SORCAR: Property-Driven Algorithms for Learning Conjunctive Invariants

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

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

Abstract. We present a new learning algorithm SORCAR to synthesize conjunctive inductive invariants for proving that a program satisfies its assertions. The salient property of this algorithm is that it is *property-driven*, and for a fixed finite set of n predicates, guarantees convergence in $2n$ rounds, taking only polynomial time in each round. We implement and evaluate the algorithm and show that its performance is favorable to the existing HOUDINI algorithm (which is not property-driven) for a class of benchmarks that prove data race freedom of GPU programs and another class that synthesizes invariants for proving separation logic properties for heap manipulating programs.

Keywords: Invariant synthesis · Machine learning · Horn-ICE learning · Conjunctive formulas

1 Introduction

The deductive verification approach for proving imperative programs correct is one of the most well-established and effective methods, and automating program verification using this method has been studied extensively. This approach can be seen as consisting of two parts: (a) writing inductive invariants in terms of loop invariants, class invariants, and method contracts, and (b) proving that these annotations are indeed correct using theorem proving. Automation of the latter has seen tremendous progress in the last two decades through the identification of decidable logical theories, theory combinations, heuristics for automatically reasoning with quantified theories, and their realization using efficient SMT solvers [5,34]. There has also been significant progress on automating the former problem of discovering inductive invariants [3,7,8,12–20,23,24,27,32,40–43,48], with varying degrees of success.

In this paper, we are interested in a class of or *learning-based* techniques for invariant generation [8,16,20,48]. In this context, the invariant synthesis engine is split into two components, a learner and a teacher, who work in rounds. In each round, the teacher examines the invariant produced by the learner and produces counterexamples that consist of concrete program configurations that

B.-Y. E. Chang (Ed.): SAS 2019, LNCS 11822, pp. 323–346, 2019.
https://doi.org/10.1007/978-3-030-32304-2_16

show why the proposed formulas are not inductive invariants. The learner then uses these concrete program configurations to synthesize new proposals for the invariant, *without* looking at the program. The teacher, on the other hand, does look at the program and produces counterexamples based on failed verification attempts.

The choice to separate the learner and teacher—and not give the learner access to the program—may seem strange at first. However, a rationale for this choice has emerged over the years, and the above choice is in fact the de facto approach for synthesis in various other domains, including program synthesis, where it is usually called counter-example guided inductive synthesis [1,44,45].

Horn-ICE Learning. In a paper at CAV 2014, Garg et al. [20] studied the above learning model and identified the precise form of counterexamples needed for synthesizing invariants. Contrary to usual classification learning where one is given positive and negative examples only, the authors argued that implication counterexamples (ICE) are needed, and coined the term *ICE learning* for such a learning model. More recently, it has been recognized that program verification problems can be cast as solving *Horn implication constraints* [22]. Consequently, the implication counterexamples returned by the teacher are naturally Horn implications (*Horn-ICE*), involving concrete program configurations. New algorithms for learning from such Horn counterexamples have recently been studied [8,16].

Learning Conjunctions Over a Fixed Set of Predicates. While one can potentially learn/synthesize invariants in complex logics, one technique that has been particularly effective and scalable is to fix a finite set of predicates $\mathcal{P}$ over the program configurations and only learn inductive invariants that can be expressed as a conjunction of predicates over $\mathcal{P}$. For particular domains of programs and types of specifications, it is possible to identify classes of candidate predicates that are typically involved in invariants (e.g., based on the code of the programs and/or the specification), and learning invariants over such a class of predicates has proven very effective. A prominent example is device drivers, and Microsoft's Static Driver Verifier [28,33] (specifically the underlying tool Corral [29]) is an industry-strength tool that leverages exactly this approach.

In this paper, we are mainly motivated by two other domains where learning conjunctive invariants is very effective. The first is the class of programs handled by GPUVerify [6,9], which considers GPU programs, reduces the problem to a sequential verification problem (by simulating two threads at each parallel fork), and proceeds to find conjunctive invariants over a fixed set $\mathcal{P}$ of predicates to prove the resulting sequential program correct. The second class is the class of programs considered by Neider et al. [35], where the authors synthesize invariants in order to prove the correctness of programs that dynamically update heaps against specifications in separation logic. The verification engine in the former is an SMT solver that returns concrete Horn-ICE counterexamples. In the latter, predicates involve *inductively defined relations* (such as a list-segment, the

Table 1. Comparison of HOUDINI [18] and SORCAR

Learning algorithm	**Property driven?**	**Complexity per round**	**Maximum # rounds**	**Final conjunct**
HOUDINI	No	Polynomial	$\lvert\mathcal{P}\rvert$	Largest set
SORCAR	Yes	Polynomial	$2 \cdot \lvert\mathcal{P}\rvert$	Bias towards weaker invariants (smaller sets of conjunctions) involving only relevant predicates

heaplet associated with it, or the set of keys stored in it), and validating verification conditions is undecidable in general. Hence, the verification engine is a sound but incomplete verification engine (based on "natural proofs") that returns abstract counterexamples that can be interpreted to be Horn-ICE counterexamples. In both domains, the set $\mathcal{P}$ consists of hundreds of candidate predicates, which makes invariant synthesis challenging (as there are $2^{|\mathcal{P}|}$ possible conjunctive invariants).

HOUDINI and SORCAR. The classical algorithm for learning conjunctive invariants over a finite class of predicates is the HOUDINI algorithm [18], which mimics the *elimination algorithm* for learning conjuncts in classical machine learning [26]. HOUDINI starts with a conjectured invariant that contains *all* predicates in $\mathcal{P}$ and, in each round, uses information from a failed verification attempt to remove predicates. The most salient aspect of the algorithm is that it is guaranteed to converge to a conjunctive invariant, if one exists, in $n = |\mathcal{P}|$ rounds (which is logarithmic in the number of invariants, as there are 2^n of them). However, the HOUDINI algorithm has disadvantages as well. Most notably, it is not property-driven as it does not consider the assertions that occur in the program (which is a consequence of the fact that it was originally designed to infer invariants of unannotated programs). In fact, one can view the HOUDINI algorithm as a way of computing the least fixed point in the abstract interpretation framework, where the abstract domain consists of conjunctions over the candidate predicates.

In this paper, we develop a new class of learning algorithms for conjunctions, named SORCAR[1], *that is property-driven.*

The primary motivation to build a property-driven learning algorithm is to explore invariant generation techniques that can be potentially more efficient in proving programs correct. The SORCAR algorithm presented in this paper has the following design features (also see Table 1). First, it is property-driven—in other words, the algorithm tries to find conjunctive inductive invariants that are sufficient to prove the assertions in the program. By contrast, HOUDINI computes the *tightest* inductive invariant. Since SORCAR is property-driven, it can find weaker inductive invariants (i.e., invariants with fewer conjuncts). Our intuition is that by synthesizing weaker, property-driven invariants, we can verify programs more efficiently.

[1] Houdini and Sorcar were both magicians!

Second, Sorcar guarantees that the number of rounds of interaction with the teacher is still linear ($2n$ rounds compared to Houdini's promise of n rounds). Third, Sorcar promises to do only polynomial amount of work in each round (i.e., polynomial in n and in the number of current counterexamples), similar to Houdini.

The Sorcar algorithm works, intuitively, by finding conjunctive invariants over a set of *relevant predicates* $R \subseteq \mathcal{P}$. This set is grown slowly (but monotonically, as monotonic growth is crucial to ensure that the number of rounds of learning is linear) by adding predicates only when they were found to be relevant to prove assertions. More specifically, predicates are considered relevant based on information gained from counterexamples of failed verification conditions that involve assertions in the program. The precise mechanism of growing the set of relevant predicates can vary, and we define four variants of Sorcar (e.g., choosing all predicates that show promise of relevance or greedily choosing a minimal number of relevant predicates). The Sorcar suite of algorithms is hence a new class of property-driven learning algorithms for conjunctive invariants with different design principles.

