Invariant Synthesis for Incomplete Verification Engines

Daniel Neider[1(✉)], Pranav Garg[2], P. Madhusudan[3], Shambwaditya Saha[3], and Daejun Park[3]

[1] Max Planck Institute for Software Systems, Kaiserslautern, Germany
neider@mpi-sws.org
[2] Amazon India, Bangalore, India
[3] University of Illinois at Urbana-Champaign, Champaign, IL, USA

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 counter-example guided inductive synthesis principle (CEGIS) 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.

1 Introduction

The paradigm of *deductive verification* [15,22] 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 [34] (e.g., Vcc [8] and Dafny [29]).

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

© The Author(s) 2018
D. Beyer and M. Huisman (Eds.): TACAS 2018, LNCS 10805, pp. 232–250, 2018.
https://doi.org/10.1007/978-3-319-89960-2_13

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 verifications 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 [19] are effective in practice, and they are known to be complete in certain settings [30]. 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 [7,30,39,41].

Turning to the second problem of invariant generation, several techniques have emerged that can synthesize invariants automatically when validation of verification conditions fall in decidable classes. Prominent among these are interpolation [32] and IC3/PDR [4,12]. Moreover, a class of counter-example guided inductive synthesis (CEGIS) methods have emerged recently, including the ICE learning model [17] for which various instantiations exist [17,18,27,43]. 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 [14], 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 that 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 assumes a verification engine for an undecidable theory $\mathcal{U}$ that reduces verification conditions to a decidable theory $\mathcal{D}$ (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 $\mathcal{D}$ is valid, then we are assured that the original verification condition is $\mathcal{U}$-valid. If the verification condition

is found to be not $\mathcal{D}$-valid (i.e., its negation is satisfiable), on the other hand, our framework describes how to extract non-provability information from the $\mathcal{D}$-model. This information is encoded as conjunctions and disjunctions in a Boolean theory $\mathcal{B}$, 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 the verification of dynamically manipulated data-structures against rich logics that combine properties of structure, separation, arithmetic, and data. More precisely, we show how *natural proof verification engines* [30,39], 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. Thus, we have to leave a comparison to other approaches for future work.

In addition to verifying heap properties, we successfully applied our framework to the verification of programs against specifications with universal quantification, which occur, for instance, when defining recursive properties. Details can be found in an extended version of this paper [35], which also contains further material (e.g., proofs) that had to be omitted due to space constraints.

To the best of our knowledge, our technique 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.

Related Work

Techniques for invariant synthesis include abstract interpretation [10], interpolation [32], IC3 [4], predicate abstraction [2], abductive inference [11], as well as synthesis algorithms that rely on constraint solving [9,20,21]. Complementing them are data-driven invariant synthesis techniques based on learning, such as Daikon [13] that learn likely invariants, and Houdini [14] and ICE [17] that

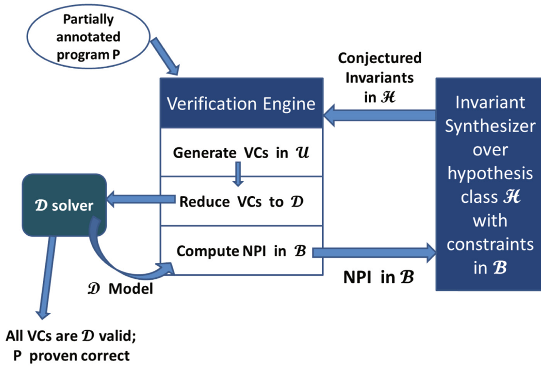
learn inductive invariants. The latter typically requires a teacher that can generate counter-examples if the conjectured invariant is not adequate or inductive. Classically, this is possible only when the verification conditions of the program fall in decidable logics. In this paper, we investigate 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 [16,17,27,37,38,43–47] 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 Dryad [39,41], 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 [6,7]. 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 [5,28,42]. However, most of them are tailored for memory safety and shallow properties rather than rich properties that check full functional correctness of data structures. 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 [23] by extending IC3 [24,25].

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 [37,38].

