IC3 - Flipping the E in ICE

Yakir Vizel[1], Arie Gurfinkel[2], Sharon Shoham[3], and Sharad Malik[1]

[1] Princeton University, USA
[2] University of Waterloo, Canada
[3] Tel-Aviv University, Israel

Abstract. Induction is a key element of state-of-the-art verification techniques. Automatically synthesizing and verifying inductive invariants is at the heart of *Model Checking* of safety properties. In this paper, we study the relationship between two popular approaches to synthesizing inductive invariants: SAT-based Model Checking (SAT-MC) and Machine Learning-based Invariant Synthesis (MLIS). Our goal is to identify and formulate the theoretical similarities and differences between the two frameworks. We focus on two flagship algorithms: IC3 (an instance of SAT-MC) and ICE (an instance of MLIS). We show that the two frameworks are very similar yet distinct. For a meaningful comparison, we introduce *RICE*, an extension of ICE with *relative induction* and show how IC3 can be implemented as an instance of RICE. We believe this work contributes to the understanding of inductive invariant synthesis and will serve as a foundation for further improvements to both SAT-MC and MLIS algorithms.

1 Introduction

State-of-the-art verification techniques [7, 1, 2, 9, 5, 3] use *induction* to verify a system with respect to a safety specification. Typically, this is a two phase process. In the first phase, a candidate inductive invariant *Inv* is chosen (the details of this choice depend on each particular technique). The second phase checks that *Inv* is really an inductive invariant and is usually done by a variety of techniques including symbolic execution, simulation, SAT- and SMT-solving. In this paper, we are focusing on *invariant inference* techniques that automate both phases.

An inductive invariant *Inv* represents a set of states closed under forward reachability: *Inv* includes all the initial states, and all states reachable from states in *Inv*. Thus, an inductive invariant over-approximates the set of all reachable states. The invariant is called *safe* when it satisfies the specification (i.e., it does not include any states that reach a *bad* state). It is possible for an invariant to be non-inductive (i.e., not closed under reachability) or non-safe (include a *bad* state). In the context of safety verification, invariant inference is restricted to discovering safe inductive invariants. While there are many such approaches, in this paper, we study the relationship between *SAT-based model checking (SAT-MC)* [1], *Machine Learning-based Invariant Synthesis (MLIS)* [8, 3, 4, 6].

SAT-based model checking algorithms, such as IC3/PDR [1, 2], Interpolation [7] and others [9, 5], are based on safe inductive invariant inference. These

algorithms iteratively conjecture safe candidate invariants until either a candidate is proven inductive or a counterexample is found. While these algorithms differ in the details of candidate discovery, they all rely on a close interaction between the main algorithm and the decision procedure used. MLIS algorithms, such as ICE [3], have been recently proposed as a Machine Learning-based alternatives for invariant inference. MLIS algorithms have two main components: a *Learner* and a *Teacher*. In each iteration of the algorithm, a learner generates a candidate invariant, and the teacher validates the conjecture and, if necessary, generates feedback in a form of a positive or negative example. The learner uses the example to improve its conjecture. The process continues until the teacher determines that the candidate is a safe inductive invariant. While, to our knowledge, ICE does not produce counterexamples, we ignore this distinction in this paper.

The goal of this paper is to study and formulate the theoretical similarities and differences between SAT-MC and MLIS approaches by focusing on the two flagship algorithms: IC3 and ICE. We concede that the comparison is somewhat complicated by the fact that ICE is a generic framework (with many potential instances), while IC3 (and algorithms that derive from it) are specific instances of SAT-based MC. At a high-level of abstraction, IC3 and ICE exhibit many similarities. It is quite natural to view the SAT-solver in IC3 as a *Teacher* and the rest of the algorithm as a *Learner* with satisfying assignments playing the role of examples. However, instead of a single candidate, IC3 maintains a collection of candidates (called a *trace* of *frames*). It reacts to the examples given by the SAT-solver by strengthening and weakening these candidates, until one of the candidates is determined to be safe and inductive. The goal of this paper is to formalize this correspondence.

When all the low-level details are taken into account, IC3 does not fit into the ICE framework. In particular, the main query used by IC3 is not whether a candidate is inductive, but whether it is inductive *relative* to a given formula. Thus, we extend the ICE framework with relative induction, calling the result *RICE*. We then show that, with the exception of several details, IC3 is as an instance of RICE and that decoupling IC3 into a Learner and a Teacher enables various different behaviors, which depend on both how the Learner reacts to examples given by the Teacher, and to how the Teacher generates the examples. This leads to many interesting observations.

As an example, IC3 only makes use of positive examples that are part of the *initial states*. Yet, positive examples in RICE may be any reachable state. Due to that, fitting IC3 into RICE requires care when encountering reachable states. We show that when reachable states are taken into account, the results is a variant of IC3, namely Quip [5].

2 Preliminaries

In this section, we briefly present two algorithms for safety verification: IC3/PDR and ICE. To simplify the presentation, we describe the algorithms in the finite-

state case, where the transition system to be verified is encoded using propositional formulas.

2.1 Safety verification

A symbolic transition system T is a tuple $(\mathcal{V}, \mathit{Init}, \mathit{Tr}, \mathit{Bad})$, where $\mathcal{V}$ is a set of Boolean variables. A state of the system is a complete valuation to all variables in $\mathcal{V}$ (i.e., the set of states is $\{0,1\}^{\mathcal{V}}$). We write $\mathcal{V}'$ $(= \{v' \mid v \in \mathcal{V}\})$ for the set of *primed* variables, used to represent the next state. *Init* and *Bad* are formulas over $\mathcal{V}$ denoting the set of initial states and bad states, respectively, and *Tr* is a formula over $\mathcal{V} \cup \mathcal{V}'$, denoting the transition relation. With abuse of notation, we use formulas and the sets of states (or transitions) that they represent interchangeably. In addition, we sometimes use a state s to denote the formula (cube) that characterizes it. For a formula φ over $\mathcal{V}$, we use $\varphi(\mathcal{V}')$, or φ' in short, to denote the formula in which every occurrence of $v \in \mathcal{V}$ is replaced by $v' \in \mathcal{V}'$.