Experimental Evaluation. We have implemented Sorcar as a Horn-ICE learning algorithm on top of the Boogie program verifier [4] and have applied it to verify both GPU programs for data races [6,9] and heap manipulating programs against separation logic specifications [35]. To assess the performance of Sorcar, we have compared it to the current state-of-the-art tools for these programs, which use the Houdini algorithm. Though Sorcar did not work more efficiently on every program, our empirical evaluation shows that it is overall more competitive than Houdini. In summary, we found that (a) Sorcar worked more efficiently overall in verifying these programs, and (b) Sorcar verified a larger number of programs than Houdini did (for a suitably large timeout).

Related Work

Invariant synthesis lies at the heart of automated program verification. Over the years, various techniques have been proposed, including abstract interpretation [13], interpolation [32], IC3 [7], predicate abstraction [3], abductive inference [14], as well as synthesis algorithms that rely on constraint solving [12,17,23,24]. Complementing these techniques are data-driven approaches that are based on machine learning. Examples include Daikon [15] and Houdini [18], the ICE learning framework [20] and its successor Horn-ICE learning [8,16], as well as numerous other techniques that employ machine learning to synthesize inductive invariants [19,27,40–43,48].

One potentially interesting question is whether ICE/Horn-ICE algorithms (and in particular, Houdini and Sorcar) are qualitatively related to algorithms such as IC3 for synthesizing invariants. For programs with Boolean domains, Vizel et al. [47] study this question and find that the algorithms are quite different. In fact, the authors propose a new framework that generalizes both. In the setting of this paper, however, there are too many differences to reconcile with: (a) IC3 finds invariants by bounded symbolic exploration, forward from initial configurations and backward from bad configurations (hence inherently unfolding loops), while ICE/Horn-ICE algorithms do not do that, (b) ICE/Horn-ICE algorithms instead use implication/Horn counterexamples, which can relate configurations arbitrarily far away from initial or bad configurations, and there seems to be no analog to this in IC3, (c) it is not clear how to restrict IC3 to finding invariants in a particular hypothesis class, such as conjunctions over a particular set of predicates, (d) IC3 works very closely with a SAT solver, whereas ICE/Horn-ICE algorithms are essentially independent, communicating with the SAT/SMT engine only indirectly, and (e) we are not aware of any guarantees that IC3 can give in terms of the number of rounds/conjectures, whereas the ICE/Horn-ICE algorithms Houdini and Sorcar give guarantees that are linear in the number of predicates. We believe that the algorithms are in fact very different, though more general algorithms that unify them would be interesting to study.

Learning of conjunctive formulas has a long history. An early example is the so-called elimination algorithm [26], which operates in the Probably Approximately Correct Learning model (PAC). Daikon [15] was the first technique to apply the elimination algorithm in a software setting, learning likely invariants from dynamic traces. Later, the popular Houdini [18] algorithm built on top of the elimination algorithm to compute inductive invariants in a fully automated manner. In fact, as Garg et al. [21] and later Ezudheen et al. [16] argued, Houdini can be seen as a learning algorithm for conjunctive formulas in both the ICE and the Horn-ICE learning framework.

Using Houdini to compute conjunctive invariants over a finite set of candidate predicates is extremely scalable and has been used with great success in several practical settings. For example, Corral [29], which uses Houdini internally, has replaced Slam [2] and Yogi [36], and is currently shipped as part of Microsoft's industrial-strength Static Driver Verifier (SDV) [28,33]. GPUVerify [6,9] is another example that uses Houdini with great success to prove race freedom of GPU programs.

2 Background

In this section, we provide the background on learning-based invariant synthesis. In particular, we briefly recapitulate the Horn-ICE learning framework (in Sect. 2.1) and discuss the Houdini algorithm (in Sect. 2.2), specifically in the context of the Horn-ICE framework.

To make the Horn-ICE framework mathematically precise, let P be the program (with assertions) under consideration and $\mathcal{C}$ the set of all program configurations of P. Furthermore, let us fix a finite set $\mathcal{P}$ of *predicates* $p \colon \mathcal{C} \to \mathbb{B}$ over the program configurations, where $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$ is the set of Boolean values. These predicates capture interesting properties of the program and serve as the basic building blocks for constructing invariants. We assume that the values of these predicates can either be obtained directly from the program configurations or that the program is instrumented with ghost variables that track the values of the predicates at important places in the program (e.g., at the loop header and immediately after the loop). As notational convention, we write $c \models p$ if $p(c) = \mathit{true}$ and $c \not\models p$ if $p(c) = \mathit{false}$. Moreover, we lift this notation to formulas φ over $\mathcal{P}$ (i.e., arbitrary Boolean combinations of predicates from $\mathcal{P}$) and use $c \models \varphi$ ($c \not\models \varphi$) to denote that c satisfies φ (c does not satisfy φ).

To simplify the presentation in the remainder of this paper, we use conjunctions $p_1 \wedge \cdots \wedge p_n$ of predicates over $\mathcal{P}$ and the corresponding sets $\{p_1, \ldots, p_n\} \subseteq \mathcal{P}$ interchangeably. In particular, for a (sub-)set $X = \{p_1, \ldots, p_n\} \subseteq \mathcal{P}$ of predicates and a program configuration $c \in \mathcal{C}$, we write $c \models X$ if and only if $c \models p_1 \wedge \cdots \wedge p_n$.

2.1 The Horn-ICE Learning Framework

The Horn-ICE learning framework [8,16] is a general framework for learning inductive invariants in a black-box setting. We here assume without loss of generality that the task is to synthesize a single invariant. In the case of learning multiple invariants, say at different program locations, one can easily expand the given predicates to predicates of the form $(pc = l) \rightarrow p$ where pc refers to the program counter, l is the location of an invariant in the program, and $p \in \mathcal{P}$. Learning a conjunctive invariant over this extended set of predicates then corresponds to learning multiple conjunctive invariants at the various locations.

As sketched in Fig. 1, the Horn-ICE framework consists of two distinct entities—the *learner* and the *teacher*—and proceeds in rounds. In each round, the teacher receives a candidate invariant φ from the learner and checks whether φ proves the program correct. Should φ not be adequate to prove the program correct, the learner replies with a counterexample, which serves as a means to correct inadequate invariants and guide the learner towards a correct one. More precisely, a counterexample takes one of three forms:[2]

– If the pre-condition α of the program does not imply φ, then the teacher returns a *positive counterexample* $c \in \mathcal{C}$ such that $c \models \alpha$ but $c \not\models \varphi$.
– If φ does not imply the post-condition β of the program, then the teacher returns a *negative counterexample* $c \in \mathcal{C}$ such that $c \models \varphi$ but $c \not\models \beta$.
– If φ is not inductive, then the teacher returns a *Horn counterexample* $(\{c_1, \ldots, c_n\}, c) \in 2^{\mathcal{C}} \times \mathcal{C}$ such that $c_i \models \varphi$ for each $i \in \{1, \ldots, n\}$ but $c \not\models \varphi$. (We encourage the reader to think of Horn counterexamples as constraints of the form $(c_1 \wedge \cdots \wedge c_n) \rightarrow c$.)

A teacher who returns counterexamples as described above always enables the learner to make *progress* in the sense that every counterexample it returns is inconsistent with the current conjecture (i.e., it violates the current conjecture). Moreover, the Horn-ICE framework requires the teacher to be *honest*, meaning that each counterexample needs to be consistent with *all* inductive

[2] By abuse of notation, we write $c \models \alpha$ ($c \not\models \alpha$) to denote that c satisfies (violates) the formula α even if α contains predicates that do not belong to $\mathcal{P}$.

