Journal of Automated Reasoning (2020) 64:1523–1552
https://doi.org/10.1007/s10817-020-09570-z

A Learning-Based Approach to Synthesizing Invariants for Incomplete Verification Engines

Daniel Neider[1] · P. Madhusudan[2] · Shambwaditya Saha[2] · Pranav Garg[3] · Daejun Park[2]

Received: 22 June 2020 / Accepted: 26 June 2020 / Published online: 13 July 2020

Abstract
We propose a framework for synthesizing inductive invariants for incomplete verification engines, which soundly reduce logical problems in undecidable theories to decidable theories. Our framework is based on the counterexample guided inductive synthesis principle and allows verification engines to communicate non-provability information to guide invariant synthesis. We show precisely how the verification engine can compute such non-provability information and how to build effective learning algorithms when invariants are expressed as Boolean combinations of a fixed set of predicates. Moreover, we evaluate our framework in two verification settings, one in which verification engines need to handle quantified formulas and one in which verification engines have to reason about heap properties expressed in an expressive but undecidable separation logic. Our experiments show that our invariant synthesis framework based on non-provability information can both effectively synthesize inductive invariants and adequately strengthen contracts across a large suite of programs. This work is an extended version of a conference paper titled "Invariant Synthesis for Incomplete Verification Engines".

Keywords Software verification · Invariant synthesis · Undecidable theories · Incomplete decision procedures · Machine learning

✉ Daniel Neider
neider@mpi-sws.org

P. Madhusudan
madhu@illinois.edu

Shambwaditya Saha
ssaha6@illinois.edu

Pranav Garg
prangarg@amazon.com

Daejun Park
dpark69@illinois.edu

[1] Max Planck Institute for Software Systems, Kaiserslautern, Germany

[2] University of Illinois at Urbana-Champaign, Champaign, IL, USA

[3] Amazon Research, New York City, NY, USA

1 Introduction

The paradigm of *deductive verification* [24,32] combines manual annotations and semi-automated theorem proving to prove programs correct. Programmers annotate code they develop with contracts and inductive invariants, and use high-level directives to an underlying, mostly-automated logic engine to verify their programs correct. Several mature tools have emerged that support such verification, in particular tools based on the intermediate verification language BOOGIE [3] and the SMT solver Z3 [18] (e.g., VCC [13] and DAFNY [42]).

Viewed through the lens of deductive verification, the primary challenges in automating verification are two-fold. First, even when strong annotations in terms of contracts and inductive invariants are given, the validity problem for the resulting verification conditions is often undecidable (e.g., in reasoning about the heap, reasoning with quantified logics, and reasoning with non-linear arithmetic). Second, the synthesis of loop invariants and strengthenings of contracts that prove a program correct needs to be automated so as to lift this burden currently borne by the programmer.

A standard technique to solve the first problem (i.e., intractability of validity checking of verification conditions) is to build automated, sound but incomplete verification engines for validating verification conditions, thus skirting the undecidability barrier. Several such techniques exist; for instance, for reasoning with quantified formulas, tactics such as model-based quantifier instantiation [28] are effective in practice, and they are known to be complete in certain settings [44]. In the realm of heap verification, the so-called *natural proof method* explicitly aims to provide automated and sound but incomplete methods for checking validity of verification conditions with specifications in separation logic [12,44,53,55].

Turning to the second problem of invariant generation, several techniques have emerged that are able to automatically synthesize invariants if the verification conditions fall in a decidable logic. Prominent among these are interpolation [46] and IC3/PDR [6,20]. Moreover, a class of counterexample guided inductive synthesis (CEGIS) methods have emerged recently, including the ICE learning model [26] for which various instantiations exist [10,22,26,27,38,58]. The key feature of the latter methods is a program-agnostic, data-driven learner that learns invariants in tandem with a verification engine that provides concrete program configurations as counterexamples to incorrect invariants.

Although classical invariant synthesis techniques, such as HOUDINI [23], are sometimes used with incomplete verification engines, to the best of our knowledge there is no fundamental argument as to why this should work in general. In fact, we are not aware of any systematic technique for synthesizing invariants when the underlying verification problem falls in an undecidable theory. When verification is undecidable and the engine resorts to sound but incomplete heuristics to check validity of verification conditions, it is unclear how to extend interpolation/IC3/PDR techniques to this setting. Data-driven learning of invariants is also hard to extend since the verification engine typically cannot generate a concrete model for the negation of verification conditions when verification fails. Hence, it cannot produce the concrete configurations the learner needs.

The main contribution of this paper is a general, learning-based invariant synthesis framework that learns invariants using non-provability information provided by verification engines. Intuitively, when a conjectured invariant results in verification conditions that cannot be proven, the idea is that the verification engine must return information that generalizes the reason for non-provability, hence pruning the space of future conjectured invariants.

Our framework, which we present in Sect. 2, assumes a verification engine for an undecidable theory $\mathscr{U}$ that reduces verification conditions to a decidable theory $\mathscr{D}$ (e.g., using

heuristics such as bounded quantifier instantiation to remove universal quantifiers, function unfolding to remove recursive definitions, and so on) that permits producing models for satisfiable formulas. The translation is assumed to be conservative in the sense that if the translated formula in $\mathscr{D}$ is valid, then we are assured that the original verification condition is $\mathscr{U}$-valid. If the verification condition is found to be not $\mathscr{D}$-valid (i.e., its negation is satisfiable), on the other hand, our framework describes how to extract non-provability information from the $\mathscr{D}$-model. This information is encoded as conjunctions and disjunctions in a Boolean theory $\mathscr{B}$, called *conjunctive/disjunctive non-provability information (CD-NPI)*, and communicated back to the learner. To complete our framework, we show how the formula-driven problem of learning expressions from CD-NPI constraints can be reduced to the data-driven ICE model. This reduction allows us to use a host of existing ICE learning algorithms and results in a robust invariant synthesis framework that guarantees to synthesize a provable invariant if one exists.

However, our CD-NPI learning framework has non-trivial requirements on the verification engine, and building or adapting appropriate engines is not straightforward. To show that our framework is indeed applicable and effective in practice, our second contribution is an application of our technique to two verification domains where the underlying verification is undecidable:

- Our first setting, presented in Sect. 3, is the verification of dynamically manipulated data-structures against rich logics that combine properties of structure, separation, arithmetic, and data. We show how *natural proof verification engines* [44,53], which are sound but incomplete verification engines that translate a powerful undecidable separation logic called DRYAD to decidable logics, can be fit into our framework. Moreover, we implement a prototype of such a verification engine on top of the program verifer Boogie [3] and demonstrate that this prototype is able to fully automatically verify a large suite of benchmarks, containing standard algorithms for manipulating singly and doubly linked lists, sorted lists, as well as balanced and sorted trees. Automatically synthesizing invariants for this suite of heap-manipulating programs against an expressive separation logic is very challenging, and we do not know of any other technique that can automatically prove all of them. In fact, heap-based reasoning is already challenging when invariants are given, and even then the number of automatic tools able to handle heap verification for rich separation logic specifications is low. For instance, the SL- COMP benchmarks for separation logic [62] are still at the level of logic solvers for entailment in separation logic, rather than for program verification. Thus, we have to leave a comparison to other approaches for future work.
- The second setting, presented in Sect. 4, addresses the verification of programs against specifications with universal quantification, which renders verification undecidable in general. In this situation, automated verification engines commonly use a variety of bounded quantifier instantiation techniques (such as E-matching, triggers, and model-based quantifier instantiation) to replace universal quantification by conjunctions over a specific set of terms. This soundly reduces satisfiability checking of the negated verification conditions to a decidable theory. Based on such techniques, we implement our framework and we show that it is able to effectively generate invariants that prove a challenging suite of programs correct against universally quantified specifications.

To the best of our knowledge, our framework is the first to systematically address the problem of invariant synthesis for incomplete verification engines that work by soundly reducing undecidable logics to decidable ones. We believe our experimental results provide the first evidence of the tractability of this important problem.

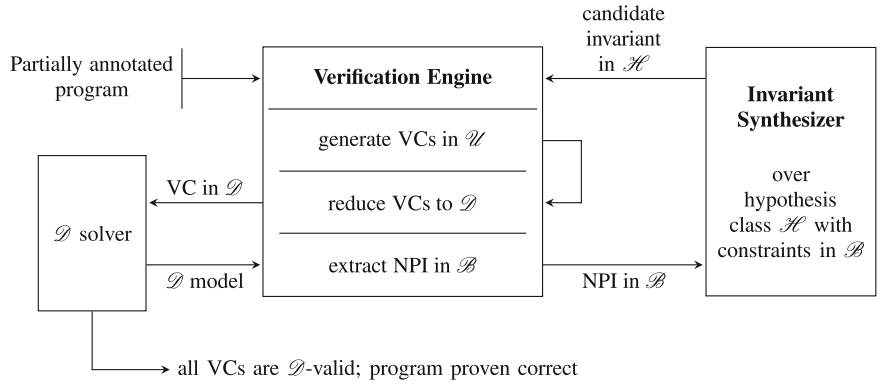
This work in an extended version of a conference paper [48]. Compared to the conference version, this work contains all proofs, an illustrative example of our framework, a section discussing the limitations of our framework (Sect. 2.4), as well as an extended description of the implementation of our framework for verifying heap-manipulating programs (Sect. 3). In addition, this work presents a second case study, namely the implementation of our framework for verifying programs against specifications with universal quantification as well as an empirical evaluation for this setting (Sect. 4), which was not contained in the conference paper.

Related Work

Techniques for invariant synthesis include abstract interpretation [15], interpolation [46], IC3 [6], predicate abstraction [2], abductive inference [19], as well as synthesis algorithms that rely on constraint solving [14,29,30]. Complementing them are data-driven invariant synthesis approaches that are based on machine learning. Examples include techniques that learn likely invariants, such as Daikon [21], and techniques that learn inductive invariants, such as HOUDINI [23], ICE [26], and Horn-ICE [10,22]. The latter typically requires a teacher that can generate counter-examples if the conjectured invariant is not adequate or inductive. Classically, this is possible only when the verification conditions of the program fall in decidable logics. In this paper, we investigate data-driven invariant synthesis for incomplete verification engines and show that the problem can be reduced to ICE learning if the learning algorithm learns from non-provability information and produces hypotheses in a class that is restricted to positive Boolean formulas over a fixed set of predicates. Data-driven synthesis of invariants has regained recent interest [25,26,38,51,52,57–61,63], and our work addresses an important problem of synthesizing invariants for programs whose verification conditions fall in undecidable fragments.

Our application to learning invariants for heap-manipulating programs builds upon the logic DRYAD [53,55], and the natural proof technique line of work for heap verification developed by Qiu et al. Techniques, similar to DRYAD, for automated reasoning of dynamically manipulated data structure programs have also been proposed in [11,12]. However, unlike our current work, none of these works synthesize heap invariants. Given invariant annotations in their respective logics, they provide procedures to validate if the verification conditions are valid. There has also been a lot of work on synthesizing invariants for separation logic using shape analysis [9,41,56]. However, most of these techniques are tailored for memory safety and shallow properties, but they do not handle rich properties that check full functional correctness of data structures as we do in this work. Interpolation has also been suggested recently to synthesize invariants involving a combination of data and shape properties [1]. It is, however, not clear how the technique can be applied to a more complicated heap structure, such as an AVL tree, where shape and data properties are not cleanly separated but are intricately connected. Recent work also includes synthesizing heap invariants in the logic from [33] by extending IC3 [34,35].

In this work, our learning algorithm synthesizes invariants over a fixed set of predicates. When all programs belong to a specific class, such as the class of programs manipulating data structures, these predicates can be uniformly chosen using templates. Investigating automated ways for discovering candidate predicates is a very interesting future direction. Related work in this direction includes recent works [51,52].