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 the setting where the hypothesis space consists of formulas that are Boolean combinations of a fixed set of predicates $\mathcal{P}$, which need not be finite for the general framework—when developing concrete learning algorithms later, we will assume $\mathcal{P}$ is a finite set of predicates. 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



$\mathcal{H}$ – The hypothesis class of invariants
$\mathcal{U}$ – The underlying theory of the program; undecidable
$\mathcal{D}$ – The theory that the verification engine soundly reduces verification conditions to; decidable and can produce models
$\mathcal{B}$ – 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

assertions). Moreover, we refer to a formula α being weaker (or stronger) than β in a logic $\mathcal{L}$, and by this we mean that $\vdash_{\mathcal{L}} \beta \Rightarrow \alpha$ (or $\vdash_{\mathcal{L}} \alpha \Rightarrow \beta$), respectively, where $\vdash_{\mathcal{L}} \varphi$ means that φ is valid in $\mathcal{L}$.

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 logic, in terms of constant symbols, relation symbols, functions, and types. We will, for simplicity of exposition, use the same syntactic logic for the various logics $\mathcal{U}$, $\mathcal{D}$, $\mathcal{B}$ in our framework as well as for the logic $\mathcal{H}$ used to express invariants.

Let us fix $\mathcal{U}$ as the underlying theory that is ideally needed for validating the verification conditions that arise for the program; we presume validity of formulas in $\mathcal{U}$ is undecidable. Since $\mathcal{U}$ is an undecidable theory, the engine will resort to sound approximations (e.g., using bounded quantifier instantiations using mechanisms such as triggers [33], bounded unfolding of recursive functions, or natural proofs [30,39]) to reduce this logical task to a *decidable* theory $\mathcal{D}$. This reduction is assumed to be sound in the sense that if the resulting formulas in $\mathcal{D}$ are valid, then the verification conditions are valid in $\mathcal{U}$ as well. If a formula is found *not valid* in $\mathcal{D}$, then we require that the logic solver for $\mathcal{D}$ 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 $\mathcal{U}$.

Moreover, we fix a hypothesis class $\mathcal{H}$ for invariants consisting of *positive* Boolean combination of predicates in a fixed set of predicates $\mathcal{P}$. Note that restricting to *positive* formulas over $\mathcal{P}$ is not a restriction, as one can always add negations of predicates to $\mathcal{P}$, thus effectively synthesizing any Boolean combination of predicates. The restriction to positive Boolean formulas is in fact desirable, as it allows restricting invariants to *not* negate certain predicates,

[1] Note that our framework requires model construction in the theory $\mathcal{D}$. Hence, incomplete logic solvers for $\mathcal{U}$ 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.

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 $\mathcal{H}$. The verification engine generates verification conditions in accordance to these invariants in the underlying theory $\mathcal{U}$. It then proceeds to translate them into the decidable theory $\mathcal{D}$, and gives them to a solver that decides validity in the theory $\mathcal{D}$. If the verification conditions are found to be $\mathcal{D}$-valid, then by virtue of the fact that the verification engine reduced VCs in a sound fashion to $\mathcal{D}$, we are done proving the program P.

However, if the formula is found not to be $\mathcal{D}$-valid, the solver returns a $\mathcal{D}$-model for the negation of the formula. The verification engine then extracts from this model certain *non-provability information (NPI)*, expressed as Boolean formulas in a Boolean theory $\mathcal{B}$, that 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* $\vdash_{\mathcal{B}} \delta \Rightarrow \gamma$*, then it cannot prove the validity of the Hoare triple* $\{\alpha\}s\{\delta\}$*; and*
2. *if the engine cannot prove the validity of the Hoare triple* $\{\gamma\}s\{\beta\}$ *and* $\vdash_{\mathcal{B}} \gamma \Rightarrow \delta$*, then it cannot prove the validity of the Hoare triple* $\{\delta\}s\{\beta\}$*.*

The remainder of this section is now structured as follows. In Sect. 2.1, we first 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 $\mathcal{P}$, 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 $\mathcal{D}$-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. Section 2.3 argues the soundness of our framework and guarantees of convergence.

2.1 Conjunctive/Disjunctive Non-provability Information

We assume that the underlying decidable theory $\mathcal{D}$ is stronger than propositional theory $\mathcal{B}$, meaning that every valid statement in $\mathcal{B}$ is valid in $\mathcal{D}$ as well.