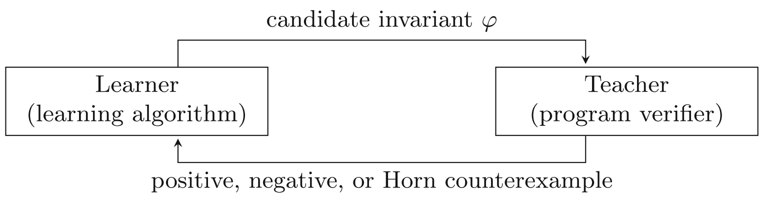
candidate invariant φ
Learner
(learning algorithm)
Teacher
(program verifier)
positive, negative, or Horn counterexample

Fig. 1. The Horn-ICE learning framework [8,16]

invariants that prove the program correct (i.e., the teacher does not rule out possible solutions). Finally, note that such a teacher can indeed be built since program verification can be stated by means of *constrained Horn clauses* [22]. When the candidate invariant does not make such clauses true, some Horn clause failed, and the teacher can find a Horn counterexample using a logic solver (positive counterexamples arise when the left-hand-side of the Horn counterexample is empty, while negative counterexamples arise when the left-hand-side has one element and the-right-hand side is *false*).

The objective of the learner, on the other hand, is to construct a formula φ over $\mathcal{P}$ from the counterexamples received thus far. For the sake of simplicity, we assume that the learner collects all counterexamples in a data structure $\mathcal{S} = (S_+, S_-, S_H)$, called *Horn-ICE sample*, where

1. $S_+ \subseteq \mathcal{C}$ is a finite set of positive counterexamples;
2. $S_- \subseteq \mathcal{C}$ is a finite set of negative counterexamples; and
3. $S_H \subseteq 2^{\mathcal{C}} \times \mathcal{C}$ is a finite set of Horn counterexamples.

To measure the complexity of a sample, we define its *size*, denoted by $|\mathcal{S}|$, to be $|S_+| + |S_-| + \sum_{(L,c) \in S_H} (|L| + 1)$.

Given a Horn-ICE sample $\mathcal{S} = (S_+, S_-, S_H)$, the learner's task is then to construct a formula φ over $\mathcal{P}$ that is *consistent* with $\mathcal{S}$ in that

1. $c \models \varphi$ for each $c \in S_+$;
2. $c \not\models \varphi$ for each $c \in S_-$; and
3. for each $(\{c_1, \ldots, c_n\}, c) \in S_H$, if $c_i \models \varphi$ for all $i \in \{1, \ldots, n\}$, then $c \models \varphi$.

This task is called *passive Horn-ICE learning*, while the overall learning setup can be though of as *iterative (or online) Horn-ICE learning*. In the special case that the learner produces conjunctive formulas, we say that a set $X \subseteq \mathcal{P}$ is consistent with $\mathcal{S}$ if and only if the corresponding conjunction $\bigwedge_{p \in X} p$ is consistent with $\mathcal{S}$.

In general, the Horn-ICE learning framework permits arbitrary formulas over the predicates as candidate invariants. In this paper, however, we exclusively focus on conjunctive formulas (i.e., conjunctions of predicates from $\mathcal{P}$). In fact, conjunctive invariants form an important subclass in practice as they are sufficient to prove many programs correct [18,35] (also see our experimental evaluation in Sect. 4). Moreover, one can design efficient learning algorithms for conjunctive Boolean formulas, as we show next.

2.2 Houdini as a Horn-ICE Learning Algorithm

Houdini [18] is a popular algorithm to synthesize conjunctive invariants in interaction with a theorem prover. For our purposes, however, it is helpful to think of Houdini as an adaptation of the classical elimination algorithm [26] to the Horn-ICE learning framework that is modified to account for Horn counterexamples. To avoid confusion, we refer to algorithmic component that the Houdini learning algorithm as the *"elimination algorithm"* and the implementation of the elimination algorithm as a learner in the context of the Horn-ICE framework as Houdini-ICE.

Let us now describe the elimination algorithm as it is used in the design of Sorcar as well. Given a Horn-ICE sample $\mathcal{S} = (S_+, S_-, S_H)$, the elimination algorithm computes the largest conjunctive formula $X \subseteq \mathcal{P}$ in terms of the number of predicates in X (i.e., the semantically smallest set of program configurations expressible by a conjunctive formula) that is consistent with $\mathcal{S}$. Starting with the set $X = \mathcal{P}$ of all predicates, the elimination algorithm proceeds as follows:

1. The elimination algorithm removes all predicates $p \in X$ from X that violate a positive counterexample (i.e., there exists a positive counterexample $c \in S_+$ such that $c \not\models p$). The result is the unique largest set X of predicates—alternatively the largest conjunctive formula—that is consistent with S_+ (i.e., $c \models X$ for all $c \in S_+$).
2. The elimination algorithm checks whether all Horn counterexamples are satisfied. If a Horn counterexample $(\{c_1, \ldots, c_n\}, c) \in S_H$ is not satisfied, it means that each program configuration c_i of the left-hand-side satisfies X, but the configuration c on the right-hand-side does not. However, X corresponds to the semantically smallest set of program configurations expressible by a conjunctive formula that is consistent with S_+. Moreover, all program configurations c_i on the left-hand-side of the Horn counterexample also satisfy X. Thus, the right-hand-side c necessarily has to satisfy X as well (otherwise X would not satisfy the Horn counterexample). To account for this, the elimination algorithm adds c as a new positive counterexample to S_+.
3. The elimination algorithm repeats Steps 1 and 2 until a fixed point is reached. Once this happens, X is the unique largest set of predicates that is consistent with S_+ and S_H.

Finally, the elimination algorithm checks whether each negative counterexample violates X (i.e., $c \not\models X$ for each $c \in S_-$). If this is the case, X is the largest set of predicates that is consistent with $\mathcal{S}$; otherwise, no consistent conjunctive formula exists. Note that the elimination algorithm does not learn from negative counterexamples.

It is not hard to verify that the time the elimination algorithm spends in each round is *polynomial* in the number of predicates and the size of the Horn-ICE sample (provided predicates can be evaluated in constant time). If the elimination algorithm is employed in the iterative Horn-ICE setting (as Houdini-ICE), it is guaranteed to converge in at most $|\mathcal{P}|$ rounds, or it reports that no conjunctive invariant over $\mathcal{P}$ exists.

The property that HOUDINI-ICE converges in at most $|\mathcal{P}|$ rounds is of great importance in practice. One can, for instance, in every round learn the *smallest* set of conjuncts satisfying the sample, say using a SAT solver. Doing so would not significantly increase the time taken for learning in each round (thanks to highly-optimized SAT solvers), but the worst-case number of iterations to converge to an invariant becomes exponential. An exponential number of rounds, however, makes learning invariants often intractable in practice (we implemented such a SAT-based learner, but it performed poorly on our set of benchmarks). Hence, it is important to keep the number of iterations small when learning invariants. Note that HOUDINI-ICE does not use *negative examples* to learn formulas and, hence, is *not property-driven* (negative examples come from configurations that lead to violating assertions). The SORCAR algorithm, which we describe in the next section, has this feature and aims for potentially weaker invariants that are sufficient to prove the assertions in the program. Note, however, that HOUDINI-ICE is complete in the sense that it is guaranteed to find an inductive invariant that proves the program correct against its assertions, if one exists that can be expressed as a conjunction over the given predicates.