$\mathscr{H}$ – the hypothesis class of invariants
$\mathscr{U}$ – the underlying theory of the program, assumed to be undecidable
$\mathscr{D}$ – the theory that the verification engine soundly reduces verification conditions to; decidable and can produce models
$\mathscr{B}$ – the theory of propositional logic that the verification engine uses to communicate to the invariant synthesis engine

Fig. 1 A non-provability information (NPI) framework for invariant synthesis

2 An Invariant Synthesis Framework for Incomplete Verification Engines

In this section, we develop our framework for synthesizing inductive invariants for incomplete verification engines, using a counter-example guided inductive synthesis approach. We do this in a setting where the hypothesis space consists of formulas that are Boolean combinations of a fixed set of predicates $\mathscr{P}$, which need not be finite for the general framework (although we assume $\mathscr{P}$ to be a finite set of predicates when developing concrete learning algorithms later). For the rest of this section, let us fix a program P that is annotated with assertions (and possibly with some partial annotations describing pre-conditions, post-conditions, and assertions). Moreover, we say that a formula α is *weaker* (*stronger*) than a formula β in a logic $\mathscr{L}$ if $\vdash_{\mathscr{L}} \beta \Rightarrow \alpha$ ($\vdash_{\mathscr{L}} \alpha \Rightarrow \beta$) where $\vdash_{\mathscr{L}} \varphi$ means that φ is valid in $\mathscr{L}$.

Figure 1 depicts our general framework of invariant synthesis when verification is undecidable. We fix several parameters for our verification effort. First, let us assume a uniform signature for logics in terms of constant symbols, relation symbols, functions, and types. For simplicity of exposition, we use the same syntactic logic for the various logics $\mathscr{U}$, $\mathscr{D}$, $\mathscr{B}$ in our framework as well as for the logic $\mathscr{H}$ used to express invariants.

Let us fix $\mathscr{U}$ as the underlying theory that is ideally needed for validating the verification conditions that arise for the program; we presume validity of formulas in $\mathscr{U}$ is undecidable. Since $\mathscr{U}$ is an undecidable theory, the engine will resort to sound approximations (e.g., using bounded quantifier instantiations using mechanisms such as triggers [17], bounded unfolding of recursive functions, or natural proofs [44,53]) to reduce this logical task to a *decidable* theory $\mathscr{D}$. This reduction is assumed to be sound in the sense that if the resulting formulas in $\mathscr{D}$ are valid, then the verification conditions are valid in $\mathscr{U}$ as well. If a formula is found *not*

valid in $\mathscr{D}$, then we require that the logic solver for $\mathscr{D}$ returns a model for the negation of the formula.[1] Note that this model may not be a model for the negation of the formula in $\mathscr{U}$.

Moreover, we fix a hypothesis class $\mathscr{H}$ for invariants consisting of *positive* Boolean combination of predicates over a fixed set of predicates $\mathscr{P}$. Note that considering only *positive* formulas over $\mathscr{P}$ is not a restriction in general because one can always add negations of predicates to $\mathscr{P}$, thus effectively synthesizing any Boolean combination of predicates (in negation normal form). The restriction to positive Boolean formulas is in fact desirable as it allows restricting invariants to *not* negate certain predicates, which is useful when predicates have intuitionistic definitions (as several recursive definitions of heap properties do).

The invariant synthesis proceeds in rounds, where in each round the synthesizer proposes invariants in $\mathscr{H}$. The verification engine generates verification conditions in accordance to these invariants in the underlying theory $\mathscr{U}$. It then proceeds to translate them into the decidable theory $\mathscr{D}$, and gives them to a solver that decides their validity in the theory $\mathscr{D}$. If the verification conditions are found to be $\mathscr{D}$-valid, we have successfully proven the program correct by virtue of the fact that the verification engine reduced verification conditions in a sound fashion to $\mathscr{D}$.

However, if the formula is found not to be $\mathscr{D}$-valid, the solver returns a $\mathscr{D}$-model for its negation. The verification engine then extracts from this model certain *non-provability information (NPI)*, expressed as Boolean formulas in a Boolean theory $\mathscr{B}$, which captures more general reasons why the verification failed (the rest of this section is devoted to developing this notion of non-provability information). This non-provability information is communicated to the synthesizer, which then proceeds to synthesize a new conjecture invariant that satisfies the non-provability constraints provided in all previous rounds.

In order for the verification engine to extract meaningful non-provability information, we make the following natural assumption, called *normality*, which essentially states that the engine can do at least some minimal Boolean reasoning (if a Hoare triple is not provable, then Boolean weakenings of the precondition and Boolean strengthening of the post-condition must also be unprovable):

Definition 1 A verification engine is *normal* if it satisfies two properties:

1. If the engine cannot prove the validity of the Hoare triple $\{\alpha\}s\{\gamma\}$ and $\vdash_{\mathscr{B}} \delta \Rightarrow \gamma$, then it cannot prove the validity of the Hoare triple $\{\alpha\}s\{\delta\}$.
2. If the engine cannot prove the validity of the Hoare triple $\{\gamma\}s\{\beta\}$ and $\vdash_{\mathscr{B}} \gamma \Rightarrow \delta$, then it cannot prove the validity of the Hoare triple $\{\delta\}s\{\beta\}$.

Throughout this section, we use a running example to illustrate the components of our framework. Let us begin by introducing this example and specific logics $\mathscr{U}$, $\mathscr{D}$, and $\mathscr{B}$.

Example 1 Consider the program in Fig. 2. This program is taken from the Software Verification Competition [37] and computes the inverse B of a bijective (i.e., injective and surjective) mapping A. Note that the post-condition of this program expresses that the mapping B is injective. The program appears with the name "inverse" in Sect. 4.

To prove this program correct, one needs to specify adequate invariants at the loop header and before the return statement of the function *inverse*. We wish to synthesize these invariants.

For simplicity, let us assume we are provided with a small set $\mathscr{P} = \{p_1, p_2, p_3\}$ of predicates that serve as the basic building blocks for the invariants to be synthesized: p_1 at

[1] Note that our framework requires model construction in the theory $\mathscr{D}$. Hence, incomplete logic solvers for $\mathscr{U}$ that simply time out after some time threshold or search for a proof of a particular kind and give up otherwise are not suitable candidates.

```
int A[], B[];
int N;      axiom (N > 0);
bool inImage(int i) { return true; }

procedure inverse()
requires (∀x,y. 0 ≤ x < y < N ⟹ A[x] ≠ A[y]); // A is injective
requires (∀x. 0 ≤ x < N ∧ inImage(x) ⟹ (∃y. 0 ≤ y < N ∧ A[y] = x)); // A is surjective
ensures (∀x,y. 0 ≤ x < y < N ⟹ B[x] ≠ B[y]); // B is injective
{
  int i = 0;
  while (i < N)
  SynthesizeInv (∀x. 0 ≤ x < i ⟹ B[A[x]] = x); // p1
  {
        B[A[i]] = i;
        i = i + 1;
  }
  SynthesizeInv (∀x. 0 ≤ x < N ⟹ A[B[x]] = x, // p2
                  ∀x. 0 ≤ x < N ∧ inImage(x) ⟹ A[B[x]] = x); // p3
  return;
}
```

Fig. 2 Synthesizing invariants for the program that constructs an inverse B of a bijective (i.e., injective and surjective) mapping A, taken from the Verified software competition [37]

the loop header and p_2, p_3 before the return statement. Therefore, the task is to synthesize adequate invariants for this program over the predicates $\mathscr{P}$.[2]

For this specific verification task, we choose

– the universally quantified theory of arrays as undecidable theory $\mathscr{U}$.

Moreover, we assume a verification engine that soundly reduces verification conditions in $\mathscr{U}$ to

– the decidable theory $\mathscr{D}$ of linear arithmetic and uninterpreted functions by means of bounded quantifier instatiation.

Finally, we fix

– propositional logic as the Boolean theory $\mathscr{B}$.

Note the constant Boolean function *inImage* is crucially required to validate certain verification conditions by triggering appropriate quantifier instantiations in the surjectivity condition. Moreover, note that a verification engine using the theories $\mathscr{U}$ and $\mathscr{D}$ as above is normal. □

In Sect. 2.1, we now develop an appropriate language to communicate non-provability constraints, which allow the learner to appropriately weaken or strengthen a future hypothesis. It turns out that *pure conjunctions* and *pure disjunctions* over $\mathscr{P}$, which we term *CD-NPI constraints* (conjunctive/disjunctive non-provability information constraints), are sufficient for this purpose. We also describe concretely how the verification engine can extract this non-provability information from $\mathscr{D}$-models that witness that negations of VCs are satisfiable. Then, in Sect. 2.2, we show how to build learners for CD-NPI constraints by reducing this learning problem to another, well-studied learning framework for invariants called ICE learning. Finally, Sect. 2.3 argues the correctness of our framework, while Sect. 2.4 discusses its limitations.

[2] In general, one starts with a much larger set of candidate predicates that are automatically generated using program/specification-dependent heuristics.

2.1 Conjunctive/Disjunctive Non-provability Information

We assume that the underlying decidable theory $\mathscr{D}$ is stronger than propositional theory $\mathscr{B}$, meaning that every valid statement in $\mathscr{B}$ is valid in $\mathscr{D}$ as well. The reader may want to keep the following as a running example where $\mathscr{D}$ is the decidable theory of uninterpreted functions and linear arithmetic, say. In this setting, a formula is $\mathscr{B}$-valid if, when treating atomic formulas as Boolean variables, the formula is propositionally valid. For instance, $f(x) = y \Rightarrow f(f(x)) = f(y)$ will not be $\mathscr{B}$-valid though it is $\mathscr{D}$-valid, while $f(x) = y \vee \neg(f(x) = y)$ is $\mathscr{B}$-valid.

To formally define CD-NPI constraints and their extraction from a failed verification attempt, let us first introduce the following notation. For any $\mathscr{U}$-formula φ, let $approx(\varphi)$ denote the $\mathscr{D}$-formula that the verification engine generates such that the $\mathscr{D}$-validity of $approx(\varphi)$ implies the $\mathscr{U}$-validity of φ. Moreover, for any Hoare triple of the form $\{\alpha\}s\{\beta\}$, let $VC(\{\alpha\}s\{\beta\})$ denote the verification condition in $\mathscr{U}$ corresponding to the Hoare triple that the verification engine generates.

For the sake of a simpler exposition, let us assume that

1. the program has a single annotation hole A where we need to synthesize an inductive invariant to prove the program correct; and
2. every snippet s of the program for which a verification condition is generated has been augmented with a set of ghost variables $g_1, \ldots, g_n$ that track the predicates $p_1, \ldots, p_n$ over which the learner synthesizes the invariant (i.e., these ghost variables are assigned the values of the predicates).

As a shorthand notation, we denote the values of the ghost variables before the execution of the snippet s by $\mathbf{v} = \langle v_1, \ldots, v_n \rangle$ and their values after the execution of s by $\mathbf{v}' = \langle v_1', \ldots, v_n' \rangle$.

Suppose now that the learner conjectures an annotation γ as an inductive invariant for the annotation hole A, and the verification engine fails to prove the verification condition corresponding to a Hoare triple $\{\alpha\}s\{\beta\}$, where either α, β, or both could involve the synthesized annotation. This means that the negation of $approx(VC(\{\alpha\}s\{\gamma\}))$ is $\mathscr{D}$-satisfiable, and the verification engine needs to extract non-provability information from a model of it. To this end, the verification engine first extracts the values $\mathbf{v}$ and $\mathbf{v}'$ of the ghost variables before and after the execution of s. Then, it generates one of three different types of non-provability information, depending on where the annotation appears in a Hoare triple $\{\alpha\}s\{\beta\}$ (either in α, in β, or in both). We now handle all three cases individually.