For a set of states S and a transition relation Tr, the forward image, $\mathit{post}^i_{\mathit{Tr}}(S)$, is the set of states reachable from S in i steps or less of the Tr. Formally, it is defined as follows:

$$\begin{cases} \mathit{post}^0_{\mathit{Tr}}(S) = S \\ \mathit{post}^i_{\mathit{Tr}}(S) = \mathit{post}^{i-1}_{\mathit{Tr}}(S) \cup \{t \mid s \in S \wedge (s,t) \in \mathit{Tr}\} \end{cases} \tag{1}$$

Similarly, the backward image $\mathit{pre}^i_{\mathit{Tr}}(S)$, is the set of all states that can reach S in i steps or less. Formally, it is defined as follows:

$$\begin{cases} \mathit{pre}^0_{\mathit{Tr}}(S) = S \\ \mathit{pre}^i_{\mathit{Tr}}(S) = \mathit{pre}^{i-1}_{\mathit{Tr}}(S) \cup \{t \mid s \in S \wedge (t,s) \in \mathit{Tr}\} \end{cases} \tag{2}$$

We often omit the transition relation Tr when it is clear from the context, writing $\mathit{post}^i(S)$ and $\mathit{pre}^i(S)$. We denote transitive closure of *post* and *pre* as post^* and pre^*, respectively.

We say that a state s in T is reachable if $s \in \mathit{post}^*(\mathit{Init})$, and a state s is *bad* when $s \in \mathit{pre}^*(\mathit{Bad})$. A transition system T is UNSAFE when the intersection of $\mathit{post}^*(\mathit{Init})$ and $\mathit{pre}^*(\mathit{Bad})$ is non-empty. That is, T has a bad reachable state. T is SAFE when $\mathit{post}^*(\mathit{Init})$ and $\mathit{pre}^*(\mathit{Bad})$ are disjoint. It is well known that T is SAFE iff there exists a formula *Inv*, called a *safe inductive invariant*, that satisfies:

$$\mathit{Init}(\mathcal{V}) \Rightarrow \mathit{Inv}(\mathcal{V}) \quad \mathit{Inv}(\mathcal{V}) \wedge \mathit{Tr}(\mathcal{V}, \mathcal{V}') \Rightarrow \mathit{Inv}(\mathcal{V}') \quad \mathit{Inv}(\mathcal{V}) \Rightarrow \neg \mathit{Bad}(\mathcal{V}) \tag{3}$$

2.2 IC3/PDR

IC3 [1, 2] is a SAT-based MC algorithm for safety verification. It maintains a growing sequence of candidate invariants, referred to as an *inductive trace*. An *inductive trace*, or simply a trace, is a sequence of CNF formulas $\boldsymbol{F} = [F_0, \ldots, F_N]$

that satisfy:

$$Init \Rightarrow F_0 \qquad \forall 0 \leq i < N \cdot F_i(\mathcal{V}) \wedge Tr(\mathcal{V}, \mathcal{V}') \rightarrow F_{i+1}(\mathcal{V}') \tag{4}$$

The *size* of $\boldsymbol{F} = [F_0, \ldots, F_N]$ is N. For a $k \leq N$, $\boldsymbol{F}^k = [F_0, \ldots, F_k]$ is a k-prefix of $\boldsymbol{F}$. Each element F_i of the trace is called a *frame*, and each conjunct of F_i is called a *lemma*. With abuse of notation, we sometimes view a frame as a set of lemmas.

A trace is *safe* if each F_i is safe: $\forall i \cdot F_i \Rightarrow \neg Bad$; it is *monotone* if $\forall 0 \leq i < N \cdot F_i \Rightarrow F_{i+1}$. It is easy to see that the definition of a monotone inductive trace ensures that F_i overapproximates $post^i_{Tr}(S)$.

Given a monotone safe trace $[F_0, \ldots, F_N]$, if there exists i s.t. $i < N$ and $F_{i+1} \Rightarrow F_i$, then F_i is a safe inductive invariant.

A high-level description of IC3 is given in Algorithm 1. Our presentation of IC3 is rather non-standard, but we find it useful for the rest of the presentation. Many details are omitted and simplified. For a more standard and complete presentation, we refer the reader to [1, 2]. In addition to a monotone (and safe up to F_{N-1}) trace, IC3 maintains a queue Q of *Counterexamples To Induction (CTIs)*, such that $Q \subseteq pre^*(Bad)$. The function MinBad returns the minimal frame in $\boldsymbol{F}$ that includes a state $s \in Q$. If no such frame is found, a new frame is initialized to *true*, and MinBad returns a state $s \notin Bad$ from the new frame (s is also added to Q). The function Strengthen (line 4) tries to strengthen the frame returned by MinBad by blocking the bad state s at the corresponding frame. While doing so, it iteratively adds more bad states from $pre^*(Bad)$ to Q. If an initial state is added to Q, IC3 terminates and returns a counterexample (line 5). Otherwise, it uses the fact that a state $s \in pre^*(Bad)$ is unreachable in a given number of steps to add a lemma that strengthens the proper candidate invariants. The function Push (line 7) "pushes" lemmas from one frame to a subsequent frame. A lemma ℓ in F_j indicates that some states (i.e., $\neg\ell$) are unreachable in up to j steps. If a lemma is pushed to F_{j+1}, it indicates those states are unreachable in up to $j + 1$ steps. In case all lemmas from F_j were pushed to F_{j+1}, the candidate F_j is a safe inductive invariant and IC3 terminates (line 7).