3 The SORCAR Horn-ICE Learning Algorithm

One disadvantage of HOUDINI-ICE is that it learns in each round the largest set of conjuncts, *independent* of negative counterexamples, and, hence, independent of the assertions and specifications in the program—in fact, it learns the semantically smallest inductive invariant expressible as a set of conjuncts over $\mathcal{P}$. As a consequence, HOUDINI-ICE may spend a lot of time finding the tightest invariant (involving many predicates) although a simpler and weaker invariant suffices to prove the program correct. This motivates the development of our novel SORCAR Horn-ICE learning algorithm for conjuncts, which is property-driven (i.e., it also considers the assertions in the program) and has a bias towards learning conjunctions with a smaller number of predicates.

The salient feature of SORCAR is that it always learns invariants involving what we call *relevant* predicates, which are predicates that have shown some evidence to affect the assertions in the program. More precisely, we say that a predicate is *relevant* if it evaluates to *false* on some negative counterexample or on a program configuration appearing on the left-hand-side of a Horn counterexample. This indicates that *not* assuming this predicate leads to an assertion violation or the invariant not being inductive, and is hence deemed important as a candidate predicate in the synthesized invariant. However, naively choosing relevant predicates does, in general, lead to an exponential number of rounds. Thus, SORCAR is designed to select relevant predicates carefully and requires at most $2|\mathcal{P}|$ rounds to converge to an invariant (which is twice the number that HOUDINI-ICE guarantees). Moreover, the set of predicates learned by SORCAR is always a subset of those learned by HOUDINI-ICE.

Algorithm 1. The Sorcar Horn-ICE learning algorithm

```
Function Relevant-Predicates (N, H, X, R):
    return a set of $R' \subseteq \mathcal{P}$ of relevant predicates such that $R' \setminus R \neq \emptyset$;
end

Procedure Sorcar-Passive $\mathcal{S} = (S_+, S_-, S_H)$, $R$:
    Run the elimination algorithm to compute the set $X = \{p_1, \ldots, p_n\}$,
    corresponding to the largest conjunctive formula $\bigwedge_{i=1}^{n} p_i$ over $\mathcal{P}$ that is
    consistent with $\mathcal{S}$ (abort if no such formula exists);
    while $X \cap R$ is not consistent with $\mathcal{S}$ do
        $N \leftarrow \emptyset$;          // Stores inconsistent negative counterexamples
        $H \leftarrow \emptyset$;          // Stores inconsistent Horn counterexamples
        foreach negative counterexample $c \in S_-$ not consistent with $X \cap R$ do
            $N \leftarrow N \cup \{c\}$;
        end
        foreach Horn counterexample $(L, c) \in S_H$ not consistent with $X \cap R$ do
            $H \leftarrow H \cup \{(L, c)\}$;
        end
        $R \leftarrow R \cup$ Relevant-Predicates $(N, H, X, R)$;
    end
    return $(X \cap R, R)$;
end

static $R \leftarrow \emptyset$;           // Stores relevant predicates across rounds
Procedure Sorcar-Iterative$\mathcal{S}$:
    $(Y, R) \leftarrow$ Sorcar-Passive$\mathcal{S}, R$;
    return $Y$;
end
```

Algorithm 1 presents the Sorcar Horn-ICE learner in pseudo code. In contrast to Houdini-ICE, it is not a purely passive learning algorithm but is divided into a passive part (`Sorcar-Passive`) and an iterative part (`Sorcar-Iterative`), the latter being invoked in every round of the Horn-ICE framework. More precisely, `Sorcar-Iterative` maintains a state in form of a set $R \subseteq \mathcal{P}$ in the course of the iterative learning, which is empty in the beginning and used to accumulate *relevant predicates* (Line 19). The exact choice of relevant predicates, however, is delegated to an external function `Relevant-Predicates`. We treat this function as a parameter for the Sorcar algorithm and discuss four possible implementations at the end of this section. Let us now present Sorcar in detail.

3.1 The Passive Sorcar Algorithm

Given a Horn-ICE sample $\mathcal{S}$ and a set $R \subseteq \mathcal{P}$, `Sorcar-Passive` first constructs the largest conjunction $X \subseteq \mathcal{P}$ that is consistent with $\mathcal{S}$ (Line 5). This construction follows the elimination algorithm described in Sect. 2.2 and ensures that X is consistent with all counterexamples in $\mathcal{S}$. Since X is the largest set of predicates consistent with $\mathcal{S}$, it represents the smallest consistent set of program configurations expressible as a conjunction over $\mathcal{P}$. As a consequence, it follows that $X \cap R$—in fact, any subset of X—is consistent with S_+. However, $X \cap R$ might not be consistent with S_- or S_H. To fix this problem, `Sorcar-Passive` collects all inconsistent negative counterexamples in a set N and all inconsistent Horn counterexamples in a set H (Lines 7 to 14). Based on these two sets, `Sorcar-Passive` then computes a set of relevant predicates, which it adds to R (Line 15). As mentioned above, the exact computation of relevant predicates is delegated to a function `Relevant-Predicates`, which we treat as a parameter. The result of this function is a set $R' \subseteq \mathcal{P}$ of predicates that needs to contain at least one new predicate that is not yet present in R. Once such a set has been computed and added to R, the process repeats (R grows monotonically larger) until a consistent conjunctive formula is found. Then, `Sorcar-Passive` returns both the conjunction $X \cap R$ as well as the new set R of relevant predicates. Note that the resulting conjunction is always a subset of the relevant predicates.

The condition of the loop in Line 6 immediately shows that the set $X \cap R$ is consistent with the Horn-ICE sample $\mathcal{S}$ once `Sorcar-Passive` terminates. The termination argument, however, is less obvious. To argue termination, we first observe that X is consistent with each positive counterexample in S_+ and, hence, $X \cap R$ remains consistent with all positive counterexamples during the run of `Sorcar-Passive`. Next, we observe that the termination argument is independent of the exact choice of predicates added to R—in fact, the predicates need not even be relevant in order to prove termination of `Sorcar-Passive`. More precisely, since the function `Relevant-Predicates` is required to return a set $R' \subseteq \mathcal{P}$ that contains at least one new (relevant) predicate not currently present in R, we know that R grows strictly monotonically. In the worst case, the loop in Lines 6 to 16 repeats $|\mathcal{P}|$ times until $R = \mathcal{P}$; then, $X \cap R = X$, which is guaranteed to be consistent with $\mathcal{S}$ by construction of X (see Line 5). Depending on the implementation of `Relevant-Predicates`, however, `Sorcar-Passive` can terminate earlier with a much smaller consistent set $X \cap R \subsetneq X$. Since the time spent in each iteration of the loop in Lines 6 to 16 is proportional to $|\mathcal{P}| \cdot |\mathcal{S}| + f(|\mathcal{S}|)$, where f is a function capturing the complexity of `Relevant-Predicates`, the overall runtime of `Sorcar-Passive` is in $\mathcal{O}\big(|\mathcal{P}|^2 \cdot |\mathcal{S}| + |\mathcal{P}| \cdot f(|\mathcal{S}|)\big)$. This is summarized in the following theorem.

Theorem 1 (Passive Sorcar algorithm). *Given a Horn-ICE sample $\mathcal{S}$ and a set $R \subseteq \mathcal{P}$ of relevant predicates, the passive* Sorcar *algorithm learns a consistent set of predicates (i.e., a consistent conjunction over $\mathcal{P}$) in time $\mathcal{O}\big(|\mathcal{P}|^2 \cdot |\mathcal{S}| + |\mathcal{P}| \cdot f(|\mathcal{S}|)\big)$ where f is a function capturing the complexity of the function* `Relevant-Predicates`.

Before we continue, let us briefly mention that the set of predicates returned by Sorcar is always a subset of those returned by Houdini-ICE.