– Assume the verification of a Hoare triple of the form $\{\alpha\}s\{\gamma\}$ fails (i.e., the verification engine cannot prove a verification condition where the pre-condition α is a user-supplied annotation and the post-condition is the synthesized annotation γ). Then, $approx(VC(\{\alpha\}s\{\gamma\}))$ is not $\mathscr{D}$-valid, and the decision procedure for $\mathscr{D}$ would generate a model for its negation.
  Since γ is a positive Boolean combination, the reason why $\mathbf{v}'$ does not satisfy γ is due to the variables mapped to $false$ by $\mathbf{v}'$, as any valuation extending this will not satisfy γ. Intuitively, this means that the $\mathscr{D}$-solver is not able to prove the predicates in $P_{false} = \{p_i \mid v_i' = false\}$. In other words, $\{\alpha\}s\{\bigvee P_{false}\}$ is unprovable (a witness to this fact is the model of the negation of $approx(VC(\{\alpha\}s\{\gamma\}))$ from which the values $\mathbf{v}'$ are derived). Note that any invariant γ' that is stronger than $\bigvee P_{false}$ will result in an unprovable verification condition due to the verification engine being normal. Consequently we can choose $\chi = \bigvee P_{false}$ as the weakening constraint, demanding that future invariants should not be stronger than χ.

The verification engine now communicates χ to the synthesizer, asking it never to conjecture in future rounds invariants γ'' that are stronger than χ (i.e., such that $\vDash_{\mathscr{B}} \gamma'' \Rightarrow \chi$).

- The next case is when a Hoare triple of the form $\{\gamma\}s\{\beta\}$ fails to be proven (i.e., the verification engine cannot prove a verification condition where the post-condition β is a user-supplied annotation and the pre-condition is the synthesized annotation γ). Using similar arguments as above, the *conjunction* $\eta = \bigwedge\{p_i \mid v_i = true\}$ of the predicates mapped to $true$ by $\mathbf{v}$ in the corresponding $\mathscr{D}$-model gives a stronger precondition η such that $\{\eta\}s\{\alpha\}$ is not provable. Hence, η is a valid *strengthening* constraint. The verification engine now communicates η to the synthesizer, asking it never to conjecture in future rounds invariants γ'' that are weaker than η (i.e., such that $\vDash_{\mathscr{B}} \eta \Rightarrow \gamma''$).
- Finally, consider the case when the Hoare triple is of the form $\{\gamma\}s\{\gamma\}$ and fails to be proven (i.e., the verification engine cannot prove a verification condition where the pre- and post-condition is the synthesized annotation γ). In this case, the verification engine can offer advice on how γ can be strengthened *or* weakened to avoid this model. Analogous to the two cases above, the verification engine extracts a pair of formulas (η, χ), called an *inductivity constraint*, based on the variables mapped to $true$ by $\mathbf{v}$ and to $false$ by $\mathbf{v}'$. The meaning of such a constraint is that the invariant synthesizer must conjecture in future rounds invariants γ'' such that either $\vDash_{\mathscr{B}} \eta \Rightarrow \gamma''$ or $\vDash_{\mathscr{B}} \gamma'' \Rightarrow \chi$ holds.

This leads to the following scheme, where γ denotes the conjectured invariant:

- When a Hoare triple of the form $\{\alpha\}s\{\gamma\}$ fails, the verification engine returns the $\mathscr{B}$-formula $\bigvee_{i|v_i'=false} p_i$ as a weakening constraint.
- When a Hoare triple of the form $\{\gamma\}s\{\beta\}$ fails, the verification engine returns the $\mathscr{B}$-formula $\bigwedge_{i|v_i=true} p_i$ as a strengthening constraint.
- When a Hoare triple of the form $\{\gamma\}s\{\gamma\}$ fails, the verification engine returns the pair $(\bigwedge_{i|v_i=true} p_i, \bigvee_{i|v_i'=false} p_i)$ of $\mathscr{B}$-formulas as an inductivity constraint.

It is not hard to verify that the above formulas are proper strengthening and weakening constraints in the sense that *any* inductive invariant must satisfy these constraints. This motivates the following form of non-provability information.

Definition 2 (CD-NPI Samples) Let $\mathscr{P}$ be a set of predicates. A *CD-NPI sample* (short for *conjunction-disjunction-NPI sample*) is a triple $\mathfrak{S} = (W, S, I)$ consisting of

- a finite set W of disjunctions over $\mathscr{P}$ (weakening constraints);
- a finite set S of conjunctions over $\mathscr{P}$ (strengthening constraints); and
- a finite set I of pairs, where the first element is a conjunction and the second is a disjunction over $\mathscr{P}$ (inductivity constraints).

An annotation γ is *consistent* with a CD-NPI sample $\mathfrak{S} = (W, S, I)$ if $\nvDash_{\mathscr{B}} \gamma \Rightarrow \chi$ for each $\chi \in W$, $\nvDash_{\mathscr{B}} \eta \Rightarrow \gamma$ for each $\eta \in S$, and $\nvDash_{\mathscr{B}} \eta \Rightarrow \gamma$ or $\nvDash_{\mathscr{B}} \gamma \Rightarrow \chi$ for each $(\eta, \chi) \in I$.

A CD-NPI learner is an effective procedure that synthesizes, given an CD-NPI sample, an annotation γ consistent with the sample. In our framework, the process of proposing candidate annotations and checking them repeats until the learner proposes a valid annotation or it detects that no valid annotation exists (e.g., if the class of candidate annotations is finite and all annotations are exhausted). We comment on using an CD-NPI learner in this iterative fashion in the next section.

Example 2 Let us continue Example 1 (on Page 6) and assume that the learner conjectures $\gamma_L = p_1$ as the loop invariant and $\gamma_R = p_2 \wedge p_3$ as the invariant at the return statement.

Moreover, suppose that the verification condition $VC(\{\gamma_L\}s\{\gamma_R\})$ along the path from the loop exit to the return statement, though valid in the undecidable theory $\mathscr{U}$, is not provable in the decidable theory $\mathscr{D}$ (i.e., $approx(VC(\{\gamma_L\}s\{\gamma_R\}))$ is not valid).

In this situation, the $\mathscr{D}$-solver returns a model $\mathscr{M}$ for the negation of the approximated verification condition $approx(VC(\{\gamma_L\}s\{\gamma_R\}))$. The verification engine now inspects this model and extracts the values of the predicates p_1, p_2, and p_3. As explained above, we assume that the program is equipped with ghost variables that track the values of all predicates (in our case g_1, g_2, and g_3), and a verification engine can simply extract the value of a predicate from the value of the corresponding ghost variable in the model.

For this particular verification condition, let us assume that $\mathscr{M}(g_1) = true$, $\mathscr{M}(g_2) = false$, and $\mathscr{M}(g_3) = true$, indicating that p_1 and p_3 hold in $\mathscr{M}$, while p_2 does not hold. From this information, the verification engine constructs a pair of formulas (η, χ) with $\eta = p_1$ and $\chi = p_2$, which it communicates as an inductivity constraint to the learner. Intuitively, this constraint means that the verification condition obtained by substituting γ_L with η and γ_R with χ is itself not provable. In subsequent rounds, the learner thus needs to conjecture only such invariants where γ_L is not weaker than η (i.e., $\vDash_{\mathscr{B}} p_1 \Rightarrow \gamma_L$) or γ_R is not stronger than χ (i.e., $\vDash_{\mathscr{B}} \gamma_R \Rightarrow p_2$). Note that both η and χ are formulas in the logic $\mathscr{B}$ since we assume a uniform signature for all logics (i.e., p_1, p_2, and p_3 are seen as propositional variables in $\mathscr{B}$). Moreover, note that the formula η is a conjunction with a single conjuct, whereas the formula χ is a disjunction with a single disjunct. □

2.2 Building CD-NPI Learners

Let us now turn to the problem of building efficient learning algorithms for CD-NPI constraints. To this end, we assume that the set of predicates $\mathscr{P}$ is finite.

Roughly speaking, the CD-NPI learning problem is to synthesize annotations that are positive Boolean combinations of predicates in $\mathscr{P}$ and that are consistent with a given CD-NPI sample. Though this is a learning problem where samples are *formulas*, in this section we reduce CD-NPI learning to a learning problem from *data*. In particular, we show that CD-NPI learning reduces to the ICE learning framework for learning positive Boolean formulas. The latter is a well-studied framework, and the reduction allows us to use efficient learning algorithms developed for ICE learning in order to build CD-NPI learners.

We now first recap the ICE-learning framework and then reduce CD-NPI learning to ICE learning. Finally, we briefly sketch how the popular HOUDINI algorithm can be seen as an ICE learning algorithm, which, in turn, allows us to use HOUDINI as an CD-NPI learning algorithm.

The ICE Learning Framework

Although the ICE learning framework [26] is a general framework for learning inductive invariants, we here consider the case of learning Boolean formulas. To this end, let us fix a set B of Boolean variables. Moreover, let $\mathscr{H}$ be a subclass of positive Boolean formulas over B (i.e., Boolean combinations of variables from B without negation). This class, called the hypothesis class, specifies the admissible solutions to the learning task.

The objective of the (passive) ICE learning algorithm is to learn a formula in $\mathscr{H}$ from positive examples, negative examples, and implication examples. More formally, if $\mathscr{V}$ is the set of valuations $v\colon B \to \{true, false\}$ (mapping variables in B to true or false), then an *ICE sample* is a triple $\mathscr{S} = (S_+, S_-, S_\Rightarrow)$ where $S_+ \subseteq \mathscr{V}$ is a set of *positive examples*,

$S_- \subseteq \mathscr{V}$ is a set of *negative examples*, and $S_\Rightarrow \subseteq \mathscr{V} \times \mathscr{V}$ is a set of *implications*. Note that positive and negative examples are *concrete* valuations of the variables in B, and the implication examples are pairs of such concrete valuations.

A formula φ is said to be *consistent with an ICE sample* $\mathscr{S}$ if it satisfies the following three conditions:[3] $v \models \varphi$ for each $v \in S_+$, $v \not\models \varphi$ for each $v \in S_-$, and $v_1 \models \varphi$ implies $v_2 \models \varphi$, for each $(v_1, v_2) \in S_\Rightarrow$.

In algorithmic learning theory, one distinguishes between *passive learning* and *iterative learning*. The former refers to a learning setting in which a learning algorithm is confronted with a finite set of data and has to learn a concept that is consistent with this data. Using our terminology, the *passive ICE learning problem* for a hypothesis class $\mathscr{H}$ is then

"given an ICE sample $\mathscr{S}$, find a formula in $\mathscr{H}$ that is consistent with $\mathscr{S}$".

Recall that we here require the learning algorithm to learn positive Boolean formulas, which is stricter than the original ICE framework [26].

Iterative learning, on the other hand, is the iteration of passive learning where new data is added to the sample from one iteration to the next. In a verification context, this new data is generated by the verification engine in response to incorrect annotations and used to guide the learning algorithm towards an annotation that is adequate to prove the program. To reduce our learning framework to ICE learning, it is therefore sufficient to reduce the (passive) CD-NPI learning problem described above to the passive ICE learning problem. We do this next.

Reduction of Passive CD-NPI Learning to Passive ICE Learning

Let $\mathscr{H}$ be a subclass of positive Boolean formulas. We reduce the CD-NPI learning problem for $\mathscr{H}$ to the ICE learning problem for $\mathscr{H}$. The main idea is to (a) treat each predicate $p \in \mathscr{P}$ as a Boolean variable for the purpose of ICE learning and (b) to translate a CD-NPI sample $\mathfrak{S}$ into an *equi-consistent* ICE sample $\mathscr{S}_\mathfrak{S}$, meaning that a positive Boolean formula is consistent with $\mathfrak{S}$ if and only if it is consistent with $\mathscr{S}_\mathfrak{S}$. Then, learning a consistent formula in the CD-NPI framework reduces to learning a consistent formula in the ICE learning framework.

The following lemma will us help translate between the two frameworks. Its proof is straightforward and follows from the following observation about any *positive* formula α: if a valuation v sets a larger subset of variables to true than v' does and $v' \models \alpha$, then $v \models \alpha$ holds as well.