The reader may want to keep the following as a running example where $\mathcal{D}$ is the decidable theory of uninterpreted functions and linear arithmetic, say. In this setting, a formula is $\mathcal{B}$-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 $\mathcal{B}$-valid though it is $\mathcal{D}$-valid, while $f(x) = y \vee \neg(f(x) = y)$ is $\mathcal{B}$-valid.

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

Let us now assume, for the sake of a simpler exposition, that the program has a single annotation hole A where we need to synthesize an inductive invariant and prove the program correct. Further, suppose 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 $\mathit{approx}(VC(\{\alpha\}s\{\gamma\}))$ is $\mathcal{D}$-satisfiable and the verification engine needs to extract non-provability information from a model of it. To this end, we assume that every program snippet s has been augmented with a set of ghost variables $g_1, \ldots, g_n$ that track the predicates $p_1, \ldots, p_n$ mentioned in the invariant (i.e., these ghost variables are assigned the values of the predicates). The valuation $\boldsymbol{v} = \langle v_1, \ldots, v_n \rangle$ of the ghost variables in the model before the execution of s and the valuation $\boldsymbol{v}' = \langle v'_1, \ldots, v'_n \rangle$ after the execution of s can then be used to derive non-provability information, as we describe shortly.

The type of non-provability information the verification engine extracts depends on where the annotation appears in a Hoare triple $\{\alpha\}s\{\beta\}$. More specifically, the synthesized annotation might appear 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, $\mathit{approx}(VC(\{\alpha\}s\{\gamma\}))$ is not $\mathcal{D}$-valid, and the decision procedure for $\mathcal{D}$ would generate a model for its negation.
  Since γ is a positive Boolean combination, the reason why $\boldsymbol{v}'$ does not satisfy γ is due to the variables mapped to *false* by $\boldsymbol{v}'$, as any valuation extending this will not satisfy γ. Intuitively, this means that the $\mathcal{D}$-solver is not able to prove the predicates in $P_{\mathit{false}} = \{p_i \mid v'_i = \mathit{false}\}$. In other words, $\{\alpha\}s\{\bigvee P_{\mathit{false}}\}$ is unprovable (a witness to this fact is the model of the negation of $\mathit{approx}(VC(\{\alpha\}s\{\gamma\}))$ from which the values $\boldsymbol{v}'$ are derived). Note that any invariant γ' that is stronger than $\bigvee P_{\mathit{false}}$ will result in an unprovable VC due to the verification engine being normal. Consequently we can choose $\chi = \bigvee P_{\mathit{false}}$ 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 $\not\models_{\mathcal{B}} \gamma'' \Rightarrow \chi$).

- 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* $\eta = \bigwedge\{p_i \mid v_i = \mathit{true}\}$ of the predicates mapped to *true* by $\boldsymbol{v}$ in the corresponding $\mathcal{D}$-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 $\not\models_{\mathcal{B}} \eta \Rightarrow \gamma''$).
- 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 $\boldsymbol{v}$ and to *false* by $\boldsymbol{v}'$. The meaning of such a constraint is that the invariant synthesizer must conjecture in future rounds invariants γ'' such that either $\not\models_{\mathcal{B}} \eta \Rightarrow \gamma''$ or $\not\models_{\mathcal{B}} \gamma'' \Rightarrow \chi$ 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 $\mathcal{B}$-formula $\bigvee_{i \mid v'_i = \mathit{false}} p_i$ as a weakening constraint.
- When a Hoare triple of the form $\{\gamma\}s\{\beta\}$ fails, the verification engine returns the $\mathcal{B}$-formula $\bigwedge_{i \mid v_i = \mathit{true}} p_i$ as a strengthening constraint.
- When a Hoare triple of the form $\{\gamma\}s\{\gamma\}$ fails, the verification engine returns the pair $(\bigwedge_{i \mid v_i = \mathit{true}} p_i, \bigvee_{i \mid v'_i = \mathit{false}} p_i)$ of $\mathcal{B}$-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 $\mathcal{P}$ 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 $\mathcal{P}$ (weakening constraints);*
- *a finite set S of conjunctions over $\mathcal{P}$ (strengthening constraints); and*
- *a finite set I of pairs, where the first element is a conjunction and the second is a disjunction over $\mathcal{P}$ (inductivity constraints).*

An annotation γ *is* consistent *with a CD-NPI sample* $\mathfrak{S} = (W, S, I)$ *if* $\not\vdash_{\mathcal{B}} \gamma \Rightarrow \chi$ *for each* $\chi \in W$, $\not\vdash_{\mathcal{B}} \eta \Rightarrow \gamma$ *for each* $\eta \in S$, *and* $\not\vdash_{\mathcal{B}} \eta \Rightarrow \gamma$ *or* $\not\vdash_{\mathcal{B}} \gamma \Rightarrow \chi$ *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 below.