IC3 is incremental both in the way it computes the candidates (via iterative strengthening), and the way it uses previously learned lemmas during its execution.

Importantly, the discovery of bad states, generation of lemmas and pushing, are preformed using *relative induction* queries. A formula *RelInv* is inductive relative to a formula G if it satisfies:

$$Init(\mathcal{V}) \Rightarrow RelInv(\mathcal{V}) \qquad (RelInv(\mathcal{V}) \wedge G(\mathcal{V})) \wedge Tr(\mathcal{V}, \mathcal{V}') \Rightarrow RelInv(\mathcal{V}') \tag{5}$$

For example, given a trace $\boldsymbol{F} = [F_0, \ldots, F_N]$, assume that IC3 tries to block a bad state $s \in F_i$, where $0 < i \leq N$. s is unreachable in i steps when both $Init \wedge s$ and $(F_{i-1} \wedge \neg s) \wedge Tr \wedge s'$ are unsatisfiable, i.e. if $\neg s$ is inductive relative to F_{i-1}. In this case, s can be removed from F_i by adding some lemma c, such that $Init \Rightarrow c$ and $c \Rightarrow \neg s$, to F_i. The lemma c is found by repeatedly executing

Input: A transition system $T = (\mathcal{V}, \mathit{Init}, \mathit{Tr}, \mathit{Bad})$

```
1 F_0 ← Init ; F = [F_0, true] ; Q ← ∅;
2 repeat
3 |  (s, k) ← MinBad(F, Q);
4 |  r ← Strengthen(s, k, Q);
5 |  if r = cex then return UNSAFE;
6 |  if r = blocked then
7 |  |  if Push(F) then return SAFE;
8 until ∞;
```

Algorithm 1: IC3/PDR: Main loop.

relative induction queries starting with $\neg s$ and greedily dropping literals. The details are irrelevant for our purposes, but are available in [1, 2].

2.3 ICE

ICE [3] is an ML-based framework for synthesizing safe inductive invariants. A high level description of ICE is shown in Algorithm 2. Following the ML paradigm of active learning, ICE consists of two components: a *Teacher* and a *Learner*. The Teacher and the Learner communicate in the following manner: The Learner synthesizes a candidate invariant J (line 11) and sends it to the Teacher (line 12) that either confirms J is a safe inductive invariant (line 13) or finds an (counter-)example that rules J out (line 14). In the latter case, the Learner uses the examples returned by the Teacher to synthesize a different candidate. Note that Q accumulates all examples returned by the Teacher. The newly synthesized candidate must conform with all of the examples the Teacher has reported.

Definition 1 (ICE Examples). *Let S be a set of states. An* ICE state *is a triple (E, C, I), where $E \subseteq S$ is a set of* E-Examples*, $C \subseteq S$ is a set of* C-Examples*, and $I \subseteq S \times S$ is a set of* I-Examples.

Definition 2. *Let S be a set of states. A set $J \subseteq S$ is* consistent *with an ICE state (E, C, I) if (a) $E \subseteq J$, (b) $J \cap C = \emptyset$ and (c) for every $(s, t) \in I$, if $s \in J$, then $t \in J$.*

In the context of learning inductive invariants, S is the set of states in a transition system T. Each E-example is a reachable state of T (i.e., $E \subseteq \mathit{post}^*(\mathit{Init})$), each C-example is a state that reaches a bad state (i.e., $C \subseteq \mathit{pre}^*(\mathit{Bad})$), and each I-example is a pair of states $(s, t) \in \mathit{Tr}$.

The Teacher in the ICE framework must return one of the ICE-Examples as a reason showing why J is not a safe inductive invariant, according to the following rules:

- I-example: $\exists (s, t) \in \mathit{Tr} \cdot s \in J \wedge t \notin J$
- C-example: $\exists s \in \mathrm{pre}^*(\mathit{Bad}) \cdot s \in J$

Input: A transition system $T = (\mathcal{V}, \mathit{Init}, \mathit{Tr}, \mathit{Bad})$

```
9  Q ← ∅ ; Learner() ; Teacher(T);
10 repeat
11 |  J ← Learner.SynCandidate(Q);
12 |  ε ← Teacher.IsInd(J);
13 |  if ε = ⊥ then return SAFE;
14 |  Q ← Q ∪ {ε};
15 until ∞;
```

Algorithm 2: ICE: Main loop.

– E-example: $\exists s \in \mathrm{post}^*(\mathit{Init}) \cdot s \notin J$

If multiple choices are possible, the Teacher is free to chose an example to return, but it must return an example if one exists.

If J is not a safe inductive invariant, the Learner needs to modify it in order to be consistent with the new example provided by the Teacher. The required modification depends on the kind of an example the Teacher returns. Given a C-example, J must be strengthened to exclude the bad state, while for an E-example, J must be weakened to include the reachable state. In the case of an I-example (s, t), both options are possible. More precisely, either s is removed from J or t is included in J. The ICE framework leaves the details of how the candidate is improved to an underlying learning algorithm. The only requirement is that each new candidate must be consistent with the current ICE state. In particular, the new candidate might be incomparable to the previous one: it does not have to be the result of pure strengthening or weakening.

3 The RICE Framework for Invariant Synthesis

In this section, we present our MLIS framework *RICE* that extends ICE with relative induction. In an MLIS framework such as ICE, the Learner can only control the Teacher indirectly through the query interface. On the other hand, MC-SAT algorithms such as IC3 rely on a tight integration with a SAT-solver. In Section 4, we show that allowing for relatively inductive queries provides the Learner sufficient power to control the Teacher.