Lemma 1 *Let v be a valuation of $\mathscr{P}$ and α be a positive Boolean formula over $\mathscr{P}$. Then, the following holds:*

- *$v \models \alpha$ if and only if $\vdash_\mathscr{B} (\bigwedge_{p|v(p)=true} p) \Rightarrow \alpha$ (and, therefore, $v \not\models \alpha$ if and only if $\nvdash_\mathscr{B} (\bigwedge_{p|v(p)=true} p) \Rightarrow \alpha$).*
- *$v \models \alpha$ if and only if $\nvdash_\mathscr{B} \alpha \Rightarrow (\bigvee_{p|v(p)=false} p)$.*

This motivates our translation, which relies on two functions, c and d. The function c translates a conjunction $\bigwedge J$, where $J \subseteq \mathscr{P}$, into the valuation

$$c\big(\bigwedge J\big) = v \text{ with } v(p) = true \text{ if and only if } p \in J.$$

The function d, on the other hand, translates a disjunction $\bigvee J$, where $J \subseteq \mathscr{P}$ is a subset of propositions, into the valuation

$$d\big(\bigvee J\big) = v \text{ with } v(p) = false \text{ if and only if } p \in J.$$

[3] In the following, $\models$ denotes the usual satisfaction relation.

By substituting v in Lemma 1 with $c(\bigwedge J)$ and $d(\bigvee J)$, respectively, one immediately obtains the following result.

Lemma 2 *Let $J \subseteq \mathscr{P}$ and α be a positive Boolean formula over $\mathscr{P}$. Then, the following holds:*

- $c\big(\bigwedge J\big) \models \alpha$ *if and only if* $\vdash_{\mathscr{B}} \bigwedge J \Rightarrow \alpha$ *(and, therefore,* $c\big(\bigwedge J\big) \not\models \alpha$ *if and only if* $\nvdash_{\mathscr{B}} \bigwedge J \Rightarrow \alpha$*).*
- $d\big(\bigvee J\big) \models \alpha$ *if and only if* $\nvdash_{\mathscr{B}} \alpha \Rightarrow \bigvee J$*.*

Based on the functions c and d, the translation of a CD-NPI sample into an equi-consistent ICE sample is as follows.

Definition 3 Given a CD-NPI sample $\mathfrak{S} = (W, S, I)$, the ICE sample $\mathscr{S}_{\mathfrak{S}} = (S_+, S_-, S_{\Rightarrow})$ is defined by

- $S_+ = \big\{d(\bigvee J) \mid \bigvee J \in W\big\}$;
- $S_- = \big\{c(\bigwedge J) \mid \bigwedge J \in S\big\}$; and
- $S_{\Rightarrow} = \big\{\big(c(\bigwedge J_1), d(\bigvee J_2)\big) \mid (\bigwedge J_1, \bigvee J_2) \in I\big\}$.

By virtue of the lemma above, we can now establish the correctness of the reduction from the CD-NPI learning problem to the ICE learning problem as follows.

Theorem 1 *Let $\mathfrak{S} = (W, S, I)$ be a CD-NPI sample, $\mathscr{S}_{\mathfrak{S}} = (S_+, S_-, S_{\Rightarrow})$ the ICE sample as in Definition 3, γ a positive Boolean formula over $\mathscr{P}$. Then, γ is consistent with $\mathfrak{S}$ if and only if γ is consistent with $\mathscr{S}_{\mathfrak{S}}$.*

Proof Let $\mathfrak{S} = (W, S, I)$ be an CD-NPI sample, and let $\mathscr{S}_{\mathfrak{S}} = (S_+, S_-, S_{\Rightarrow})$ the ICE sample as in Definition 3. Moreover, let γ be a positive Boolean formula. We prove Theorem 1 by considering each weakening, strengthening, and inductivity constraint together with their corresponding positive, negative, and implication examples individually.

- Pick a weakening constraint $\bigvee J \in W$, and let $v \in S_+$ with $v = d(\bigvee J)$ be the corresponding positive sample. Moreover, assume that γ is consistent with $\mathfrak{S}$ and, thus, $\nvdash_{\mathscr{B}} \gamma \Rightarrow \bigvee J$. By Lemma 2, this is true if and only if $d\big(\bigvee J\big) \models \gamma$. Hence, $v \models \gamma$. Conversely, assume that γ is consistent with $\mathscr{S}$. Thus, $v \models \gamma$, which means $d\big(\bigvee J\big) \models \gamma$. By Lemma 2, this is true if and only if $\nvdash_{\mathscr{B}} \gamma \Rightarrow \bigvee J$.
- Pick a strengthening constraint $\bigwedge J \in S$, and let $v \in S_-$ with $v = c(\bigwedge J)$ be the corresponding negative sample. Moreover, assume that γ is consistent with $\mathfrak{S}$ and, thus, $\nvdash_{\mathscr{B}} \bigwedge J \Rightarrow \gamma$. By Lemma 2, this is true if and only if $c\big(\bigwedge J\big) \not\models \gamma$. Hence, $v \not\models \gamma$. Conversely, assume that γ is consistent with $\mathscr{S}$. Thus, $v \not\models \gamma$, which means $c\big(\bigwedge J\big) \not\models \gamma$. By Lemma 2, this is true if and only if $\nvdash_{\mathscr{B}} \bigwedge J \Rightarrow \gamma$.
- Following the definition of implication, we split the proof into two cases, depending on whether $\nvdash_{\mathscr{B}} \bigwedge J \Rightarrow \gamma$ or $\nvdash_{\mathscr{B}} \gamma \Rightarrow \bigvee J$ (and $v_1 \not\models \gamma$ or $v_2 \models \gamma$ for the reverse direction). However, the proof of the former case uses the same arguments as the proof for strengthening constraints, while the proof of the latter case uses the same arguments as the proof for weakening constraints. Hence, combining both proofs immediately yields the claim. □

Let us illustrate the reduction of Definition 3 with an example.

Example 3 We continue Example 2 (on Page 10), in which the verification engine returned an inductivity constraint (η, χ) with $\eta = p_1$ and $\chi = p_2$. The task of the learner is now

to construct invariants where γ_L is not weaker than η (i.e., $\vdash_{\mathscr{B}} p_1 \Rightarrow \gamma_L$) or γ_R is not stronger than χ (i.e., $\vdash_{\mathscr{B}} \gamma_R \Rightarrow p_2$). To simplify this example, let us assume that the CD-NPI sample $\mathfrak{S} = (W, S, I)$ of the learner was initially empty and now only contains the returned inductivity constraint (p_1, p_2) (i.e., $W = S = \emptyset$ and $I = \{(p_1, p_2)\}$).

As described above, our learner works by reducing the CD-NPI learning problem to ICE learning. More precisely, the learner translates its CD-NPI sample $\mathfrak{S} = (W, S, I)$ to an equisatisfiable ICE sample $\mathscr{S}_{\mathfrak{S}} = (S_+, S_-, S_{\Rightarrow})$, where the elements of the ICE sample are valuations $v\colon \mathscr{P} \to \{true, false\}$. For the sake of simplicity, we write these functions as Boolean vectors of length three.

For the actual translation, the learner applies the functions c and d described above. Given the inductivity constraint (p_1, p_2), it constructs the implication constraint

$$\big(c(p_1), d(p_2)\big) = \big((1, 0, 0), (1, 0, 1)\big)$$

and adds to its ICE sample (recall that p_1 is in fact a conjunction with a single conjunct, while p_2 is a disjunction with a single disjunct). Thus, the learner obtains the ICE sample $\mathscr{S}_{\mathfrak{S}}$ with $S_+ = S_- = \emptyset$ and $S_{\Rightarrow} = \big((1, 0, 0), (1, 0, 1)\big)$. Note that $\mathfrak{S}$ and $\mathscr{S}_{\mathfrak{S}}$ are in fact equisatisfiable, as stated by Theorem 1. □

ICE Learners for Boolean Formulas

The reduction above allows us to use any ICE learning algorithm in the literature that synthesizes positive Boolean formulas. As we have mentioned earlier, we can add negations of predicates as first-class predicates and, hence, synthesize invariants over the more general class of all Boolean combinations as well.

The problem of passive ICE learning for one round, synthesizing a formula that satisfies the ICE sample, can usually be achieved efficiently and in a variety of ways. However, the crucial aspect is not the complexity of learning in one round but the *number* of rounds it takes to converge to an adequate invariant that proves the program correct. When the set $\mathscr{P}$ of candidate predicates is large (hundreds in our experiments), since the number of Boolean formulas over $\mathscr{P}$ is doubly exponential in $n = |\mathscr{P}|$, building an effective learner is not easy.

However, there is one class of formulas that are particularly amenable to efficient ICE learning: *conjunctions of predicates over* $\mathscr{P}$. For this specific case, ICE learning algorithms exist that promise learning the invariant in a linear number of rounds (provided one exists expressible as a conjunct over $\mathscr{P}$) [23,50]. Note that this learning is essentially finding an invariant in a hypothesis class $\mathscr{H}$ of size 2^n in a linear number of rounds.

HOUDINI [23] is such a learning algorithm for conjunctive formulas. Though it is typically seen as a particular way to synthesize invariants, it is a prime example of an ICE learner for conjuncts, as described in the work by Garg et al. [26]. In fact, Houdini is similar to the classical PAC learning algorithm for conjunctions [36], but extended to the ICE model. The time HOUDINI spends in each round is *polynomial* in the size of the sample, and it is guaranteed to converge to an invariant in at most $n + 1$ rounds (or reports that no conjunctive invariant over $\mathscr{P}$ exists). In our applications, we use this ICE learner to build a CD-NPI learner for conjunctions.

Example 4 Let us continue Example 3 and assume that the learner uses HOUDINI as ICE learning algorithm. Given the ICE sample $\mathscr{S}_{\mathfrak{S}}$, the learner now constructs the invariants $\gamma_L = p_1$ and $\gamma_R = p_3$. Note that these formulas are consistent with both the ICE sample $\mathscr{S}_{\mathfrak{S}}$ and the CD-NPI sample $\mathfrak{S}$.

With this new hypothesis, the verification engine can prove all verification conditions of the program valid in the theory $\mathscr{D}$. At this point, our invariant synthesis procedure terminates with these as adequate inductive invariants. □

2.3 Correctness and Convergence of the Invariant Learning Framework

To state the main result of this paper, let us first assume that the set $\mathscr{P}$ of predicates is finite. We comment on the case of infinitely many predicates at the end of this section.

Theorem 2 *Assume a normal verification engine for a program P to be given. Moreover, let $\mathscr{P}$ be a finite set of predicates over the variables in P and $\mathscr{H}$ a hypothesis class consisting of positive Boolean combinations of predicates in $\mathscr{P}$. If there exists an annotation in $\mathscr{H}$ that the verification engine can use to prove P correct, then the CD-NPI framework described in Sect. 2.1 is guaranteed to converge to such an annotation in finite time.*

Proof The proof proceeds in two steps. First, we show that a normal verification engine is *honest*, meaning that the non-provability information returned by such an engine does not rule out any adequate and provable annotation. Second, we show that any consistent learner (i.e., a learner that only produces consistent hypotheses), when paired with an honest verification engine, makes *progress* from one round to another. Finally, we combine both results to show that the framework eventually converges to an adequate and provable annotation.

Honesty of the verification engine We show honesty of the verification engine by contradiction.

- Suppose that the verification replies to a candidate invariant γ proposed by the learner with a weakening constraint χ because it could not prove the validity of the Hoare triple $\{\alpha\}s\{\gamma\}$. This effectively forces any future conjecture γ' to satisfy $\nvdash_{\mathscr{B}} \gamma' \Rightarrow \chi$.
  Now, suppose that there exists an invariant δ such that $\vdash_{\mathscr{B}} \delta \Rightarrow \chi$ and the verification engine can prove the validity of $\{\alpha\}s\{\delta\}$ (in other words, the adequate invariant δ is ruled out by the weakening constraint χ). Due to the fact that the verification engine is normal (in particular, by contraposition of Part 1 of Definition 1), this implies that the verification engine can also prove the validity of $\{\alpha\}s\{\chi\}$. However, this is a contradiction to χ being a weakening constraint.
- Suppose that the verification engine replies to a candidate invariant γ proposed by the learner with a strengthening constraint η because it could not prove the validity of the Hoare triple $\{\gamma\}s\{\beta\}$. This effectively forces any future conjecture γ to satisfy $\nvdash_{\mathscr{B}} \eta \Rightarrow \gamma'$.
  Now, suppose that there exists an invariant δ such that $\vdash_{\mathscr{B}} \eta \Rightarrow \delta$ and the verification engine can prove the validity of $\{\delta\}s\{\beta\}$ (in other words, the adequate invariant δ is ruled out by the weakening constraint η). Due to the fact that the verification engine is normal (in particular, by contraposition of Part 2 of Definition 1), this implies that the verification engine can also prove the validity of $\{\eta\}s\{\beta\}$. However, this is a contradiction to η being a strengthening constraint.
- Combining the arguments for weakening and strengthening constraints immediately results in a contradiction for the case of inductivity constraints as well.