3.2 The Iterative Sorcar Algorithm

`Sorcar-Iterative` maintains a state in form of a set $R \subseteq \mathcal{P}$ of relevant predicates in the course of the learning process (Line 19). In each round of the Horn-ICE learning framework, the learner invokes `Sorcar-Iterative` with the current Horn-ICE sample $\mathcal{S}$ as input, which contains all counterexamples that the learner has received thus far. Internally, `Sorcar-Iterative` calls `Sorcar-Passive`, updates the set R, and returns a new conjunctive formula, which the learner then proposes as new candidate invariant to the teacher. If `Sorcar-Passive` aborts (because no conjunctive formula over $\mathcal{P}$ that is consistent with $\mathcal{S}$ exists), so does `Sorcar-Iterative`.

To ease the presentation in the remainder of this section, let us assume that the program under consideration can be proven correct using an inductive invariant expressible as a conjunction over $\mathcal{P}$. Under this assumption, the iterative Sorcar algorithm identifies such an inductive invariant in at most $2|\mathcal{P}|$ rounds, as stated in the following theorem.

Theorem 2 (Iterative Sorcar algorithm). *Let P be a program and $\mathcal{P}$ a finite set of predicates over the configurations of P. When paired with an honest teacher that enables progress, the iterative* Sorcar *algorithm learns an inductive invariant (in the form of a conjunctive formula over $\mathcal{P}$) that proves the program correct in at most $2|\mathcal{P}|$ rounds, provided that such an invariant exists.*

Proof (of Theorem 2). We first observe that the computation of the set X in Line 5 of `Sorcar-Passive` always succeeds. This is a direct consequence of the honesty of the teacher (see Sect. 2.1) and the assumption that at least one inductive invariant exists that is expressible as a conjunction over $\mathcal{P}$. This observation is essential as it shows that `Sorcar-Iterative` does not abort.

Next, recall that the teacher enables progress in the sense that every counterexample is inconsistent with the current conjecture (see Sect. 2.1). We use this property to argue that the number of iterations of `Sorcar-Iterative` has an upper bound of at most $2|\mathcal{P}|$, which can be verified by carefully examining the updates of X and R as counterexamples are added to the Horn-ICE sample $\mathcal{S}$:

- If a positive counterexample c is added to $\mathcal{S}$, then it is added because $c \not\models X \cap R$ (as the teacher enforces progress). This implies $c \not\models X$, which in turn means that there exists a predicate $p \in X$ with $c \not\models p$. In the subsequent round of the passive Sorcar algorithm, p is no longer present in X (see Line 5) and $|X|$ decreases by at least one as a result.
- If a negative counterexample c is added to $\mathcal{S}$, then it is added because $c \models X \cap R$ (as the teacher enforces progress). This means that the set X remains unchanged in the next iteration but at least one relevant predicate is added to R in order to account for the new negative counterexample (Line 15). This increases $|R|$ by at least one.
- If a Horn counterexample $(\{c_1, \ldots, c_n\}, c)$ is added to $\mathcal{S}$, then it is added because $c_i \models X \cap R$ for each $i \in \{1, \ldots, n\}$ but $c \not\models X \cap R$ (as the teacher enforces progress). In this situation, two distinct cases can arise:

1. If $(\{c_1, \dots, c_n\}, c)$ is not consistent with X (i.e., $c_i \models X$ for each $i \in \{1, \dots, n\}$ but $c \not\models X$), the computation in Line 5 identifies and removes a predicate $p \in X$ with $c \not\models X$ in order to make X consistent with $\mathcal{S}$. This means that $|X|$ decreases by at least one.
2. If $(\{c_1, \dots, c_n\}, c)$ is consistent with X but not with $X \cap R$, then X remains unchanged. However, at least one new relevant predicate is added to R in order to account for the new Horn counterexample (Line 15). This means that $|R|$ increases by at least one.

Thus, either $|X|$ decreases or $|R|$ increases by at least one.

In the worst case, `Sorcar-Iterative` arrives at a state with $X = \emptyset$ and $R = \mathcal{P}$ (if it does not find an inductive invariant earlier). Since the algorithm starts with $X = \mathcal{P}$ and $R = \emptyset$, this worst-case situation occurs after at most $2|\mathcal{P}|$ iterations.

Let us now assume that `Sorcar-Iterative` indeed arrives at a state with $X = \emptyset$ and $R = \mathcal{P}$. Then, we claim that the result of `Sorcar-Iterative`, namely $X \cap R = \emptyset$, is an inductive invariant. To prove this claim, first recall that Theorem 1 shows that `Sorcar-Passive` always learns a set of predicates that is consistent with the given Horn-ICE sample $\mathcal{S}$. In particular, Line 5 of `Sorcar-Passive` computes the (unique) largest set $X \subseteq \mathcal{P}$ that is consistent with $\mathcal{S}$. Second, we know that every inductive invariant $X^\star$ is consistent with $\mathcal{S}$ because the teacher is honest. Thus, we obtain $X^\star \subseteq X = \emptyset$ and, hence, $X^\star = X$ because both X and $X^\star$ are consistent with $\mathcal{S}$ and X is the largest consistent set. This means that X is an inductive invariant because $X^\star$ is one.

Note, however, that `Sorcar-Iterative` might terminate earlier, in which case the current conjecture is an inductive invariant by definition of the Horn-ICE framework. In summary, we have shown that `Sorcar-Iterative` terminates in at most $2|\mathcal{P}|$ iterations with an inductive invariant (if one is expressible as an conjunctive formula over $\mathcal{P}$). $\square$

Finally, let us note that `Sorcar-Iterative` can also detect if no inductive invariant exists that is expressible as a conjunction over $\mathcal{P}$. In this case, the computation of X in Line 5 of `Sorcar-Passive` fails and the algorithm aborts.

3.3 Computing Relevant Predicates

In the following, we develop *four* different implementations of the function `Relevant-Predicates`. All of these functions share the property that the search for relevant predicates is limited to the set $X \setminus R$ because only predicates in this set can help making $X \cap R$ consistent with negative and Horn counterexamples (cf. Line 6 of Algorithm 1). Moreover, recall that we define a predicate to be relevant if it evaluates to *false* on some negative counterexample or on a program configuration appearing on the left-hand-side of a Horn counterexample. Intuitively, these are predicates in $\mathcal{P}$ that have shown some relevancy in the sense that they can be used to establish consistency with the Horn-ICE sample.

`Relevant-Predicates-Max`. The function `Relevant-Predicates-Max`, shown as Algorithm 2, computes the maximal set of relevant predicates from $X \setminus R$

Algorithm 2. Computing the maximal set of relevant predicates

```
Function Relevant-Predicates-Max (N, H, X, R):
    $R' \leftarrow \emptyset$;
    foreach negative counterexample $c \in N$ do
        $R' \leftarrow R' \cup \{p \in X \setminus R \mid c \not\models p\}$;
    end
    foreach Horn counterexample $(\{c_1, \ldots, c_n\}, c) \in H$ do
        $R' \leftarrow R' \cup \bigcup_{i=1}^{n} \{p \in X \setminus R \mid c_i \not\models p\}$;
    end
    return $R'$;
end
```

with respect to the negative counterexamples in N and the Horn counterexamples in H. To this end, it accumulates all predicates that evaluate to *false* on a negative counterexample in N or on a program configuration appearing on the left-hand-side of a Horn counterexample in H. The resulting set R' can be large, but $X \cap R'$ is guaranteed to be consistent with N and H (because each negative counterexample and each program configuration on the left-hand-side of a Horn counterexample violates at least one predicates in R', the latter causing each Horn counterexample to be violated). Since $X \cap R$ was neither consistent with N nor with H, and since $R' \subseteq X \setminus R$, it follows that R' must contain at least one relevant predicate not in R, thus satisfying the requirement of `Relevant-Predicates`. Finally, the runtime of `Relevant-Predicates-Max` is in $\mathcal{O}(|\mathcal{P}| \cdot |\mathcal{S}|)$ since $X \setminus R \subseteq \mathcal{P}$, $N \subseteq S_-$, and $H \subseteq S_H$.