As in ICE, a RICE state is a triple (E, C, I) (see Definition 1). Recall that F is inductive relative to G when $F \wedge G \wedge \mathit{Tr} \Rightarrow F'$. A pair (F, G) is consistent with a RICE state (E, C, I), if F is consistent (as in ICE) with E and C, and F is consistent with all pairs $\{(s, t) \in I \mid s \in G\}$. Formally,

Definition 3. *Let S be a set of states in a transition system T. A set $F \subseteq S$ is* consistent *with a RICE state (E, C, I) relative to a set $G \subseteq S$, if $E \subseteq F$, $F \cap C = \emptyset$ and for every $(s, t) \in I$, if $s \in F \cap G$, $t \in F$ as well.*

That is, the *base* set G can be used to communicate to the Teacher which I-examples the Learner is interested in.

Input: A transition system $T = (\mathcal{V}, \mathit{Init}, \mathit{Tr}, \mathit{Bad})$

16 $Q \leftarrow \emptyset$; $H \leftarrow \mathit{false}$;
17 LEARNER() ; TEACHER(T);
18 **repeat**
19 | $(F, G) \leftarrow$ LEARNER.SYNCANDANDBASE(Q);
20 | $\varepsilon \leftarrow$ TEACHER.ISRELIND(F, G);
21 | **if** $\varepsilon = \bot \wedge G = \mathit{true}$ **then return** SAFE;
22 | $Q \leftarrow Q \cup \{\varepsilon\}$;
23 **until** ∞;

Algorithm 3: RICE.

Accordingly, the Teacher in a RICE framework must adhere to the following rules. Given a formula F (called a *candidate*) and a formula G (called a *base*), then if any one of the following holds:

- $\exists (s,t) \in \mathit{Tr} \cdot s \in (F \wedge G) \wedge t \notin F$ (I-example)
- $\exists s \in \mathrm{pre}^*(\mathit{Bad}) \cdot s \in F$ (C-example)
- $\exists s \in \mathrm{post}^*(\mathit{Init}) \cdot s \notin F$ (E-example)

the Teacher returns a corresponding ICE-example, and, otherwise, it returns $\bot$ to indicate that no ICE-example exists. We refer to such a Teacher as a RICE-Teacher.

The RICE framework is shown in Algorithm 3. There are a few major differences compared to ICE. First, the Learner returns a pair (F, G) as a candidate and asks the Teacher if F is inductive relative to G (lines 19-20). If an ICE-example exists, it is added to Q and the Learner must synthesize a new pair (F, G) that is consistent with Q (Definition 3). Second, when no ICE example exists the behavior depends on the value of G. Whenever G is *true* the Learner asks if F is a safe inductive invariant (similar to ICE). Thus, only when $G = \mathit{true}$ and no ICE-examples exists, RICE terminates declaring F a safe inductive invariant (line 21).

Note that termination of the RICE framework depends on the Learner setting G to *true* occasionally. It is easy to see that RICE simulates ICE whenever the learner sets G to *true* infinitely often: When G is set to *true*, the set F in the pair (F, G) is in fact an ICE-candidate (it is consistent with the ICE-state). Therefore, the sequence of steps between two subsequent candidates with $G = \mathit{true}$ can be viewed as an active variant of the ICE Learner: instead of computing a candidate F in isolation, it communicates with the Teacher. In this sense, RICE is a generalization of ICE, in which the Learner is an active learning algorithm.

Lemma 1. *RICE terminates if $G = \mathit{true}$ infinitely often.*

4 IC3 as an Instance of (R)ICE

RICE and ICE are generic MLIS frameworks. In this section, we show that IC3 is an instance of RICE by suggesting an IC3-specific implementations for both the Learner and the Teacher.

Re-factoring IC3 into a Teacher and a Learner. In an MLIS framework, the Learner does not have a direct access to the transition system T. It learns information about it only through the interaction with the Teacher. Thus, when re-factoring IC3 for an MLIS framework, we delegate the role traditionally taken by the SAT-solver to the Teacher and keep the rest for the Learner.

Challenges in simulating the IC3 queries. While RICE supports relative induction queries, these do not immediately correspond to the queries performed by IC3. There are two reasons for that. First, the requirement of the Learner to be consistent with the current RICE state, and second, the requirement of the Teacher to return an ICE-example when it exists. Namely, if neither I-example nor E-example exist, F is inductive relative to G. Yet, the converse does not necessarily hold – relative induction only requires $\mathit{Init} \Rightarrow F$, hence, E-examples might exist even when F is inductive relative to G. Moreover, if F contains a bad state, the Teacher will return a C-example even when F is inductive relative to G. This highlights the complexity in simulating relative induction queries of IC3 using the Teacher-Learner paradigm.

4.1 The IC3-Learner

Contrary to the "white-box" implementation of IC3, in the MLIS framework the Learner has no control over the "type" of answers the Teacher returns (e.g., the Teacher is free to choose between C- and I-examples when both are available). Therefore, the Learner must be robust such that soundness and completeness are maintained even when the Teacher is adversarial.

Similarly to the original IC3, the IC3-Learner maintains a monotone inductive trace $\boldsymbol{F}$. In addition, it maintains the RICE state, $Q = (I, C, E)$, containing the ICE-examples returned by the Teacher (which can be viewed as a generalization of the set of CTIs maintained by IC3). However, the IC3-Learner maintains the I-examples *explicitly*. These are only maintained implicitly by IC3 – they are captured by the lemmas that were not pushed.