Progress of the learner Now suppose that the learning algorithm is consistent, meaning that it always produces an annotation that is consistent with the current sample. Moreover, assume that the sample in iteration $i \in \mathbb{N}$ is $\mathfrak{S}_i$ and the learner produces the annotation γ_i. If γ_i is inadequate to prove the program correct, the verification engine returns a constraint c. The learner adds this constraint to the sample, obtaining the sample $\mathfrak{S}_{i+1}$ of the next iteration.

Since verification with γ_i failed, which is witnessed by c, we know that γ_i is not consistent with c. The next conjecture γ_{i+1}, however, is guaranteed to be consistent with $\mathfrak{S}_{i+1}$ (which contains c) because the learner is consistent. Hence, γ_i and γ_{i+1} are semantically different. Using this argument repeatedly shows that each annotation γ_i that a consistent learner has produced is semantically different from any previous annotation γ_j for $j < i$.

Convergence We first make two observations.

1. The number of semantically different hypotheses in the hypothesis space $\mathscr{H}$ is finite because the set $\mathscr{P}$ is finite. Recall that $\mathscr{H}$ is the class of all positive Boolean combinations of predicates in $\mathscr{P}$.
2. Due to the honesty of the verification engine, every annotation that the verification engine can use to prove the program correct is guaranteed to be consistent with any sample produced during the learning process.

Now, suppose that there exists an annotation that the verification engine can use to prove the program correct. Since the learner is consistent, all conjectures produced during the learning process are semantically different. Thus, the learner will at some point have exhausted all incorrect annotations in $\mathscr{H}$ (due to Observation 1). However, there exists at least one annotation that the verification engine can use to prove the program correct. Moreover, any such annotation is guaranteed to be consistent with the current sample (due to Observation 2). Thus, the annotation conjectured next is necessarily one that the verification engine can use to prove the program correct. □

Under certain realistic assumptions on the CD-NPI learning algorithm, Theorem 2 remains true even if the number of predicates is infinite. An example of such an assumption is that the learning algorithm always conjectures a smallest consistent annotation with respect to some fixed total order on $\mathscr{H}$. In this case, one can show that such a learner will at some point have proposed all inadequate annotation up to the smallest annotation the verification engine can use to prove the program correct. It will then conjecture this annotation in the next iteration. A correctness proof of this informal argument in a more general setting, called *abstract learning frameworks for synthesis*, has been given by Löding, Madhusudan, and Neider [43].

2.4 Limitations

One limiting factor is the effort and knowledge required to generate a set of suitable predicates over which to search for invariants. For several domains of programs and types of specifications, however, searching for invariants over a fixed set of predicates has proven to be extremely effective. Prominent examples include Microsoft's Static Driver Verifier [39,47] (specifically the underlying tool CORRAL [40] is an industry-strength tool that leverages exactly this approach) as well as GPUVerify [4], a fully-automated too to verify race freedom of GPU kernels. Note that for both examples, the predicates are generated automatically based on the code of the programs and/or the specification to verify—similar to what we did in Sect. 3.

In case the current set of predicates is insufficient to prove the program correct, a simple recourse is to negate predicates (that did not already occur negated) or add more complex predicates (e.g., as we do in Sect. 3). Although this increases the complexity of the verification task (as it enlarges the search space), the specific learning algorithm we use (i.e., HOUDINI) only requires a linear number of rounds in the number of predicates to find an invariant (or report that there is no conjunctive invariant that the verification engine can use to prove the

program correct). In fact, the increased number of rounds was never a limiting factor in our experiments (see Sect. 3 and "Appendix A)". Note that the succinctness of a certain class of formulas is also less of a concern compared to the size of the hypothesis space (since the number of rounds depends on the latter).

Finally, note that our framework is designed to prove the correctness of programs, but not do find bugs. As such, it does not necessarily terminate if the input program is not correct with respect to its specification. In our experience, however, the ICE framework can often detect specification violations, which manifest as inconsistent ICE samples (i.e., samples with internal inconsistencies such as an implication leading from a positive example to a negative example).

3 Application: Learning Invariants that Aid Natural Proofs for Heap Reasoning

We now develop an instantiation of our learning framework for verification engines based on natural proofs for heap reasoning [53,55].

Background: Natural Proofs and DRYAD

DRYAD [53,55] is a dialect of separation logic that allows expressing second order properties using recursive functions and predicates. DRYAD has a few restrictions, such as disallowing negations inside recursive definitions and in sub-formulas connected by spatial conjunctions (see Pek, Qiu, and Madhusudan [53]). However, it is expressive enough to define a variety of data-structures (singly and doubly linked lists, sorted lists, binary search trees, AVL trees, maxheaps, treaps) and recursive definitions over them that map to numbers (length, height, etc.). DRYAD also allows expressing properties about the data stored within the heap (the multiset of keys stored in lists, trees, etc.).

Natural proofs [53,55] is a sound but incomplete strategy for deciding satisfiability of DRYAD formulas. The first step the natural proof verifier performs is to convert all predicates and functions in a DRYAD-annotated program to classical logic. This translation introduces *heaplets* (modeled as sets of locations) explicitly in the logic. Furthermore, it introduces assertions demanding that the access of each method is contained to the heaplet implicitly defined by its pre-condition (taking into account newly allocated or freed nodes) and the modified heaplet at the end of the program precisely matches the heaplet implicitly defined by the post-condition.

In the second step, the natural proof verifier applies the following three transformations to the program: (a) it abstracts all recursive definitions on the heap using uninterpreted functions and introduces finite-depth unfoldings of these definitions at every place in the code where locations are dereferenced, (b) it models heaplets and other sets using the decidable theory of maps, and (c) it inserts frame reasoning explicitly in the code, which allows the verifier to derive that certain properties continue to hold across a heap update (or function call) using the heaplet that is being modified. Subsequently, the natural proof verifier translates the transformed program to BOOGIE [3], an intermediate verification language that handles proof obligations using automatic theorem provers (typically SMT solvers).

To perform both steps automatically, we used the tool VCDRYAD [53], which extends VCC [13] and operates on heap-manipulating C programs. The result is a BOOGIE program with no recursive definitions, where all verification conditions are in decidable logics,

and where a standard SMT solver, such as Z3 [18], can return models when formulas are satisfiable. The program in question can then be verified if supplied with correct inductive loop-invariants and adequate pre/post-conditions. We refer the reader to the work on DRYAD [55] and VCDRYAD [53] for more details.

Learning Heap Invariants

We have implemented a prototype[4] that consists of the entire VCDRYAD pipeline, which takes C programs annotated in DRYAD and converts them to BOOGIE programs via the natural proof transformations described above. We then apply our transformation to the ICE learning framework and pair BOOGIE with the ICE learning algorithm HOUDINI [23] that learns conjunctive invariants over a finite set of predicates (we describe shortly how these predicates are generated). After these transformations, BOOGIE satisfies the requirements on verification engines of our framework.

Given the DRYAD definitions of data structures, we automatically generated a set $\mathscr{P}$ of *predicates*, which serve as the basic building blocks of our invariants. Following other successful work on template-based verification, such as GPUVerify [4], we constructed the predicates from generic templates, which we obtained by inspecting a small number of programs. These templates were then instantiated using all combinations of program variables that occur in the program being verified. Figure 3 presents these templates in detail.

Our templates cover a fairly exhaustive set of predicates. This includes properties of the store (equality of pointer variables, equality and inequalities between integer variables, etc.), shape properties (singly and doubly linked lists and list segments, sorted lists, trees, binary search trees, AVL trees, treaps, etc.), and recursive definitions that map data structures to numbers, involving arithmetic relationships and set relationships (keys/data stored in a structure, lengths of lists and list segments, height of trees). In addition, our templates include predicates describing heaplets of various structures, which involve set operations, disjointness, and equalities. The structures and predicates are extensible, of course, to any recursive definition expressible in DRYAD.

The templates are grouped into three categories, roughly in increasing complexity. Predicates of Category 1 involve shape-related properties, predicates of Category 2 involve properties related to the keys stored in the data-structure, and predicates of Category 3 involve size-related properties (lengths of lists and heights of trees). Given a program to verify and its annotations, we choose the category of templates depending on whether the specification refers to shape only, shapes and keys, or shapes, keys, and sizes (choosing a category includes the predicates of lower category as well). Then, predicates are automatically generated by instantiating the templates with all (combinations of) program variables. This approach gives us a fairly fine-grained control over the set of predicates used in the verification process.

Benchmarks

We have evaluated our prototype on ten benchmark suites that contain standard algorithms on dynamic data structures, such as searching, inserting, or deleting items in lists and trees. These benchmarks were taken from the following sources:

1. GNU C Library(glibc) singly/sorted linked lists;

[4] This prototype as well as the prototype of Sect. 4 for learning universally quantified invariants is publicly available as one artifact on figshare [49]. This artifact also contains our benchmarks and all scripts required to replicate the results in this paper.

$$x, y \in \mathit{PointerVars} \quad \mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathit{PointerVars}^* \quad \mathit{pf} \in \mathit{PointerFields} \quad \mathit{df} \in \mathit{DataFields}$$
$$i, j \in \mathit{IntegerVars} \cup \{0, \mathsf{IntMax}, \mathsf{IntMin}\}$$

$$\mathit{listshape}(\mathbf{x}) := \mathsf{LinkedList}(x_1) \mid \mathsf{DoublyLinkedList}(x_1) \mid \mathsf{SortedLinkedList}(x_1) \mid \mathsf{LinkedListSeg}(x_1, x_2) \mid \mathsf{DoublyLinkedListSeg}(x_1, x_2)$$

$$\mathit{treeshape}(x) := \mathit{BST}(x) \mid \mathit{AVLtree}(x) \mid \mathit{Treap}(x)$$

$$\mathit{shape}(\mathbf{x}) := \mathit{listshape}(\mathbf{x}) \mid \mathit{treeshape}(\mathbf{x})$$

$$\mathit{size}(\mathbf{x}) := \mathit{listshape}\_\mathsf{length}(\mathbf{x}) \mid \mathit{treeshape}\_\mathsf{height}(\mathbf{x})$$

Category 1

$x = \mathsf{nil}$	$x = y$
$x \neq \mathsf{nil}$	$x \neq y$
$\mathit{shape}(\mathbf{x})$	$x.\mathit{pf} = \mathsf{nil}$
$x \in \mathit{shape}\_\mathsf{heaplet}(\mathbf{y})$	$x.\mathit{pf} \neq \mathsf{nil}$
$x \notin \mathit{shape}\_\mathsf{heaplet}(\mathbf{y})$	$x.\mathit{pf} = y$
$\mathit{shape}\_\mathsf{heaplet}(\mathbf{x}) \cap \mathit{shape}\_\mathsf{heaplet}(\mathbf{y}) = \emptyset$	$x.\mathit{pf} \neq y$