`Relevant-Predicates-First`. The function `Relevant-Predicates-First` is shown as Algorithm 3. Its goal is to select a smaller set of relevant predicates than `Relevant-Predicates-Max`, while giving the user some control over which predicates to choose. More precisely, `Relevant-Predicates-First` selects for each negative counterexample and each Horn counterexample only one relevant predicate $p \in X \setminus R$. The exact choice is determined by a total ordering $<_\mathcal{P}$ over the predicates, which reflects a preference among predicates and which we assume to be a priori given by the user. Using the same arguments as for the function `Relevant-Predicates-Max`, it is not hard to verify that the resulting set R' contains at least one additional relevant predicate not in R and that $X \cap R'$ is consistent with N and H. Moreover, R' clearly contains only a subset of the predicates returned by `Relevant-Predicates-Max`. Again, the runtime is in $\mathcal{O}(|\mathcal{P}| \cdot |\mathcal{S}|)$.

`Relevant-Predicates-Min`. The function `Relevant-Predicates-Min`, shown as Algorithm 4, takes the idea of `Relevant-Predicates-First` one step further and computes a (not necessarily unique) *minimum* set of relevant predicates with respect to N and H. It does so by means of a reduction to a well-known optimization problem called *minimum hitting set* [25].[3] For a collection

[3] Note that the corresponding decision problem is NP-complete.

Algorithm 3. Computing relevant predicates based on a preference ordering

```
1  Function Relevant-Predicates-First (N, H, X, R):
2      Define a total order $<_\mathcal{P}$ over $\mathcal{P}$;
3      $R' \leftarrow \emptyset$;
4      foreach negative counterexample $c \in N$ do
5          $R' \leftarrow R' \cup \{p\}$ where $p$ is the $<_\mathcal{P}$-smallest predicate with $p \in X \setminus R$ and
           $c \not\models p$;
6      end
7      foreach Horn counterexample $(\{c_1, \dots, c_n\}, c) \in H$ do
8          $R' \leftarrow R' \cup \{p\}$ where $p$ is the $<_\mathcal{P}$-smallest predicate from the set
           $\bigcup_{i=1}^{n} \{p \in X \setminus R \mid c_i \not\models p\}$;
9      end
10     return $R'$;
11 end
```

Algorithm 4. Computing a minimal set of relevant predicates

```
1  Function Relevant-Predicates-Min (N, H, X, R):
2      For each $c \in N$, construct $A_c \coloneqq \{p \in X \setminus R \mid c \not\models p\}$;
3      For each $(L, c) \in H$, construct $A_{(L,c)} \coloneqq \{p \in X \setminus R \mid \exists c' \in L \colon c' \not\models p\}$;
4      Compute a minimal hitting set $R'$ for the instance
       $Q \coloneqq \{A_c \mid c \in N\} \cup \{A_{(L,c)} \mid (L, c) \in H\}$ (e.g., using a SAT solver);
5      return $R'$;
6  end
```

$\{A_1, \dots, A_\ell\}$ of finite sets, a set B is a *hitting set* if $B \cap A_i \neq \emptyset$ for all $i \in \{1, \dots, \ell\}$, and the minimum hitting set problem asks to compute a hitting set of minimum cardinality. In the first step of the reduction, the function `Relevant-Predicates-Min` constructs for each negative counterexample $c \in N$ the set A_c of all predicates $p \in X \setminus R$ violating c and for each Horn counterexample $(L, c) \in H$ the set $A_{(L,c)}$ of all predicates $p \in X \setminus R$ violating some program configuration $c' \in L$. In a second step, it uses an exact algorithm (e.g., a SAT solver) to find a minimum hitting set R' for the problem instance $Q \coloneqq \{A_c \mid c \in N\} \cup \{A_{(L,c)} \mid (L, c) \in H\}$. By construction of the sets A_c and $A_{(L,c)}$, the resulting minimum hitting set R' then is a minimum set of relevant predicates guaranteeing that $X \cap R'$ is consistent with N and H. Moreover, R' contains at least one relevant predicate not in R. However, the downside of approach is that it is not a polynomial time algorithm as the underlying decision problem is NP-complete.

`Relevant-Predicates-Greedy`. The key idea underlying the function `Relevant-Predicates-Greedy`, which is shown as Algorithm 5, is to replace the exact computation of a minimum hitting set with a polynomial-time approximation algorithm. More precisely, `Relevant-Predicates-Greedy` implements a

Algorithm 5. Greedily computing a "small" set of relevant predicates

```
Function Relevant-Predicates-Greedy (N, H, X, R):
    For each $c \in N$, construct $A_c \coloneqq \{p \in X \setminus R \mid c \not\models p\}$;
    For each $(L, c) \in H$, construct $A_{(L,c)} \coloneqq \{p \in X \setminus R \mid \exists c' \in L \colon c' \not\models p\}$;
    $R' \leftarrow \emptyset$;
    $Q \leftarrow \{A_c \mid c \in N\} \cup \{A_{(L,c)} \mid (L, c) \in H\}$;
    while $Q \neq \emptyset$ do
        Pick $p \in X \setminus (R \cup R')$ such that $|\{A \in Q \mid p \in A\}|$ is maximal;
        $R' \leftarrow R' \cup \{p\}$;
        $Q \leftarrow Q \setminus \{A \in Q \mid p \in A\}$;
    end
    return $R'$;
end
```

straightforward greedy heuristic that successively chooses predicates $p \in X \setminus R$ that have the largest number of a non-empty intersections with sets in Q. This heuristic is essentially the dual of the well-known greedy algorithm for the minimum set cover problem [10] and guarantees to find a solution that is at most logarithmically larger than the optimal one. Apart from being an approximation of the minimal set, choosing relevant predicates greedily based on the *number* of sets it hits also has a statistical bias (choosing predicates more commonly occurring in the sets). Otherwise, except for a runtime in $\mathcal{O}(|\mathcal{P}| \cdot |\mathcal{S}|^2)$ and an approximation factor of $\log |\mathcal{S}|$, `Relevant-Predicates-Greedy` shares the same properties as the function `Relevant-Predicates-Min`.

4 Experimental Evaluation

To evaluate the performance of Sorcar, we implement a prototype, featuring all four variants of Sorcar (as well as more heuristics, which we do not discuss here).[4] This prototype is built on top of the program verifier Boogie [4], which natively supports Houdini and provides a so-called "Abstract-Houdini framework" [46] on top of which we have implemented ICE/Horn-ICE algorithms, including Sorcar. Consequently, Sorcar can easily be integrated into existing, Boogie-based verification tool chains.

We compared Sorcar with two Houdini-based tools: GPUVerify [6,9], a tool for checking data race freedom in GPU kernels, and a tool by Neider et al. [35] for verifying programs that dynamically manipulate heaps against specifications in separation logic. Since separation logic is undecidable in general, the latter tool is designed to work in tandem with a sound-but-incomplete verification engine rather than a complete decision procedure. To the best of our knowledge, both tools are the best ones available for their respective domains.

[4] The sources of Sorcar are publicly available at https://github.com/horn-ice/sorcar.

We have evaluated our implementation on two benchmarks suites: the first suite is shipped with GPUVerify, while the second is included in Neider et al.'s tool. As both of these tools use HOUDINI, all benchmarks were already equipped with a large number of predicates (often several hundred). We describe each benchmark suite in more detail shortly.

The goal of our experimental evaluation was twofold: (a) to determine whether SORCAR can prove programs correct that the HOUDINI-based tools cannot (and vice versa) as well as (b) to assess the performance of SORCAR in comparison to these two tools. Since one of the key design principles of SORCAR is to improve verification by constructing weaker invariant (smaller sets of conjuncts), we also report on the size of the invariants (number of conjuncts) inferred by SORCAR and compare to the other tools.