Detailed description of the IC3-Learner. The IC3-Learner is shown in Algorithm 4. It maintains an inductive trace $\boldsymbol{F}$ and the RICE state, $Q = (I, C, E)$, containing the ICE-examples returned by the Teacher, where $C(Q)$, $E(Q)$ and $I(Q)$ denote all C-examples, E-examples and I-examples in Q, respectively. Intuitively, from the learning perspective, each element of the trace is a candidate to be a safe inductive invariant. The Learner refines those candidates incrementally based on answers returned by the Teacher. Each iteration of the loop either discovers a new C-example that is added to Q or removes a C-example from a

Input: initial states *Init* and a RICE-Teacher TEACHER

24 $F_0 \leftarrow \mathit{Init}$; $\boldsymbol{F} = [F_0, \mathit{true}]$; $Q \leftarrow \emptyset$;
25 **repeat**
26 | $(s, k) \leftarrow \text{MINBAD}(\boldsymbol{F}, Q)$; // s is a C-example
27 | $r \leftarrow \text{STRENGTHEN}(s, k, Q)$;
28 | **if** $r = \mathit{cex}$ **then return** UNSAFE;
29 | **if** $r = \mathit{blocked}$ **then**
30 | | **if** $\text{PUSH}(\boldsymbol{F}, Q, k-1)$ **then return** SAFE;
31 **until** ∞;

Algorithm 4: IC3-Learner.

Input: a trace $\boldsymbol{F}$ and a set of ICE-examples Q

32 $N \leftarrow \boldsymbol{F}.\mathit{size}()$; $k \leftarrow \min(\{N+1\} \cup \{j \mid C(Q) \cap F_j \neq \emptyset\})$;
33 **if** $k < N+1$ **then** $s \leftarrow$ a state from $C(Q) \cap F_k$;
34 **else**
35 | $\varepsilon \leftarrow \text{TEACHER.ISRELIND}(F_N \wedge \neg C(Q), \mathit{true})$;
36 | **if** $\varepsilon \neq \bot$ **then** $Q \leftarrow Q \cup \{\varepsilon\}$;
37 | **if** ε *is a C-example* **then**
38 | | $s \leftarrow \varepsilon$; $k \leftarrow \min\{j \mid C(Q) \cap F_j \neq \emptyset\}$
39 | **else**
40 | | $\boldsymbol{F}.\mathit{addFrame}(\mathit{true})$;
41 | | $s \leftarrow$ a state from $C(Q)$; $k \leftarrow N+1$;
42 **return** (s, k);

Algorithm 5: IC3-Learner: implementation of MINBAD.

given candidate F_k (line 27). In the latter case, other candidates are refined as well based on the refinement of F_k (line 30).

The candidate F_k and the C-example to be removed are set by calling MINBAD (line 26). MINBAD (shown in Algorithm 5) returns the minimal element of $\boldsymbol{F}$ that contains a C-example from Q. In particular, if no C-example from $C(Q)$ exists in any element of $\boldsymbol{F}$, MINBAD calls the Teacher. If the Teacher returns a C-example, then it necessarily belongs to one of the existing frames. Hence, it is added to Q and the minimal frame, F_k, in which it resides is returned. Otherwise, a new frame is initialized to *true*. The new frame necessarily contains one of the existing C-examples from $C(Q)$, and is therefore returned. Since MINBAD returns the minimal k, we have that $\forall 0 \leq i < k \cdot F_i \Rightarrow \neg C(Q)$. We refer to such a prefix $\boldsymbol{F}^{k-1}$ as Q-safe:

Definition 4. *Let $\boldsymbol{F} = [F_0, \ldots, F_N]$ be an inductive trace, Q a RICE state and $0 \leq j \leq N$. If $\forall 0 \leq i \leq j \cdot F_i \Rightarrow \neg C(Q)$, then $\boldsymbol{F}^j$ is Q-safe.*

Removing C-examples by strengthening. The IC3-Learner's function STRENGTHEN is shown in Algorithm 6. It is used to remove a C-example s from frame F_k. If $k = 0$, i.e., the C-example is in F_0 then no safe inductive invariant exists and,

Input: a state s, index k, set of ICE-examples Q

43 **if** $k = 0$ **then return** cex;
44 $\varepsilon \leftarrow$ TEACHER.ISRELIND($\neg s \wedge \neg C(Q), F_{k-1}$);
45 **if** ε *is an E-example* **then return** cex;
46 **if** $\varepsilon = \bot$ **then**
47 | $c \leftarrow$ GENERALIZE($\neg s, F_{k-1}, F_k, Q$);
48 | $\forall 0 \leq j \leq k \cdot F_j \leftarrow F_j \wedge c$;
49 | **return** blocked;
50 $Q \leftarrow Q \cup \{\varepsilon\}$;
51 **return** not_blocked;

Algorithm 6: IC3-Learner: implementation of STRENGTHEN.

thus, STRENGTHEN concludes a counterexample exists. Otherwise, it asks the Teacher if $\neg s \wedge \neg C(Q)$ is inductive relative to F_{k-1}. Note that this is in contrast to the original IC3 that checks if $\neg s$ is inductive relative to F_{k-1}. The conjunct $\neg C(Q)$ is added to the query in order to conform with the requirement of providing a candidate that is consistent with the RICE state. In particular, it prevents the Teacher from returning a C-example that already exists in Q. Moreover, using this query is sound since the property that $F_{k-1} \Rightarrow \neg C(Q)$ ($\boldsymbol{F}^{k-1}$ is Q-safe) ensures that relative inductiveness of $\neg s \wedge \neg C(Q)$ w.r.t. F_{k-1} implies relative inductiveness of $\neg s$ as well.

If an E-example e is returned, then a counterexample to the property must exist (line 45) since the reachable state e is either s or some other state in $C(Q)$. On the other hand, if either an I-example or a C-example is returned, STRENGTHEN treats it as a new C-example that is added to Q (line 50). The justification for treating I-examples as C-examples is that in this case an I-example represents a predecessor to a bad state (either s, or another state in $C(Q)$), and is, therefore, also a C-example.