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 $\mathcal{P}$ is finite.

Roughly speaking, the CD-NPI learning problem is to synthesize annotations that are positive Boolean combinations of predicates in $\mathcal{P}$ and that are consistent with given CD-NPI samples. Though this is a learning problem where samples are *formulas*, in this section we will reduce CD-NPI learning to a learning problem from *data*. In particular, we will 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 [17] is a general framework for learning inductive invariants, we consider here the case of learning Boolean formulas. To this end, let us fix a set B of Boolean variables, and let $\mathcal{H}$ be a subclass of positive Boolean formulas over B, called the hypothesis class, which specifies the admissible solutions to the learning task.

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

A formula φ is said to be *consistent with an ICE sample* $\mathcal{S}$ if it satisfies the following three conditions:[2] $v \models \varphi$ 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 $(v_1, v_2) \in S_\Rightarrow$.

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

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 $\mathcal{H}$ is then *"given an ICE sample $\mathcal{S}$, find a formula in $\mathcal{H}$ that is consistent with $\mathcal{S}$"*. Recall that we here require the learner to learn positive Boolean formulas, which is slightly stricter than the original definition [17].

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.

Reduction of Passive CD-NPI Learning to Passive ICE Learning. Let $\mathcal{H}$ be a subclass of positive Boolean formulas. We reduce the CD-NPI learning problem for $\mathcal{H}$ to the ICE learning problem for $\mathcal{H}$. The main idea is to (a) treat each predicate $p \in \mathcal{P}$ 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 $\mathcal{S}_{\mathfrak{S}}$, meaning that a positive Boolean formula is consistent with $\mathfrak{S}$ if and only if it is consistent with $\mathcal{S}_{\mathfrak{S}}$. Then, learning a consistent formula in the CD-NPI framework for the hypothesis class $\mathcal{H}$ reduces to learning consistent formulas in $\mathcal{H}$ in the ICE learning framework.

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

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

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

This motivates our translation, which relies on two functions, d and c. The function d translates a disjunction $\bigvee J$, where $J \subseteq \mathcal{P}$ is a subset of propositions, into the valuation $d(\bigvee J) = v$ with $v(p) = false$ if and only if $p \in J$. The function c translates a conjunction $\bigwedge J$, where $J \subseteq \mathcal{P}$, into the valuation $c(\bigwedge J) = v$ with $v(p) = true$ if and only if $p \in J$. By substituting v in Lemma 1 with $c(\bigwedge J)$ and $d(\bigvee J)$, respectively, one immediately obtains the following.

Lemma 2. *Let $J \subseteq \mathcal{P}$ and α be a positive Boolean formula over $\mathcal{P}$. Then, the following holds:(a)* $c(\bigwedge J) \not\models \alpha$ *if and only if* $\not\vdash_{\mathcal{B}} \bigwedge J \Rightarrow \alpha$*, and (b)* $d(\bigvee J) \models \alpha$ *if and only if* $\not\vdash_{\mathcal{B}} \alpha \Rightarrow \bigvee J$*.*

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* $\mathcal{S}_{\mathfrak{S}} = (S_+, S_-, S_\Rightarrow)$ *is defined by* $S_+ = \{d(\bigvee J) \mid \bigvee J \in W\}$*,* $S_- = \{c(\bigwedge J) \mid \bigwedge J \in S\}$*, and* $S_\Rightarrow = \{(c(\bigwedge J_1), d(\bigvee J_2)) \mid (\bigwedge J_1, \bigvee J_2) \in I\}$*.*

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 (a proof can be found in our extended paper [35]).