Category 2

$i \in \mathit{shape\_key}\_\mathsf{set}(\mathbf{x})$	$x.\mathit{df} = i$
$i \notin \mathit{shape\_key}\_\mathsf{set}(\mathbf{x})$	$x.\mathit{df} \neq i$
$\mathit{shape\_key}\_\mathsf{set}(\mathbf{x}) \leq_{\mathsf{set}} \{i\}$	$x.\mathit{df} \leq i$
$\mathit{shape\_key}\_\mathsf{set}(\mathbf{x}) \geq_{\mathsf{set}} \{i\}$	$x.\mathit{df} \geq i$
$\mathit{shape\_key}\_\mathsf{set}(\mathbf{x}) \leq_{\mathsf{set}} \{y.\mathit{df}\}$	$x.\mathit{df} = y.\mathit{df}$
$\mathit{shape\_key}\_\mathsf{set}(\mathbf{x}) \geq_{\mathsf{set}} \{y.\mathit{df}\}$	$x.\mathit{df} \neq y.\mathit{df}$
$\mathit{shape\_key}\_\mathsf{set}(\mathbf{x}) = \mathit{shape\_key}\_\mathsf{set}(\mathbf{y})$	$x.\mathit{df} \leq y.\mathit{df}$
$\mathit{shape\_key}\_\mathsf{set}(\mathbf{x}) \leq_{\mathsf{set}} \mathit{shape\_key}\_\mathsf{set}(\mathbf{y})$	$x.\mathit{df} \geq y.\mathit{df}$
$\mathit{shape\_key}\_\mathsf{set}(\mathbf{x}) \geq_{\mathsf{set}} \mathit{shape\_key}\_\mathsf{set}(\mathbf{y})$	
$\mathit{shape\_key}\_\mathsf{set}(\mathbf{x}) = \mathit{shape\_key}\_\mathsf{set}(\mathbf{y}) \cup \mathit{shape\_key}\_\mathsf{set}(\mathbf{z})$	

Category 3

$\mathit{size}(\mathbf{x}) = i - j$	$\mathit{size}(\mathbf{x}) = i$
$\mathit{size}(\mathbf{x}) - \mathit{size}(\mathbf{y}) = i$	$\mathit{size}(\mathbf{x}) \leq i$
$\mathit{size}(\mathbf{x}) - \mathit{size}(\mathbf{y}) = i - j$	$\mathit{size}(\mathbf{x}) \geq i$

Fig. 3 Templates of DRYAD predicates. The operator $\leq_{\mathsf{set}}$ denotes comparison between integer sets, where $A \leq_{\mathsf{set}} B$ if and only if $x \leq y$ holds for all $x \in A$ and $y \in B$. The operator $\geq_{\mathsf{set}}$ is similarly defined. Shape properties such as LinkedList, AVLtree, and so on are recursively defined in DRYAD (not shown here) and are extensible to any class of DRYAD-definable shapes. Similarly, the definitions related to keys stored in a data structure and the sizes of data structures also stem from recursive definitions in DRYAD

2. GNU C Library(glibc) doubly linked lists;
3. OpenBSD SysQueue;
4. GRASSHOPPER [54] singly linked lists;
5. GRASSHOPPER [54] doubly linked lists;
6. GRASSHOPPER [54] sorted linked lists;
7. VCDRYAD [53] sorted linked lists;
8. VCDRYAD [53] binary search trees, AVL trees, and treaps;
9. AFWP [33] singly/sorted linked lists; and
10. ExpressOS [45] MemoryRegion.

The specifications for these programs are generally checks for their full functional correctness, such as preserving or altering shapes of data structures, inserting or deleting keys, filtering or finding elements, as well as sortedness of elements. The specifications, hence, involve separation logic with arithmetic as well as recursive definitions that compute numbers (such as lengths and heights), data-aggregating recursive functions (such as multisets of keys stored in the data structures), and complex combinations of these properties (e.g., to specify binary search trees, AVL trees, and treaps). All programs are annotated in DRYAD, and checking validity of the resulting verification conditions is undecidable.

From these benchmark suites, we first picked all programs that contained iterative loops and erased the user-provided loop invariants. We also selected some programs that were purely recursive and where the contract for the function had manually been strengthened to make the verification succeed. We weakened these contracts to only state the specification (typically by removing formulas in the post-conditions of recursively called functions) and introduced annotation holes instead. The goal was to synthesize strengthenings of these contracts that allow proving the program correct (we left out programs where weakening was unnatural). We also chose five straight-line programs, deleted their post-conditions, and evaluated whether our prototype was able to learn post-conditions for them (since our conjunctive learner learns the strongest invariant expressible as a conjunct, we can use our framework to synthesize post-conditions as well). Since our framework requires a verification engine that terminates when given incorrect invariants, we had to modify some axiomatizations for sets (to handle heaplets). We were unable to make a small number of programs work with these simpler axioms and had to exclude them. In total, we obtained 83 routines.[5]

After removing annotations from the benchmarks, we automatically inserted appropriate predicates over which to build invariants and contracts as described above. For all benchmark suites, conjunctions of these predicates were sufficient to prove the programs correct.

Experimental Results

We performed all experiments sequentially on a single core of an Intel Core i5-6600 3.3 GHz CPU with 16 GB of RAM running Debian GNU/Linux 9.12 (stretch). For each benchmark, we limited the memory available to the underlying SMT solver (Z3 [18] in our case) to 1 GB. This amount was sufficient for all benchmarks.

The box plots in Fig. 4 summarize the results of this empirical evaluation aggregated by benchmark suite (full details can be found in "Appendix A"). This includes the time required to verify the programs, the number of automatically inserted base predicates (i.e., $|\mathscr{P}|$), and the number of iterations in the learning process. Each box in the diagrams shows the lower and upper quartile (left and right border of the box, respectively), the median (line within the box), as well as the minimum and maximum (left and right whisker, respectively).

Our prototype was successful in learning invariants and contracts for all 83 programs. The fact that the median time for a great majority of benchmark suites is less than 10 *s* shows that our technique is extremely effective in finding inductive DRYAD invariants. Despite many examples starting with hundreds of base predicates, which suggests a worst-case complexity of hundreds of iterations, the learner was able to learn with much fewer iterations. Moreover,

[5] Please note that we did not consider benchmarks from the Software Verification Competition [5] because these programs tend to have simple assertions (e.g., no null-pointer dereference) or pointer arithmetic assertions, which typically can be proven correct using decidable reasoning. Our framework, however, targets expressive specifications written in separation logic, where verification conditions do not fall in a decidable theory.

Benchmark suite	1	2	3	4	5	6	7	8	9	10
Number of programs	16	9	3	8	8	11	3	11	9	5
Max. category of templates	3	3	1	1	1	2	2	3	2	1

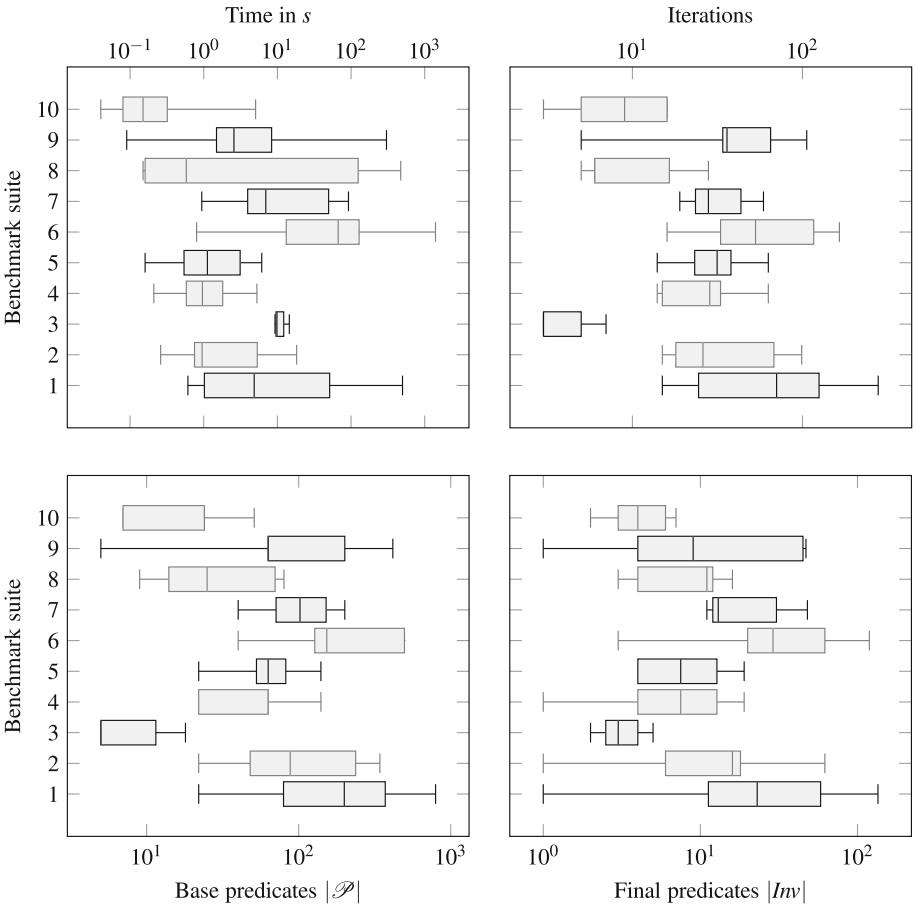


Fig. 4 Aggregated experimental results of the DRYAD benchmarks

the number of predicates in the final invariant is small. This shows that non-provability information of our framework provide much more information than the worst-case suggests.

To the best of our knowledge, our prototype is currently the only tool able of fully automatically verifying these challenging benchmark suites. We must emphasize, however, that there are subsets of these benchmarks that can be verified by reformulating the verification problem in decidable fragments of separation logic—we refer the reader to the related work in Sect. 1 for a survey of such work. Our goal in this evaluation, however, is not to compete with other, mature tools on a subset of benchmarks, but to measure the efficacy of our proposed CD-NPI-based invariant synthesis framework on the complete benchmark set.

4 Application: Learning Invariants in the Presence of Bounded Quantifier Instantiation

Software verification must deal with quantification in numerous application domains. For instance, quantifiers are often needed for axiomatizing theories that are not already equipped with decision procedures, for specifying properties of unbounded data structures and dynamically allocated memory, as well as for defining recursive properties of programs. For instance, the power of two function can be axiomatized using quantifiers:

$$pow_2(0) = 1 \ \wedge \ \forall n \in \mathbb{N}\colon n > 0 \Rightarrow pow_2(n) = 2 \cdot pow_2(n-1).$$

Despite the fact that various important first-order theories are undecidable (e.g., the first-order theory of arithmetic with uninterpreted functions), modern SMT solvers implement a host of heuristics to cope with quantifier reasoning. Quantifier instantiation, including pattern-based quantifier instantiation (e.g., E-matching [16]) and model-based quantifier instantiation [28], are particularly effective heuristics in this context. The key idea of instantiation-based heuristics is to instantiate universally quantified formulas with a finite number of ground terms and then check for validity of the resulting quantifier-free formulas (whose theory needs to be decidable). The exact instantiation of ground terms varies from method to method, but most instantiation-based heuristics are necessarily incomplete in general due to the undecidability of the underlying decision problems.

We can apply invariant synthesis framework for verification engines that employ quantifier instantiation in the following way. Assume that $\mathscr{U}$ is an undecidable first-order theory allowing uninterpreted functions and that $\mathscr{D}$ is its decidable quantifier-free fragment. Then, quantifier instantiation can be seen as a transformation of a $\mathscr{U}$-formula φ (potentially containing quantifiers) into a $\mathscr{D}$-formula $approx(\varphi)$ in which all existential quantifiers have been eliminated (e.g., using skolemization) and all universal quantifiers have been replaced by finite conjunctions over ground terms. [6] This means that if the $\mathscr{D}$-formula $approx(\varphi)$ is valid, then the $\mathscr{U}$-formula φ is valid as well. On the other hand, if $approx(\varphi)$ is not valid, one cannot deduce any information about the validity of φ. However, a $\mathscr{D}$-model of $approx(\varphi)$ can be used to derive non-provability information as described in Sect. 2.1.