In case no ICE-example is returned, $\neg s \wedge \neg C(Q)$ is relatively inductive (and, hence, also $\neg s$), and a new lemma that blocks s is added to F_k and to all previous frames (lines 47-49).

Finding a new lemma by generalization. Whenever a C-example s is blocked, the fact that $\boldsymbol{F}^{k-1}$ is Q-safe implies that

$$F_0 \Rightarrow \neg s \qquad\qquad (F_{k-1} \wedge \neg s)(\mathcal{V}) \wedge Tr(\mathcal{V}, \mathcal{V}') \Rightarrow \neg s(\mathcal{V}') \tag{6}$$

holds. Generalization relies on the above and tries to find a lemma c such that $c \Rightarrow \neg s$, $F_0 \Rightarrow c$ and

$$(F_{k-1} \wedge c)(\mathcal{V}) \wedge Tr(\mathcal{V}, \mathcal{V}') \Rightarrow c(\mathcal{V}') \tag{7}$$

The generalization procedure is shown in Algorithm 7. The invariant of the main loop is that c is inductive relative to F_i. The loop starts by looking for a new stronger lemma d s.t. $d \Rightarrow c$. If d is not consistent with the current I- and E-examples, a new d is tried. Note that $\neg s$ itself is consistent with the I- and

Input: a lemma $\neg s$, frames F_i and F_{i+1}, and a set of ICE-examples Q

52 $c \leftarrow \neg s$;
53 **repeat**
54 | choose d s.t. $d \Rightarrow c$;
55 | **if** $d \equiv c$ **then break**;
56 | **if** $\neg$IE-CONSISTENT(d, F_i, Q) **then continue**;
57 | $\varepsilon \leftarrow$ TEACHER.ISRELIND$(d \wedge \neg C(Q) \wedge F_{i+1}, F_i)$;
58 | **if** $\varepsilon = \bot$ **then** $c \leftarrow d$;
59 | **else** $Q \leftarrow Q \cup \{\varepsilon\}$;
60 **until** ∞;
61 **return** c

Algorithm 7: IC3-Learner: implementation of GENERALIZE.

Input: a lemma l, a frame F_i, and a set of ICE-examples Q

62 **if** $\exists (s,t) \in I(Q) \cdot s \in (F_i \wedge l) \wedge t \notin l$ **then return** $false$;
63 **if** $\exists s \in E(Q) \cdot s \notin l$ **then return** $false$;
64 **return** $true$

Algorithm 8: IC3-Learner: implementation of IE-CONSISTENT.

E-examples (this is ensured by STRENGTHEN), hence a d that is consistent with the I- and E-examples always exists. Otherwise, a relative induction query is sent to the Teacher (line 57). If an ICE-example is found, it is added to Q. Otherwise, $d \wedge \neg C(Q) \wedge F_{i+1}$ is inductive relative to F_i, and therefore c is updated. The loop terminates, and the corresponding lemma is returned, when c does not change.

It is important to note that while GENERALIZE tries to find a lemma d that is inductive relative to F_i, it asks the Teacher if $d \wedge \neg C(Q) \wedge F_{i+1}$ is inductive relative to F_i. The addition of $\neg C(Q)$ to the query is similar in reason and justification to STRENGTHEN. As for the addition of F_{i+1}, the following properties:

$$\begin{cases} F_i(\mathcal{V}) \Rightarrow F_{i+1}(\mathcal{V}), \text{ and} \\ F_i(\mathcal{V}) \wedge Tr(\mathcal{V}, \mathcal{V}') \Rightarrow F_{i+1}(\mathcal{V}') \end{cases} \tag{8}$$

imply that the query is equivalent to asking if $d \wedge \neg C(Q)$ is inductive relative to F_i. Adding F_{i+1} to the candidate is meant to prevent the Teacher from returning C-examples (bad states) that are already known to be blocked by F_{i+1}.

Making conjectures by pushing. Similarly to the original IC3, the IC3-Learner also tries to push lemmas (line 30). PUSH is shown in Algorithm 9. PUSH takes a trace $\boldsymbol{F}$, a RICE state Q, and an index k such that $\boldsymbol{F}^k$ is Q-safe. Given a lemma $c \in F_i$ for $i \leq k$, the Learner first checks if pushing is prevented by any of the known I-examples or E-examples (line 68). If not, the Learner asks if $c \wedge \neg C(Q) \wedge F_{i+1}$ is inductive relative to F_i. Again, we emphasize that this is sufficient to show that c is inductive relative to F_i. Therefore, if no ICE-example is returned, c is added to F_{i+1} (line 70).

In addition, any I-example that is returned by the Teacher is either an indication that c is not inductive relative to F_i or that a C-example exists. This is justified by the following lemma:

Lemma 2. *Let Q be a RICE state, and $\boldsymbol{F}$ be an inductive trace such that $\boldsymbol{F}^k$ is Q-safe. For $0 \leq i \leq k$ and $c \in F_i$, if $(s,t) \in Tr$ has the property that $s \in F_i \wedge c \wedge \neg C(Q) \wedge F_{i+1}$ and $t \notin c \wedge \neg C(Q) \wedge F_{i+1}$, then $s \in F_i \wedge c$ and in addition either (i) $t \notin c$ or (ii) $i = k$ and s is a bad state in F_k.*