Benchmarks and Compared Tools. The first benchmark suite originates from GPUVerify [6,9] and was obtained from GPU kernels written in OpenCL and CUDA. GPUVerify processes such programs automatically by means of a complex process, involving sequentialization and compilation to the BOOGIE programming language. After removing all programs that did not have loops or recursion, this benchmark suite contained 287 programs.

GPUVerify proceeds in three stages. The first stage compiles an OpenCL or CUDA program into a BOOGIE program. The second stage uses HOUDINI in a custom version of BOOGIE to infer an inductive invariant; in this phase, the assertions are in fact removed as HOUDINI is anyway agnostic to the property being verified. Finally, the third phase substitutes the synthesized invariants, inserts the assertions back into the BOOGIE program, and verifies it.

The second benchmark suite is taken from Neider et al. [35]. It consists of 62 heap manipulating programs, written in C and are equipped with specifications in DRYAD, a dialect of separation logic that allows expressing second order properties using recursive functions and predicates.

Neider et al.'s tool uses the following verification tool chain. First, an extension of VCC [11], called VCDRYAD [37], compiles the C code into a BOOGIE program by unfolding recursive definitions, modeling heaplets as sets, and applying frame reasoning using a technique called natural proofs [31,37,38]. The tool then poses the verification problem as an invariant synthesis problem over a class of predicates that express complex properties of the heap (such as whether the heaplets of two data structures are disjoint, whether a list is sorted, and so on). Finally, Neider et al.'s tool uses HOUDINI to infer a loop invariant.

Note that the final phase of both tools is to synthesize a conjunctive invariant over a fixed set of predicates using HOUDINI. In our experiments, we have replaced HOUDINI with SORCAR.

Evaluation. All experiments were conducted on an Intel Xeon E7-8857 v2 CPU at $3,6$ GHz, running Debian GNU/Linux 9.5. The timeout limit was 1200 s. So as to not clutter the following presentation too much, we only report on the version of SORCAR that performed best: SORCAR-MAX (using

`Relevant-Predicates-Max`). Additionally, we briefly compare SORCAR-MAX to SORCAR-GREEDY, the latter using `Relevant-Predicates-Greedy`.

Figures 2a and 2b compare SORCAR-MAX and GPUVerify on the first benchmark suite consisting of GPU kernels. Figure 2a compares the time taken to verify a program, while Fig. 2b compares the number of predicates in the final invariant (there is only one loop invariant in these programs). As can be seen from the figures, SORCAR-MAX compares highly favorably in efficiency. Specifically, SORCAR-MAX was able to verify 15 programs that GPUVerify could not verify, whereas GPUVerify verified only 2 programs that SORCAR-MAX could not verify. SORCAR-MAX was also able to show 9 programs to not have a conjunctive invariant that GPUVerify could not (GPUVerify was not able to show this for any program that SORCAR-MAX could not). On programs that both tools were able to verify (216 programs in total), SORCAR-MAX took on average 34 s per program (and synthesized invariants with an average number of 12 predicates). GPUVerify, on the other hand, took on average 89 s per program (and synthesized invariants with an average number of 23 predicates).

Additionally (not depicted in the scatter plots), we increased the time limit for programs that only one tool could verify from 1200 s to 3600 s. GPUVerify was able to verify 8 additional programs within this time limit. SORCAR, on the other hand, verified both programs that it had timed out on previously. Thus, with this larger timeout, SORCAR was able to verify a proper superset of programs that GPUVerify verified.

Figures 2c and 2d compare SORCAR-MAX to the tool of Neider et al. [35] on the second benchmark suite of programs with DRYAD specifications. Again, SORCAR-MAX outperformed the HOUDINI-based tool. Specifically, SORCAR-MAX was able to verify 3 programs that Neider et al.'s tool could not verify, whereas Neider et al.'s tool verified 2 programs that SORCAR-MAX could not verify. On programs that both tools were able to verify (57 programs in total), SORCAR-MAX took on average 20 s per program (and synthesized invariants with an average number of 19 predicates). On the other hand, Neider et al.'s tool took on average 45 s per program (and synthesized invariants with an average number of 37 predicates).

Figures 2e and 2f compare SORCAR-MAX and SORCAR-GREEDY on both benchmark suits. The latter was slightly slower overall, but synthesized invariants with fewer predicates.

Comparison of SORCAR and HOUDINI-ICE. Close to the time of writing the final version of this paper, we performed further experiments with HOUDINI-ICE (i.e., an implementation of HOUDINI as a Horn-ICE learning algorithm based on the elimination algorithm), as suggested by the reviewers. This allowed us to force the number of counterexamples returned by BOOGIE in each round to be the same for SORCAR and HOUDINI-ICE (a parameter over which we do not have control in BOOGIE's implementation of HOUDINI).

On the GPUVerify benchmark suite, SORCAR-MAX verified 5 programs that HOUDINI-ICE could not, whereas HOUDINI-ICE was able to verify 1 program that SORCAR-MAX could not. HOUDINI-ICE was also able to show 2 programs to