Benchmarks

Our benchmark suite consists of twelve slightly simplified programs from IronFleet [31] (provably correct distributed systems), the Verified Software Competition [37], ExpressOS [45] (a secure operating system for mobile devices), and tools for sparse matrix multiplication [8]. In these programs, quantifiers are used to specify recursively defined predicates, such as $power(n, m)$ and $sum(n)$, as well various array properties, such as no duplicate elements, periodic properties of array elements, and bijective (injective and surjective) maps. All these specifications are undecidable in general. In particular, the array specifications fall outside of the decidable array property fragment [7] because they involve strict comparison between universally quantified index variables, array accesses in the index guard, nested array accesses (e.g., $a_1[a_2[i]]$), arithmetic expressions over universally quantified index variables, and alternation of universal and existential quantifiers.

From this benchmark suite, we erased the user-defined loop invariants and generated a set of predicates that serve as the building blocks of our invariants. To this end, we used the

[6] Quantifier instantiation is usually performed iteratively, but we here abstract away from this fact.

Table 1 Experimental results of the quantifier instantiation benchmarks. The column $|\mathscr{P}|$ refer to the number of automatically inserted base predicates, the column *# Iterations* to the number of iterations of the teacher and learner, and the column $|Inv|$ to the number of predicates in the inferred invariant

Program	$\lvert\mathscr{P}\rvert$	# Iterations	$\lvert Inv\rvert$	Time in s
inverse	414	126	73	8.05
power2	109	55	34	1.97
powerN	160	60	31	12.11
recordArraySplit	1264	49	51	52.99
recordArrayUnzip	222	17	25	0.73
removeDuplicates	280	67	86	4.00
setFind	492	74	136	2.43
setInsert	556	73	188	5.96
sparseMatrixGen	816	278	90	20.22
sparseMatrixMul	768	313	91	13.29
sum	128	40	22	0.90
sumMax	192	61	45	3.74

pre/post-conditions of the program being verified as templates from which the actual predicates were generated—as in the case of DRYAD benchmarks, the templates were instantiated using all combinations of program variables that occur in the program. Additionally, we generated predicates for octagonal constraints over the integer variables in the programs (i.e., relations between two integer variables of the form $\pm x \pm y \leq c$). For programs involving arrays, we also generated octagonal constraints over array access expressions that appear in the program.

Experimental Results

We have implemented a prototype[7] based on BOOGIE [3] and Z3 [18] as the verification engine and HOUDINI [23] as a conjunctive ICE learning algorithm. As in the case of the DRYAD benchmarks, all experiments were conducted sequentially on a single core of an Intel Core i5-6600 3.3 GHz CPU with 16 GB of RAM running Debian GNU/Linux 9.12 (stretch). Again, we limited the memory available to the underlying SMT solver to 1 GB. The results of these experiments are listed in Table 1.

As can be seen from this table, our prototype was effective in finding inductive invariants and was able to prove each program correct in less than 1 min (in 75 % of the programs in less than 10 s). Despite having hundreds of base predicates in many examples, which in turn suggests a worst-case complexity of hundreds of rounds, the learner was able to learn an inductive invariant with much fewer rounds. As in the case of the DRYAD benchmarks, the non-provability information provided by the verification engine provided much more information than the worst-case suggests.

[7] As mentioned before, this prototype, the prototype of Sect. 3 for learning heap invariants, all benchmarks, and all scripts to replicate the results in this paper are publicly available as one artifact on figshare [49].

5 Conclusions and Future Work

We have presented a learning-based framework for invariant synthesis in the presence of sound but incomplete verification engines. To prove that our technique is effective in practice, we have successfully applied it two important and challenging verification setting: verifying heap-manipulating programs against specifications expressed in an expressive and undecidable dialect of separation logic and verifying programs against specifications with universal quantification. In particular for the former setting, we are not aware of any other technique that can handle our extremely challenging benchmark suite.

Several future research directions are interesting. First, the framework we have developed is based on the principle of counterexample-guided inductive synthesis, where the invariant synthesizer synthesizes invariants using non-provability information but does not directly work on the program's structure. It would be interesting to extend white-box invariant generation techniques such as interpolation/IC3/PDR, working using $\mathscr{D}$ (or $\mathscr{B}$) abstractions of the program directly in order to synthesize invariants for them. Second, in the CD-NPI learning framework we have put forth, it would be interesting to change the underlying logic of communication $\mathscr{B}$ to a richer logic, say the theory of arithmetic and uninterpreted functions. The challenge here would be to extract non-provability information from the models to the richer theory, and pairing them with synthesis engines that synthesize expressions against constraints in $\mathscr{B}$. Finally, we think invariant learning should also include *experience* gained in verifying other programs in the past, both manually and automatically. A learning algorithm that combines logic-based synthesis with experience and priors gained from repositories of verified programs can be more effective.

Acknowledgements Open Access funding provided by Projekt DEAL.

A Detailed Results of the Heap Invariants Benchmark

See Table 2.

Table 2 Experimental results of the heap invariants benchmarks. The column $|\mathscr{P}|$ refer to the number of candidate predicates, the column *Cat.* corresponds to the category of predicates used, the column *# Iterations* to the number of iterations of the teacher and learner, and the column |Inv| to the number of predicates in the inferred invariant

Program	$\|\mathscr{P}\|$	Cat.	# Iterations	|Inv|	Time in s
(1) GNU C Library(glibc) singly and sorted linked-list					
g_slist_copy	368	2	123	101	50.93
g_slist_find	48	2	18	9	0.68
g_slist_free	22	1	15	1	1.04
g_slist_index	237	3	68	57	5.55
g_slist_insert	464	2	160	50	211.87
g_slist_insert_before	795	2	279	114	499.70
g_slist_insert_sorted	520	2	193	135	200.85
g_slist_last	32	2	19	8	0.61
g_slist_length	54	3	20	12	0.83
g_slist_nth	88	3	26	17	0.97
g_slist_nth_data	342	3	99	62	7.89
g_slist_position	162	3	32	18	2.44
g_slist_remove	140	1	73	28	4.13
g_slist_remove_all	380	2	132	15	52.17
g_slist_remove_link	325	1	85	57	13.16
g_slist_reverse	117	2	58	6	4.13
(2) GNU C Library(glibc) doubly linked-list					
g_list_find	48	2	18	9	0.66
g_list_free	22	1	15	1	0.75
g_list_index	237	3	68	57	5.33
g_list_last	22	1	15	6	0.26
g_list_length	88	3	24	16	0.95
g_list_nth	88	3	26	17	0.95
g_list_nth_data	342	3	99	62	8.34
g_list_position	162	3	32	18	2.56
g_list_reverse	320	2	84	2	18.25
(3) OpenBSD SysQueue					
squeue_insert_head*	5	1	3	2	9.86
squeue_insert_tail*	5	1	3	3	14.51
squeue_remove_head*	18	1	7	5	9.27
(4) GRASShopper [54] singly linked-list					
sl_concat	63	1	26	15	0.77
sl_copy	63	1	32	12	2.35
sl_dispose	22	1	14	1	0.61
sl_filter	140	1	63	9	5.27
sl_insert	63	1	31	19	1.14
sl_remove	22	1	15	6	0.50
sl_reverse	63	1	36	4	1.63
sl_traverse	22	1	15	4	0.21

Table 2 continued

Program	$\lvert\mathscr{P}\rvert$	Cat.	# Iterations	|Inv|	Time in s
(5) GRASShopper [54] doubly linked-list					
dl_concat	63	1	26	15	0.60
dl_copy	63	1	32	12	2.83
dl_dispose	140	1	44	4	6.13
dl_filter	140	1	63	9	3.99
dl_insert	63	1	31	19	0.77
dl_remove	22	1	15	6	0.37
dl_reverse	63	1	36	4	1.46
dl_traverse	22	1	14	4	0.16
(6) GRASShopper [54] sorted linked-list					
sls_concat	153	2	38	27	10.03
sls_copy	496	2	144	94	1401.32
sls_dispose	40	2	16	3	1.34
sls_double_all	496	2	106	102	118.54
sls_filter	496	2	127	30	138.84
sls_insert	153	2	53	29	16.32
sls_merge	416	2	63	29	298.66
sls_remove	496	2	165	119	66.92
sls_reverse	102	2	28	13	19.62
sls_split	153	2	53	29	92.12
sls_traverse	40	2	18	9	0.80
(7) VCDryad [53] sorted linked-list					
find_last_sorted	40	2	19	11	0.94
reverse_sorted	102	2	28	13	6.96
sorted_insert_iter	201	2	59	48	92.26
(8) VCDryad [53] trees					
avl-delete-rec†	72	3	16	5	376.06
avl-find-smallest†	19	3	5	11	0.16
avl-insert-rec†	72	3	23	14	90.70
bst-delete-rec†	68	2	16	11	159.03
bst-find-rec†	23	2	6	9	0.47
bst-insert-rec†	68	2	28	16	55.25
traverse-inorder†	9	3	6	3	0.15
traverse-posttorder†	9	3	6	3	0.15
traverse-preorder†	9	3	6	3	0.15
treap-delete-rec†	80	3	17	13	472.12
treap-find-rec†	25	3	6	11	0.58
(9) AFWP [33] singly and sorted linked-list					
SLL-create	5	1	5	1	0.09
SLL-delete-all	22	1	14	1	4.39
SLL-delete	265	1	106	47	8.35

Table 2 continued

Program	$\lvert\mathscr{P}\rvert$	Cat.	# Iterations	|Inv|	Time in s
SLL-filter	63	1	34	9	1.87
SLL-find	140	1	53	45	2.57
SLL-insert	201	2	65	26	44.00
SLL-last	63	1	34	9	1.06
SLL-merge	416	2	71	46	302.69
SLL-reverse	63	1	36	4	1.49
(10) ExpressOS [45] MemoryRegion					
memory_region_create	7	1	3	7	0.04
memory_region_find	24	1	16	2	0.15
memory_region_init*	7	1	5	4	0.08
memory_region_insert	51	1	16	3	0.32
split_memory_region*	24	1	9	6	5.06

A †Indicates contract strengthening, while a * indicates post condition learning

References

1. Albarghouthi, A., Berdine, J., Cook, B., Kincaid, Z.: Spatial interpolants. In: J. Vitek (ed.) Programming Languages and Systems—24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11–18, 2015. Proceedings, Lecture Notes in Computer Science, vol. 9032, pp. 634–660. Springer (2015). https://doi.org/10.1007/978-3-662-46669-8_26
2. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: M. Burke, M.L. Soffa (eds.) Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20–22, 2001, pp. 203–213. ACM (2001). https://doi.org/10.1145/378795.378846
3. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: F.S. de Boer, M.M. Bonsangue, S. Graf, W.P. de Roever (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1–4, 2005, Revised Lectures, Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17
4. Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: Gpuverify: a verifier for GPU kernels. In: G.T. Leavens, M.B. Dwyer (eds.) Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21–25, 2012, pp. 113–132. ACM (2012). https://doi.org/10.1145/2384616.2384625
5. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: A. Biere, D. Parker (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II, Lecture Notes in Computer Science, vol. 12079, pp. 347–367. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_21
6. Bradley, A.R.: Sat-based model checking without unrolling. In: R. Jhala, D.A. Schmidt (eds.) Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23–25, 2011. Proceedings, Lecture Notes in Computer Science, vol. 6538, pp. 70–87. Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7
7. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: E.A. Emerson, K.S. Namjoshi (eds.) Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8–10, 2006, Proceedings, Lecture Notes in Computer Science, vol. 3855, pp. 427–442. Springer (2006). https://doi.org/10.1007/11609773_28
8. Buluç, A., Fineman, J.T., Frigo, M., Gilbert, J.R., Leiserson, C.E.: Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In: F.M. auf der Heide, M.A. Bender