Proof. The fact that $s \in F_i \wedge c$ is trivial. As for the second part, consider first the case where $i < k$. Note that in this case $(c \wedge \neg C(Q) \wedge F_{i+1}) \equiv c \wedge F_{i+1}$ (due to $\boldsymbol{F}^k$ being Q-safe). Thus, $t \notin c \wedge F_{i+1}$, or equivalently, $t \in \neg c \vee \neg F_{i+1}$. Let us assume $t \in \neg F_{i+1}$, this contradicts $s \in F_i$ and $F_i \wedge Tr \Rightarrow F'_{i+1}$. By that, $t \in \neg c$, equivalently, $t \notin c$. Now consider the case where $i = k$. In this case F_{k+1} may include a C-example. Still, $t \in \neg F_{k+1}$ does not hold (due to the same argument as in the $i < k$ case). Hence $t \in \neg c \vee C(Q)$. Thus, either $t \in \neg c$ or $t \in C(Q)$. In the former case, $t \notin c$. Otherwise, s is a bad state (a predecessor of a C-example). □

Therefore, in case an I-example is returned by the Teacher, it is added to Q and c is not pushed since it is not inductive relative to F_i. If the post-state of the I-example is in c, the I-example is treated as a C-example (this can only happen when $i = k$).

However, there are additional possibilities. Since $c \wedge \neg C(Q) \wedge F_{i+1}$ may include a bad state, or exclude a reachable state, a C-example or an E-example are possible. Recall that we assume a Teacher that returns ICE-examples when those exist, therefore, if a C- or E-example exists it is returned even if $c \wedge \neg C(Q) \wedge F_{i+1}$ is inductive relative to F_i.

Whenever a C-example is returned by the Teacher, PUSH terminates (line 73) and the IC3-Learner goes back to the main loop to process a new candidate and a new C-example. In case of an E-example, it is added to Q and c is skipped.

Remark 1. The use of IE-CONSISTENT in lines 56 and 68 prevents the need to approach the Teacher in cases where a previous answer that is recorded in the RICE state Q already provides the desired information (e.g., whether a lemma is relative inductive). In this sense, the IC3-Learner can be understood as an optimized version of IC3 which uses memoization.

Safe Inductive Invariant. In case all lemmas in F_i are pushed to F_{i+1}, then F_i is an inductive invariant. If F_i is also safe, i.e. $F_i \Rightarrow \neg Bad$, then the IC3-Learner terminates and reports that a safe inductive invariant is found. Recall that the IC3-Learner maintains an inductive trace, and, therefore, checking for safety is done by asking the Teacher if F_i is inductive relative to *true* (line 75). Since we know that it is indeed an inductive invariant, if F_i is not safe, the Teacher returns a new C-example, and the IC3-Learner goes back to the main loop.

Lemma 3. *If the IC3-Learner returns UNSAFE, a counterexample exists.*

Input: a trace $\boldsymbol{F}$, a set of ICE-examples Q, a number k

65 **foreach** $i \leftarrow 1$ **to** k **do**
66 $r \leftarrow \mathit{true}$;
67 **foreach** $c \in F_i$ **do**
68 **if** $\neg$IE-CONSISTENT(c, F_i, Q) **then** $(r \leftarrow \mathit{false}\,;$ **continue**$)$;
69 $\varepsilon \leftarrow$ TEACHER.ISRELIND$(c \wedge \neg C(Q) \wedge F_{i+1}, F_i)$;
70 **if** $\varepsilon = \bot$ **then** $F_{i+1} \leftarrow F_{i+1} \wedge c$;
71 **else**
72 $r \leftarrow \mathit{false}\,; Q \leftarrow Q \cup \{\varepsilon\}$;
73 **if** TYPE$(\varepsilon) = \mathit{C\text{-}example}$ **then return** *false*;
74 **if** r **then**
75 $\varepsilon \leftarrow$ TEACHER.ISRELIND(F_i, true);
76 **if** $\varepsilon = \bot$ **then return** *true*;
77 $Q \leftarrow Q \cup \{\varepsilon\}$;
78 **return** *false*;

Algorithm 9: IC3-Learner: implementation of PUSH.

Proof. A counterexample is detected in Algorithm 6. There are two cases: either a C-example s is also an initial state and, therefore, $\mathit{Init} \cap \mathit{pre}^*(\mathit{Bad}) \neq \emptyset$, or a C-example is found to be an E-example and, therefore, $\mathit{post}^*(\mathit{Init}) \cap \mathit{pre}^*(\mathit{Bad}) \neq \emptyset$.

We conclude this section with a few formal claims on soundness and completeness of IC3-Learner.

Lemma 4. *If the IC3-Learner returns SAFE, a safe inductive invariant exists.*

Lemma 5. *For a finite-state system T, the IC3-Learner terminates.*

Shortcomings of the IC3-Learner. Since the Teacher must produce an ICE-example whenever it exists, relative induction queries will fail as long as there is a potential E- or C-example. Since both generalization and pushing are done via relative induction queries, this means that the IC3-Learner is limited to generating frames F_i that are between reachable ($\mathit{post}^*(\mathit{Init}) \Rightarrow F_i$) and safe ($F_i \Rightarrow \neg \mathit{pre}^*(\mathit{Bad})$) regions. Note, however, that multiple frames might still be constructed because the frames must be closed under the transitions of the system. The search for such a frame might construct several candidates (i.e., frames). Another consequence of the Teacher's specification is that a frame will never be added before all the previous frames become safe. Therefore, MINBAD will always return a state from the last frame. In the next section, we address these limitations by placing additional restrictions on the Teacher.

4.2 An IC3-Teacher

In the previous section, we described an IC3-style Learner that assumes a generic Teacher. The behavior of the IC3-Learner is highly dependent on the answers

Input: A candidate F, a base G, numbers m and n
if $\exists s \in pre^m(Bad) \cdot s \in F$ **then return** C-example s;
if $\exists (s,t) \in Tr \cdot s \in (F \wedge G) \wedge t \notin F$ **then return** I-example (s,t);
if $\exists s \in post^n(Init) \cdot s \notin F$ **then return** E-example s;
return $\bot$;

Algorithm 10: IC3-Teacher: implementation of IsRelInd.

returned by the Teacher. In order for the IC3 MLIS framework to behave like IC3, some additional restrictions must be imposed on the Teacher.