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

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 mentioned earlier, we can add the negations of predicates as first-class predicates, and hence synthesize invariants over the class of all Boolean combinations of a finite set of predicates 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 $\mathcal{P}$ of candidate predicates is large (hundreds in our experiments), since the number of Boolean formulas over $\mathcal{P}$ is doubly exponential in $n = |\mathcal{P}|$, building an effective learner is not easy. However, there is one class of formulas that are particularly amenable to efficient ICE learning—learning *conjunctions of predicates over* $\mathcal{P}$. In this case, there are ICE learning algorithms that promise learning the invariant (provided one exists expressible as a conjunct over $\mathcal{P}$) in $n + 1$ rounds. Note that this learning is essentially finding an invariant in a hypothesis class $\mathcal{H}$ of size 2^n in $n+1$ rounds.

Houdini [14] 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. [17]. In fact, Houdini is similar to the classical PAC learning algorithm for conjunctions [26], but extended to the ICE model. The time Houdini spends in each round is *polynomial* and in an iterative setting, is guaranteed to converge in at most $n+1$ rounds or report that no conjunctive invariant over $\mathcal{P}$ exists. We use this ICE learner to build a CD-NPI learner for conjunctions.

2.3 Main Result

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

Theorem 2. *Assume a normal verification engine for a program P to be given. Moreover, let $\mathcal{P}$ be a finite set of predicates over the variables in P and $\mathcal{H}$ a hypothesis class consisting of positive Boolean combinations of predicates in $\mathcal{P}$. If there exists an annotation in $\mathcal{H}$ 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.*

The proof of Theorem 2 can be found in our extended paper [35]. 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 $\mathcal{H}$. 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.

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 [39,41].

Background: Natural Proofs and Dryad Dryad [39,41] 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 [39]). Dryad is expressive enough to state a variety of data-structures (singly and doubly linked lists, sorted lists, binary search trees, AVL trees, max-heaps, treaps), recursive definitions over them that map to numbers (length, height, etc.), as well as data stored within the heap (the multiset of keys stored in lists, trees, etc.).

The technique of using natural proofs [39,41] 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 that demand that the accesses of each method are contained in the heaplet implicitly defined by its precondition (taking into account newly allocated or freed nodes), and that at the end of the program, the modified heaplet precisely matches the implicit heaplet defined by the post-condition.

The second step the natural proof verifier does is to perform *transformations* on the program and translate it to Boogie [14], an intermediate verification language that handles proof obligations using automatic theorem provers (typically SMT solvers). VCDryad extends VCC [8] to perform several natural proof transformations on heap-manipulating C programs that essentially perform three

tasks: (a) abstract all recursive definitions on the heap using uninterpreted functions but introduce finite-depth unfoldings of recursive definitions at every place in the code where locations are dereferenced, (b) model heaplets and other sets using a decidable theory of maps, (c) insert *frame reasoning* explicitly in the code that allows the verifier to derive that certain properties continue to hold across a heap update (or function call) using the heaplet that is modified.

The resulting program is a Boogie program with no recursive definitions, where all verification conditions are in decidable logics, and where the logic engine can return models when formulas are satisfiable. The program can be verified if supplied with correct inductive loop-invariants and adequate pre/post conditions. We refer the reader to [39,41] for more details.

Learning Heap Invariants. We have implemented a prototype[3] 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 an invariant synthesis engine that learns invariants over the space of conjuncts over a finite set of predicates; we describe below 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 generate a set $\mathcal{P}$ of *predicates*, which serve as the basic building blocks of our invariants. The predicates are generated from generic templates, which are instantiated using all combinations of program variables that occur in the program being verified. We refer the reader to our extended paper [35] for a full description.

The templates define a fairly exhaustive set of predicates. These predicates include 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, BST, AVL, treaps, etc.), and recursive definitions that map data structures to numbers (keys/data stored in a structure, lengths of lists and list segments, height of trees) involving arithmetic relationships and set relationships. In addition, there are also predicates describing heaplets of various structures, involving set operations, disjointness, and equalities. The structures and predicates are extensible, of course.

The predicates are grouped into three categories, roughly in increasing complexity. Category 1 predicates involve shape-related properties, Category 2 involves properties related to the keys stored in the data-structure, and Category 3 predicates involve size-predicates on datastructures (lengths of lists and heights of trees). Given a program to verify and its annotations, we choose the category of predicates 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

[3] This prototype as well as the benchmarks used to reproduce the results presented below are publicly available on figshare [36].

generated by instantiating the templates with all (combinations of) program variables; this allows us to control the size of the set of predicates used.