(eds.) SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11–13, 2009, pp. 233–244. ACM (2009). https://doi.org/10.1145/1583991.1584053
9. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (2011). https://doi.org/10.1145/2049697.2049700
10. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: Ice-based refinement type discovery for higher-order functional programs. In: D. Beyer, M. Huisman (eds.) Tools and Algorithms for the Construction and Analysis of Systems—24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018, Proceedings, Part I, *Lecture Notes in Computer Science*, vol. 10805, pp. 365–384. Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_20
11. Chin, W., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. **77**(9), 1006–1036 (2012). https://doi.org/10.1016/j.scico.2010.07.004
12. Chu, D., Jaffar, J., Trinh, M.: Automatic induction proofs of data-structures in imperative programs. In: D. Grove, S. Blackburn (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015, pp. 457–466. ACM (2015). https://doi.org/10.1145/2737924.2737984
13. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (eds.) Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17–20, 2009. Proceedings, Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_2
14. Colon, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: W.A.H. Jr., F. Somenzi (eds.) Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8–12, 2003, Proceedings, Lecture Notes in Computer Science, vol. 2725, pp. 420–432. Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_39
15. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: R.M. Graham, M.A. Harrison, R. Sethi (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
16. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005). https://doi.org/10.1145/1066100.1066102
17. de Moura, L.M., Bjørner, N.: Efficient e-matching for SMT solvers. In: F. Pfenning (ed.) Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17–20, 2007, Proceedings, Lecture Notes in Computer Science, vol. 4603, pp. 183–198. Springer (2007). https://doi.org/10.1007/978-3-540-73595-3_13
18. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: C.R. Ramakrishnan, J. Rehof (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings, Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
19. Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: A.L. Hosking, P.T. Eugster, C.V. Lopes (eds.) Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26–31, 2013, pp. 443–456. ACM (2013). https://doi.org/10.1145/2509136.2509511
20. Een, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: P. Bjesse, A. Slobodova (eds.) International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011, pp. 125–134. FMCAD Inc. (2011). http://dl.acm.org/citation.cfm?id=2157675
21. Ernst, M.D., Czeisler, A., Griswold, W.G., Notkin, D.: Quickly detecting relevant program invariants. In: C. Ghezzi, M. Jazayeri, A.L. Wolf (eds.) Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4–11, 2000, pp. 449–458. ACM (2000). https://doi.org/10.1145/337180.337240
22. Ezudheen, P., Neider, D., D'Souza, D., Garg, P., Madhusudan, P.: Horn-ice learning for synthesizing invariants and contracts. Proc. ACM Program. Lang. **2**(OOPSLA), 131:1–131:25 (2018). https://doi.org/10.1145/3276501
23. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for esc/java. In: J.N. Oliveira, P. Zave (eds.) FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of

Formal Methods Europe, Berlin, Germany, March 12–16, 2001, Proceedings, Lecture Notes in Computer Science, vol. 2021, pp. 500–517. Springer (2001). https://doi.org/10.1007/3-540-45251-6_29
24. Floyd, R.W.: Assigning Meanings to Programs. In: J.T. Schwartz (ed.) Proceedings of a Symposium on Applied Mathematics, Mathematical Aspects of Computer Science, vol. 19, pp. 19–31. American Mathematical Society (1967)
25. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: N. Sharygina, H. Veith (eds.) Computer Aided Verification—25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Proceedings, Lecture Notes in Computer Science, vol. 8044, pp. 813–829. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_57
26. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: A. Biere, R. Bloem (eds.) Computer Aided Verification—26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18–22, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8559, pp. 69–87. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_5
27. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: R. Bodik, R. Majumdar (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016, pp. 499–512. ACM (2016). https://doi.org/10.1145/2837614.2837664
28. Ge, Y., de Moura, L.M.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: A. Bouajjani, O. Maler (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26–July 2, 2009. Proceedings, Lecture Notes in Computer Science, vol. 5643, pp. 306–320. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_25
29. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: R. Gupta, S.P. Amarasinghe (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7–13, 2008, pp. 281–292. ACM (2008). https://doi.org/10.1145/1375581.1375616
30. Gupta, A., Rybalchenko, A.: Invgen: An efficient invariant generator. In: A. Bouajjani, O. Maler (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26–July 2, 2009. Proceedings, Lecture Notes in Computer Science, vol. 5643, pp. 634–640. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_48
31. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: proving practical distributed systems correct. In: E.L. Miller, S. Hand (eds.) Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4–7, 2015, pp. 1–17. ACM (2015). https://doi.org/10.1145/2815400.2815428
32. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
33. Itzhaky, S., Banerjee, A., Immerman, N., Nanevski, A., Sagiv, M.: Effectively-propositional reasoning about reachability in linked data structures. In: N. Sharygina, H. Veith (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Proceedings, Lecture Notes in Computer Science, vol. 8044, pp. 756–772. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_53
34. Itzhaky, S., Bjørner, N., Reps, T.W., Sagiv, M., Thakur, A.V.: Property-directed shape analysis. In: A. Biere, R. Bloem (eds.) Computer Aided Verification—26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18–22, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8559, pp. 35–51. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_3
35. Karbyshev, A., Bjørner, N., Itzhaky, S., Rinetzky, N., Shoham, S.: Property-directed inference of universal invariants or proving their absence. In: D. Kroening, C.S. Pasareanu (eds.) Computer Aided Verification—27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part I, *Lecture Notes in Computer Science*, vol. 9206, pp. 583–602. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_40
36. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press (1994). https://mitpress.mit.edu/books/introduction-computational-learning-theory
37. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M.A., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st verified software competition: Experience report. In: M.J. Butler, W. Schulte (eds.) FM 2011: Formal Methods—17th International Symposium on Formal Methods, Limerick, Ireland, June 20–24, 2011. Proceedings, Lecture Notes in Computer Science, vol. 6664, pp. 154–168. Springer (2011). https://doi.org/10.1007/978-3-642-21437-0_14

38. Krishna, S., Puhrsch, C., Wies, T.: Learning invariants using decision trees. CoRR (2015). arXiv:1501.04725
39. Lal, A., Qadeer, S.: Powering the static driver verifier using corral. In: S. Cheung, A. Orso, M.D. Storey (eds.) Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16–22, 2014, pp. 202–212. ACM (2014). https://doi.org/10.1145/2635868.2635894
40. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: P. Madhusudan, S.A. Seshia (eds.) Computer Aided Verification—24th International Conference, CAV 2012, Berkeley, CA, USA, July 7–13, 2012 Proceedings, Lecture Notes in Computer Science, vol. 7358, pp. 427–443. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_32
41. Le, Q.L., Gherghina, C., Qin, S., Chin, W.: Shape analysis via second-order bi-abduction. In: A. Biere, R. Bloem (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18–22, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8559, pp. 52–68. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_4
42. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: E.M. Clarke, A. Voronkov (eds.) Logic for Programming, Artificial Intelligence, and Reasoning—16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers, Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
43. Löding, C., Madhusudan, P., Neider, D.: Abstract learning frameworks for synthesis. In: M. Chechik, J. Raskin (eds.) Tools and Algorithms for the Construction and Analysis of Systems—22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings, Lecture Notes in Computer Science, vol. 9636, pp. 167–185. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_10
44. Löding, C., Madhusudan, P., Peña, L.: Foundations for natural proofs and quantifier instantiation. Proc. ACM Program. Lang. **2**(POPL), 10:1–10:30 (2018). https://doi.org/10.1145/3158098
45. Mai, H., Pek, E., Xue, H., King, S.T., Madhusudan, P.: Verifying security invariants in expressos. In: V. Sarkar, R. Bodik (eds.) Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16–20, 2013, pp. 293–304. ACM (2013). https://doi.org/10.1145/2451116.2451148
46. McMillan, K.L.: Interpolation and sat-based model checking. In: W.A.H. Jr., F. Somenzi (eds.) Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8–12, 2003, Proceedings, Lecture Notes in Computer Science, vol. 2725, pp. 1–13. Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
47. Microsoft: Static driver verifier. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier. Accessed 8 May (2020)
48. Neider, D., Garg, P., Madhusudan, P., Saha, S., Park, D.: Invariant synthesis for incomplete verification engines. In: D. Beyer, M. Huisman (eds.) Tools and Algorithms for the Construction and Analysis of Systems—24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018, Proceedings, Part I, Lecture Notes in Computer Science, vol. 10805, pp. 232–250. Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_13
49. Neider, D., Garg, P., Madhusudan, P., Saha, S., Park, D.: Prototype and benchmarks for "invariant synthesis for incomplete verification engines" (2018). https://doi.org/10.6084/m9.figshare.5928094
50. Neider, D., Saha, S., Garg, P., Madhusudan, P.: Sorcar: Property-driven algorithms for learning conjunctive invariants. In: B.E. Chang (ed.) Static Analysis—26th International Symposium, SAS 2019, Porto, Portugal, October 8–11, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11822, pp. 323–346. Springer (2019). https://doi.org/10.1007/978-3-030-32304-2_16
51. Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: C. Krintz, E. Berger (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13–17, 2016, pp. 42–56. ACM (2016). https://doi.org/10.1145/2908080.2908099
52. Pavlinovic, Z., Lal, A., Sharma, R.: Inferring annotations for device drivers from verification histories. In: D. Lo, S. Apel, S. Khurshid (eds.) Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3–7, 2016, pp. 450–460. ACM (2016). https://doi.org/10.1145/2970276.2970305
53. Pek, E., Qiu, X., Madhusudan, P.: Natural proofs for data structure manipulation in C using separation logic. In: M.F.P. O'Boyle, K. Pingali (eds.) ACM SIGPLAN Conference on Programming Language

Design and Implementation, PLDI '14, Edinburgh, United Kingdom, June 09–11, 2014, pp. 440–451. ACM (2014). https://doi.org/10.1145/2594291.2594325
54. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: N. Sharygina, H. Veith (eds.) Computer Aided Verification—25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Proceedings, Lecture Notes in Computer Science, vol. 8044, pp. 773–789. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_54
55. Qiu, X., Garg, P., Stefanescu, A., Madhusudan, P.: Natural proofs for structure, data, and separation. In: H. Boehm, C. Flanagan (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16–19, 2013, pp. 231–242. ACM (2013). https://doi.org/10.1145/2491956.2462169
56. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: A.W. Appel, A. Aiken (eds.) POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20–22, 1999, pp. 105–118. ACM (1999). https://doi.org/10.1145/292540.292552
57. Saha, S.: Learning frameworks for program synthesis. Ph.D. thesis, University of Illinois at Urbana-Champaign (2019)
58. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: A. Biere, R. Bloem (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18–22, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8559, pp. 88–105. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_6
59. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: M. Felleisen, P. Gardner (eds.) Programming Languages and Systems—22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16–24, 2013. Proceedings, Lecture Notes in Computer Science, vol. 7792, pp. 574–592. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_31
60. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Nori, A.V.: Verification as learning geometric concepts. In: F. Logozzo, M. Fähndrich (eds.) Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20–22, 2013. Proceedings, Lecture Notes in Computer Science, vol. 7935, pp. 388–411. Springer (2013). https://doi.org/10.1007/978-3-642-38856-9_21
61. Sharma, R., Nori, A.V., Aiken, A.: Interpolants as classifiers. In: P. Madhusudan, S.A. Seshia (eds.) Computer Aided Verification—24th International Conference, CAV 2012, Berkeley, CA, USA, July 7–13, 2012 Proceedings, Lecture Notes in Computer Science, vol. 7358, pp. 71–87. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_11
62. Sighireanu, M., Perez, J.A.N., Rybalchenko, A., Gorogiannis, N., Iosif, R., Reynolds, A., Serban, C., Katelaan, J., Matheja, C., Noll, T., Zuleger, F., Chin, W., Le, Q.L., Ta, Q., Le, T., Nguyen, T., Khoo, S., Cyprian, M., Rogalewicz, A., Vojnar, T., Enea, C., Lengal, O., Gao, C., Wu, Z.: SL-COMP: competition of solvers for separation logic. In: D. Beyer, M. Huisman, F. Kordon, B. Steffen (eds.) Tools and Algorithms for the Construction and Analysis of Systems—25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III, Lecture Notes in Computer Science, vol. 11429, pp. 116–132. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_8
63. Zhu, H., Nori, A.V., Jagannathan, S.: Learning refinement types. In: K. Fisher, J.H. Reppy (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1–3, 2015, pp. 400–411. ACM (2015). https://doi.org/10.1145/2784731.2784766