles and practice of constraint programming-26th international conference, CP 2020, Louvain-la-Neuve, Belgium, Sept 7–11, 2020, Proceedings, vol. 12333 of lecture notes in computer science, Springer, pp 899–916
66. OpenAI. GPT-4 technical report. CoRR, abs/2303.08774, 2023
67. Panju M (2021) Automated Knowledge Discovery using Neural Networks. PhD thesis, University of Waterloo, Ontario, Canada

68. Polgreen E, Cheang K, Gaddamadugu P, Godbole A, Laeufer K, Lin S, Manerkar YA, Mora F, Seshia SA (2022) UCLID5: multi-modal formal modeling, verification, and synthesis. In: computer aided verification-34th international conference (CAV), vol. 13371 of lecture notes in computer science, Springer, pp 538–551
69. Pipatsrisawat K, Darwiche A (2011) On the power of clause-learning SAT solvers as resolution engines. Artif Intell 175(2):512–525
70. Andrei P, Polat ES, Alexander F, Mathias U, Müslüm A, Viet-Man L, Klaus P, Martin E, Trang TTN (2022) An overview of machine learning techniques in constraint solving. J Intell Inf Syst 58(1):91–118
71. Pimpalkhare N, Mora F, Polgreen E, Seshia SA (2021) MedleySolver: online SMT algorithm selection. In: 24th international conference on theory and applications of satisfiability testing (SAT), vol. 12831 of lecture notes in computer science, Springer, pp 453–470
72. Pogancic MV, Paulus A, Musil V, Martius G, Rolínek M (2020) Differentiation of blackbox combinatorial solvers. In: 8th international conference on learning representations, ICLR 2020, Addis Ababa, Ethiopia, April 26–30, 2020. OpenReview.net
73. Polgreen E, Reynolds A, Seshia SA (2022) Satisfiability and synthesis modulo oracles. In: Proceedings of the 23rd international conference on verification, model checking, and abstract interpretation (VMCAI)
74. De Raedt L (2008) Logical and relational learning. Cognitive technologies. Springer, Berlin
75. Russell S, Norvig P (2020) Artificial intelligence: a modern approach, 4th edn. Pearson, London
76. Rossi F, van Beek P, Walsh T (2006) editors. Handbook of Constraint Programming, vol. 2 of Foundations of Artificial Intelligence. Elsevier
77. Richard SS, Andrew GB (1998) Reinforcement learning-an introduction. Adaptive computation and machine learning. MIT Press, Cambridge
78. Seshia SA (2005) Adaptive eager boolean encoding for arithmetic reasoning in verification. PhD thesis, Carnegie Mellon University
79. Seshia SA (2012) Sciduction: combining induction, deduction, and structure for verification and synthesis. In: proceedings of the design automation conference (DAC), pp 356–365
80. Seshia SA (2015) Combining induction, deduction, and structure for verification and synthesis. Proc IEEE 103(11):2036–2051
81. Selsam D, Lamm M, Bünz B, Liang P, de Moura L, Dill DL (2019) Learning a SAT solver from single-bit supervision. In: 7th international conference on learning representations, ICLR 2019, New Orleans, LA, USA, May 6–9, 2019. OpenReview.net
82. Solar-Lezama A, Tancau L, Bodík R, Seshia SA, Saraswat VA (2006) Combinatorial sketching for finite programs. In: proceedings of the 12th international conference on architectural support for programming languages and operating systems (ASPLOS), ACM Press, pp 404–415
83. Scott J, Niemetz A, Preiner M, Nejati S, Ganesh V (2021) Machsmt: a machine learning-based algorithm selector for SMT solvers. In: Jan Friso G, Kim Guldstrand L, (eds) Tools and algorithms for the construction and analysis of systems-27th international conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part II, vol. 12652 of Lecture Notes in Computer Science, Springer, pp 303–325
84. Scott J, Panju M, Ganesh V (2020) LGML: logic guided machine learning. In: the thirty-fourth AAAI conference on artificial intelligence, AAAI 2020, the thirty-second innovative applications of artificial intelligence conference, IAAI 2020, the tenth AAAI symposium on educational advances in artificial intelligence, EAAI 2020, New York, NY, USA, Feb 7–12, 2020, AAAI Press, pp 13909–13910
85. Marques SJP, Sakallah KA (1996) GRASP-a new search algorithm for satisfiability. In: Rutenbar RA, Otten RHJM, (eds) proceedings of the 1996 IEEE/ACM international conference on computer-aided design, ICCAD 1996, San Jose, CA, USA, Nov 10–14, 1996, IEEE Computer Society / ACM, pp 220–227
86. Seshia SA, Subramanyan P (2018) UCLID5: integrating modeling, verification, synthesis, and learning. In: proceedings of the 15th ACM/IEEE international conference on formal methods and models for codesign (MEMOCODE)
87. Seshia SA, Sadigh D, Sastry SS (2022) Toward verified artificial intelligence. Commun ACM 65(7):46–55
88. Seshia SA, Sharygina N, Tripakis S (2018) Modeling for verification. In: Clarke EM, Thomas H, Helmut V, (eds) Handbook of Model Checking, chapter 3. Springer
89. Sarker Md, Kamruzzaman ZL, Aaron E, Pascal H (2021) Neuro-symbolic artificial intelligence. AI Commun 34(3):197–209

90. van Harmelen F, Lifschitz V, Porter BW (2008) (eds) Handbook of Knowledge Representation, vol. 3 of Foundations of Artificial Intelligence. Elsevier
91. The Verification of Neural Networks Library (VNN-LIB). www.vnnlib.org, 2019
92. Wang PW, Donti PL, Wilder B, Zico KJ (2019) Satnet: bridging deep learning and logical reasoning using a differentiable satisfiability solver. In: Kamalika C, Ruslan S, (eds) proceedings of the 36th international conference on machine learning, ICML 2019, 9–15 June 2019, Long Beach, California, USA, vol. 97 of Proceedings of Machine Learning Research, PMLR, pp 6545–6554
93. Jeannette MW (2021) Trustworthy AI. Commun ACM 64(10):64–71
94. Lin X, Hutter F, Hoos HH, Leyton-Brown K (2008) SATzilla: portfolio-based algorithm selection for SAT. J Artif Intell Res 32(1):565–606

Authors and Affiliations

Vijay Ganesh[1] · **Sanjit A. Seshia**[2] · **Somesh Jha**[3]

✉ Vijay Ganesh
vganesh@uwaterloo.ca

Sanjit A. Seshia
sseshia@eecs.berkeley.edu

Somesh Jha
jha@cs.wisc.edu

1 University of Waterloo, Ontario, Canada

2 University of California, Berkeley, USA

3 University of Wisconsin, Madison, USA