Evaluation. We have evaluated our prototype on ten benchmark suits (82 routines in total) 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, (2) GNU C Library(glibc) doubly linked lists, (3) OpenBSD SysQueue, (4) GRASSHOPPER [40] singly linked lists, (5) GRASSHOPPER [40] doubly linked lists, (6) GRASSHOPPER [40] sorted linked lists, (7) VCDRYAD [39] sorted linked lists, (8) VCDRYAD [39] binary search trees, AVL trees, and treaps, (9) AFWP [23] singly/sorted linked lists, and (10) ExpressOS [31] 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, and sortedness of elements. The specifications hence involve separation logic with arithmetic as well as recursive definitions that compute numbers (like lengths and heights), data-aggregating recursive functions (such as multisets of keys stored in 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, *erased* the user-provided loop invariants, and used our framework to synthesize an adequate inductive invariant. We also selected some programs that were purely recursive, where the contract for the function had 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 also chose five straight-line programs, deleted their post-conditions, and evaluated whether we can 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.

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

Experimental Results. We performed all experiments in a virtual machine running Ubuntu 16.04.1 on a single core of an Intel Core i7-7820 HK 2.9 GHz CPU with 2 GB memory. The box plots in Fig. 2 summarize the results of this empirical evaluation aggregated by benchmark suite, specifically the time required to verify the programs, the number of automatically inserted base predicates, and the number of iterations in the learning process (see our extended paper [35] for full details). Each box in the diagrams shows the lower and upper

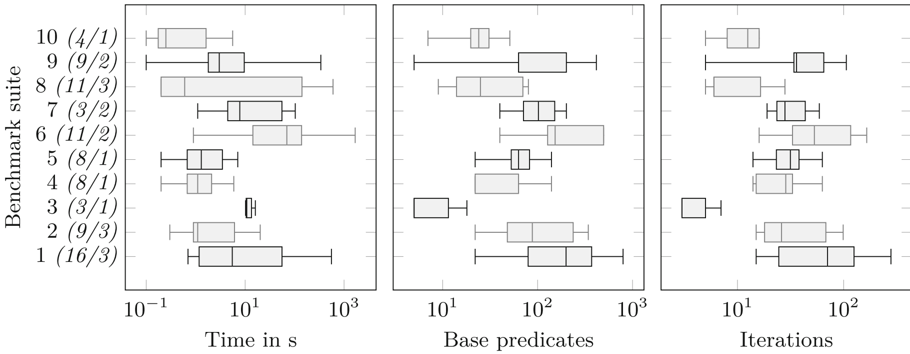


Fig. 2. Experimental evaluation of our prototype. The numbers in italic brackets shows the total number or programs in the suite (first number) and the maximum predicate category used (second number).

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 82 benchmarks. The fact that the median time for a great majority of benchmark suits is less than 10 s shows that our technique is extremely effective in finding inductive Dryad invariants. We observe that despite many examples having hundreds of base predicates, which suggests a worst-case complexity of hundreds of iterations, the learner was able to learn with much fewer iterations and the number of predicates in the final invariant is small. This shows that non-provability information provides much more information than the worst-case suggests.

To the best of our knowledge, our prototype is the only tool currently able of fully automatically verifying this challenging benchmark set. We must emphasize, however, that there are subsets of our benchmarks that can be solved by reformulating verification in decidable fragments of separation logic studied—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 whole benchmark set.

4 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 implemented a prototype, based on the natural proofs verification engine, and demonstrated that this prototype can verify a large set of heap-manipulating programs against specifications expressed in an expressive and undecidable dialect of separation logic. We are not aware of any other technique that can handle this extremely challenging benchmark suite.

Several future research directions are interesting. First, the framework we have developed is based on CEGIS 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 $\mathcal{D}$ (or $\mathcal{B}$) abstractions of the program directly in order to synthesize invariants for them. Second, in the NPI learning framework we have put forth, it would be interesting to change the underlying logic of communication $\mathcal{B}$ 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 $\mathcal{B}$. 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.

Data Availability Statement and Acknowledgments. The prototype developed in this project as well as the benchmarks used to produce the results presented in this work are available in the figshare repository at https://doi.org/10.6084/m9.figshare.5928094.v1.

This material is based upon work supported by the National Science Foundation under Grants #1138994, and #1527395.