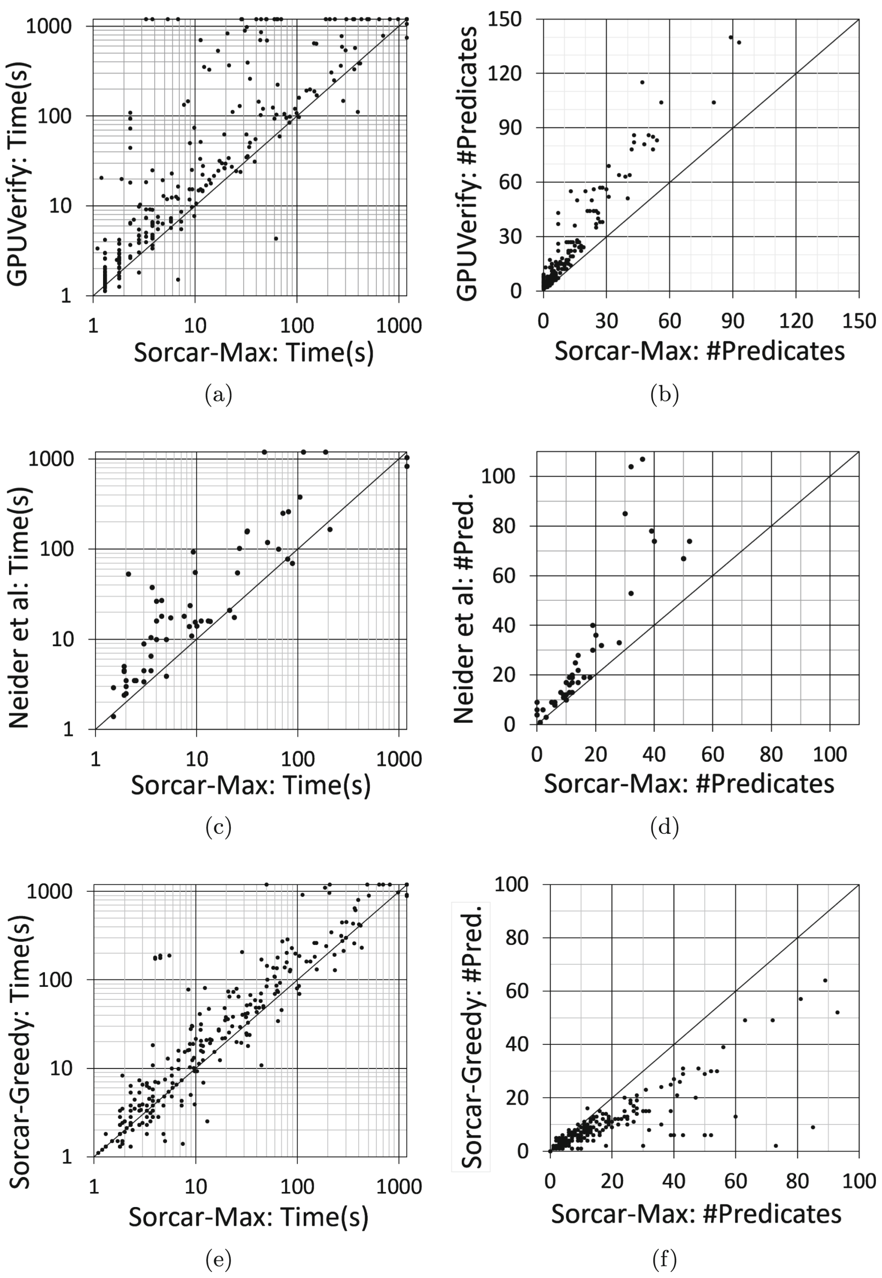

Fig. 2. Comparison of the time taken to verify a benchmark and the number of predicates in the final invariant. Subfigures (a) and (b) compare SORCAR-MAX and GPUVerify on the first benchmark suite. Subfigures (c) and (d) compare SORCAR-MAX and Neider et al.'s tool on the second benchmark suite. Subfigures (e) and (f) compare SORCAR-MAX and SORCAR-GREEDY on both benchmark suites.

not have a conjunctive invariant, which Sorcar-Max could not. On programs that both were able to verify (233 programs in total), both algorithms performed with similar times.

On the Dryad benchmark suite, Sorcar-Max was able to solve 1 more program than Houdini-ICE (and verified all programs that Houdini-ICE verified). On the 59 programs that both could verify, Sorcar-Max was roughly twice as fast (averaging 24 s per program for Sorcar-MAX vs. 51 s per program for Houdini-ICE).

While Sorcar-Max still emerges better overall than Houdini-ICE, we are not entirely sure why implementing Houdini as an external Horn-ICE learning algorithm makes it perform much better than the internal implementation of Houdini in Boogie (the internal Houdini algorithm within Boogie is embedded deep and is very hard to configure or control). For the GPUVerify benchmarks, the tool GPUVerify does invariant synthesis without assertions and then inserts assertions to verify the program, and this could be one difference. We leave answering this question for future work.

5 Conclusion

In this paper, we have developed a new class of learning algorithms for conjunctions, named Sorcar, which are biased towards the simplest conjunctive invariant that can prove the assertions correct. Sorcar is parameterized by functions to identify relevant predicates and guarantees to learn an invariant in a linear number of rounds (if one exists). We have shown that Sorcar proves programs correct significantly faster than state-of-the-art Houdini-based tools.

There are several future directions to pursue. First, we believe that further algorithms for learning conjunctions need to be explored. For instance, the Winnow algorithm [30] learns from positive and negative samples in time $\mathcal{O}(r \log n)$, where r is the size of the final formula and n is the number of predicates. Finding Horn-ICE learning algorithms that have such sublinear round guarantees can be very interesting as r is often much smaller than n in verification examples. Second, we would like to use the new Sorcar algorithms in specification mining settings where smaller invariants are valuable as they are read by humans. Third, there are several types of inference algorithms similar to Houdini (see [39]), and it would be interesting to explore how well Sorcar performs in such settings.

Acknowledgements. We thank the reviewers for their many valuable suggestions that helped improve this paper. This material is based upon work supported by the National Science Foundation under Grant No. 1527395.

References

1. Alur, R., et al.: Syntax-guided synthesis. In: Dependable Software Systems Engineering, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 40, pp. 1–25. IOS Press (2015)

2. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM **54**(7), 68–76 (2011)
3. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, 20–22 June 2001, pp. 203–213. ACM (2001)
4. 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
5. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
6. Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: 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, 21–25 October 2012, pp. 113–132. ACM (2012)
7. 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
8. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: ICE-based refinement type discovery for higher-order functional programs. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 365–384. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_20
9. Chong, N., Donaldson, A.F., Kelly, P.H.J., Ketema, J., Qadeer, S.: Barrier invariants: a shared state abstraction for the analysis of data-dependent GPU kernels. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, 26–31 October 2013, pp. 605–622. ACM (2013)
10. Chvátal, V.: A greedy heuristic for the set-covering problem. Math. Oper. Res. **4**(3), 233–235 (1979)
11. Cohen, E., et al.: 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
12. 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
13. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, pp. 238–252. ACM (1977)
14. Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, 26–31 October 2013, pp. 443–456. ACM (2013)

15. Ernst, M.D., Czeisler, A., Griswold, W.G., Notkin, D.: Quickly detecting relevant program invariants. In: Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, 4–11 June 2000, pp. 449–458. ACM (2000)
16. Ezudheen, P., Neider, D., D'Souza, D., Garg, P., Madhusudan, P.: Horn-ICE learning for synthesizing invariants and contracts. PACMPL **2**(OOPSLA), 131:1–131:25 (2018)
17. Fedyukovich, G., Kaufman, S.J., Bodík, R.: Sampling invariants from frequency distributions. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October 2017, pp. 100–107. IEEE (2017)
18. 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
19. 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
20. 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
21. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp. 499–512 (2016)
22. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China, 11–16 June 2012, pp. 405–416. ACM (2012)
23. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, 7–13 June 2008, pp. 281–292. ACM (2008)
24. 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
25. Karp, R.M.: Reducibility among combinatorial problems. In: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA, pp. 85–103. The IBM Research Symposia Series, Plenum Press, New York (1972)
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). http://arxiv.org/abs/1501.04725
28. Lal, A., Qadeer, S.: Powering the static driver verifier using corral. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, 16–22 November 2014, pp. 202–212. ACM (2014)
29. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 427–443. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_32

30. Littlestone, N.: Learning quickly when irrelevant attributes abound: a new linear-threshold algorithm. Mach. Learn. **2**(4), 285–318 (1987)
31. Löding, C., Madhusudan, P., Peña, L.: Foundations for natural proofs and quantifier instantiation. Proc. ACM Program. Lang. **2**(POPL), 10 (2017)
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. Microsoft: Static driver verifier. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier. Accessed 26 Apr 2019
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. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 232–250. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_13
36. Nori, A.V., Rajamani, S.K., Tetali, S.D., Thakur, A.V.: The Yogi project: software property checking via static analysis and testing. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 178–181. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_17
37. Pek, E., Qiu, X., Madhusudan, P.: Natural proofs for data structure manipulation in C using separation logic. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 440–451. ACM (2014)
38. Qiu, X., Garg, P., Ştefănescu, A., Madhusudan, P.: Natural proofs for structure, data, and separation. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, pp. 231–242. ACM, New York (2013). https://doi.org/10.1145/2491956.2462169
39. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, 7–13 June 2008, pp. 159–169. ACM (2008)
40. 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
41. 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
42. 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
43. 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
44. Solar-Lezama, A.: Program Synthesis by Sketching. Ph.D. thesis, University of California at Berkeley (2008)

45. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, 21–25 October 2006, pp. 404–415. ACM (2006)
46. Thakur, A., Lal, A., Lim, J., Reps, T.: Posthat and all that: automating abstract interpretation. Electron. Notes Theor. Comput. Sci. **311**, 15–32 (2015)
47. Vizel, Y., Gurfinkel, A., Shoham, S., Malik, S.: IC3 - flipping the E in ICE. In: Bouajjani, A., Monniaux, D. (eds.) VMCAI 2017. LNCS, vol. 10145, pp. 521–538. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52234-0_28
48. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, 18–22 June 2018, pp. 707–721. ACM (2018)