References

1. Albargouthi, A., Berdine, J., Cook, B., Kincaid, Z.: Spatial interpolants. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 634–660. Springer, Heidelberg (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: PLDI 2001, pp. 203–213 (2001)
3. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
4. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
5. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26 (2011)
6. Chin, W.N., 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)
7. Chu, D., Jaffar, J., Trinh, M.: Automatic induction proofs of data-structures in imperative programs. In: PLDI 2015, pp. 457–466. ACM (2015)
8. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_2

9. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_39
10. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, pp. 238–252. ACM Press (1977)
11. Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: OOPSLA 2013, pp. 443–456 (2013)
12. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: FMCAD 2011, pp. 125–134. FMCAD Inc (2011)
13. Ernst, M.D., Czeisler, A., Griswold, W.G., Notkin, D.: Quickly detecting relevant program invariants. In: ICSE 2000, pp. 449–458. ACM Press (2000)
14. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45251-6_29
15. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Proceedings of a Symposium on Applied Mathematics. Mathematical Aspects of Computer Science, vol. 19, pp. 19–31. American Mathematical Society (1967)
16. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 813–829. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_57
17. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: a robust framework for learning invariants. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 69–87. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_5
18. Garg, P., Madhusudan, P., Neider, D., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: POPL 2016, pp. 499–512 (2016)
19. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25
20. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI 2008, pp. 281–292. ACM (2008)
21. Gupta, A., Rybalchenko, A.: InvGen: an efficient invariant generator. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 634–640. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_48
22. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
23. Itzhaky, S., Banerjee, A., Immerman, N., Nanevski, A., Sagiv, M.: Effectively-propositional reasoning about reachability in linked data structures. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 756–772. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_53
24. Itzhaky, S., Bjørner, N., Reps, T., Sagiv, M., Thakur, A.: Property-directed shape analysis. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 35–51. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_3
25. Karbyshev, A., Bjørner, N., Itzhaky, S., Rinetzky, N., Shoham, S.: Property-directed inference of universal invariants or proving their absence. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 583–602. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_40

26. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
27. Krishna, S., Puhrsch, C., Wies, T.: Learning invariants using decision trees. CoRR abs/1501.04725 (2015)
28. Le, Q.L., Gherghina, C., Qin, S., Chin, W.-N.: Shape analysis via second-order bi-abduction. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 52–68. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_4
29. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
30. Löding, C., Madhusudan, P., Peña, L.: Foundations for natural proofs and quantifier instantiation. PACMPL **2**(POPL), 10:1–10:30 (2018). http://doi.acm.org/10.1145/3158098
31. Mai, H., Pek, E., Xue, H., King, S.T., Madhusudan, P.: Verifying security invariants in expressos. In: ASPLOS 2013, pp. 293–304. ACM (2013)
32. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_1
33. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_13
34. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
35. Neider, D., Garg, P., Madhusudan, P., Saha, S., Park, D.: Invariant synthesis for incomplete verification engines. CoRR abs/1712.05581 (2017). https://arxiv.org/abs/1712.05581
36. Neider, D., Garg, P., Madhusudan, P., Saha, S., Park, D.: Prototype and benchmarks for "invariant synthesis for incomplete verification engines", February 2018. https://doi.org/10.6084/m9.figshare.5928094.v1
37. Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: PLDI 2016, pp. 42–56 (2016)
38. Pavlinovic, Z., Lal, A., Sharma, R.: Inferring annotations for device drivers from verification histories. In: ASE 2016, pp. 450–460 (2016)
39. Pek, E., Qiu, X., Madhusudan, P.: Natural proofs for data structure manipulation in C using separation logic. In: PLDI 2014, p. 46. ACM (2014)
40. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 773–789. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_54
41. Qiu, X., Garg, P., Stefanescu, A., Madhusudan, P.: Natural proofs for structure, data, and separation. In: PLDI 2013, pp. 231–242. ACM (2013)
42. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL 1999, pp. 105–118. ACM (1999)
43. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 88–105. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_6
44. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 574–592. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_31

45. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Nori, A.V.: Verification as learning geometric concepts. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 388–411. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_21
46. Sharma, R., Nori, A.V., Aiken, A.: Interpolants as classifiers. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 71–87. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_11
47. Zhu, H., Nori, A.V., Jagannathan, S.: Learning refinement types. In: ICFP 2015, pp. 400–411. ACM (2015)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.