Requiring the Teacher to always produce E- and C-examples when they exists is unrealistic. Such a Teacher is required to look arbitrary far into the future (for E-examples) or past (for C-examples). In most IC3 implementations there are two implicit assumptions. First, *Init* is known a-priori and E-examples are restricted to $post^0(Init)$ (which is simply *Init*). Second, in every query to the Teacher, C-examples (i.e., bad states) are restricted to $pre^m(Bad)$ for some fixed m (which is determined by the algorithm and changes between queries). These two assumptions relate to the fact that IC3 maintains a safe inductive trace. Therefore, F_i over-approximates $post^i(Init)$, and is disjoint from $pre^{N-i}(Bad)$. Note that these requirements are easier to achieve, than over-approximating $post^*(Init)$, and being disjoint from $pre^*(Bad)$, respectively.

We use these observations to suggest an IC3-Teacher shown in Algorithm 10. Using IC3-Teacher, the IC3-Learner can restrict the E- and C-examples returned by setting specific bounds on them. By using this IC3-Teacher, the resulting algorithm is closer to IC3 than the one that uses the unrestricted RICE-Teacher.

Considering the IC3-Teacher from Algorithm 10, we write $\textsc{IsRelInd}(\psi, F_i, \infty, \infty)$ for the non-restrictive calls (i.e., parameters m and n are unlimited).

Lemma 6. *If every call of the form* $\textsc{IsRelInd}(\psi, F_i, \infty, \infty)$ *in IC3-Learner is replaced by* $\textsc{IsRelInd}(\psi, F_i, N-i-1, 0)$*, then the combination of the IC3-Learner (Algorithm 4) and IC3-Teacher (Algorithm 10) simulates IC3. In particular, every generalization and pushing step of IC3 is also feasible in the MLIS implementation.*

Note that one could consider the restricted IC3-Teacher as an abstraction of the unrestricted IC3-Teacher. The abstraction is with respect to the answers the Teacher returns. More precisely, the IC3-Teacher can indicate that a candidate is relative inductive more often, since it can ignore C- and E-examples when a bound is applied. The key element in both the IC3-Teacher and RICE-Teacher, is to never miss an I-example. In fact, restricting the IC3-Teacher to return C- and E-examples from $pre^0(Bad)$ and $post^0(Init)$ respectively (i.e. using $\textsc{IsRelInd}(\psi, F_i, 0, 0)$) will also allow to simulate IC3, and will also be simulated by IC3, making the two simulation-equivalent.

5 IC3 Variants

The behavior of RICE is determined by how the Learner reacts to ICE-examples, and how the Teacher generates them. In this section, we analyze how different instances for both the IC3-Learner and IC3-Teacher affect the result.

Recall that in (R)ICE, the Learner suggests a candidate invariant, and the Teacher either confirms the candidate or gives an ICE-example that rules out the candidate. In the case of an E- and C-example s, the Learner must either include (E-example) or exclude (C-example) s from the candidate. In the case of an I-example (s, t), the Learner can either exclude s or include t. This gives some freedom to the Learner. In this section, we show how the IC3 Learner can handle I-examples, and show that different implementations result in different variants of IC3.

5.1 IC3 and Quip

IC3-Learner in Section 4.1 reacts to all ICE-examples. This is in contrast to a classical IC3 that is built around blocking C-examples, while E-examples are only treated implicitly by considering *Init*, and I-examples are implicitly present in generalization and pushing. In all cases, IC3 *weakens* the candidate with which the I-example is associated.

A variant of IC3 called Quip [5] is different. It is based on the following observation: given a lemma $c \in F_i$, if c cannot be pushed to F_{i+1}, then either F_i is too weak to support c, or c cannot be pushed because it excludes a reachable state. In the context of RICE, if an I-example (s, t) is returned when trying to push c, either s can be removed from F_i, or an E-example exists that shows that c can never be part of an inductive invariant. Note that the IC3-Learner described previously uses this observation implicitly when checking whether a lemma c is consistent with all E-examples.

The pushing phase of an alternative IC3-Learner that explicitly exploits the above observation is shown in Algorithm 11. The main differences compared to the previously described Push (Algorithm 9) are in line 88 and line 94. Note that the function FindReachOrStrengthen (called in line 94) is similar to how IC3 handles C-examples. It receives a state s, and either blocks it (by strengthening $\boldsymbol{F}$) or proves s is reachable. Using FindReachOrStrengthen and allowing the Teacher to find E-examples that are not restricted to *Init* only reflect the changes to how the IC3-Learner reacts to I- and E-examples during pushing. We describe these differences in more detail below:

E-examples: Quip uses E-examples that are not restricted to *Init* (line 88). If a lemma excludes a reachable state it is marked to never be pushed again. Note that this already exists in the IC3-Learner to ensure that a candidate is always consistent with the current RICE state. Yet, in general, this is not present in the classical IC3. Furthermore, Quip forces the Teacher to discover reachable states from I-examples using FindReachOrStrengthen. FindReachOrStrengthen is similar to how IC3 handles C-examples but discovers reachable states that are added as E-examples.

Input: a trace $\boldsymbol{F}$, a set of all ICE examples Q, a number k

```
83  foreach i ← 1 to k do
84      r ← true;
85      foreach c ∈ F_i do
86          if ¬IE-CONSISTENT(c, F_i, Q) then (r ← false ; continue);
87          repeat
88              ε ← TEACHER.ISRELIND(c ∧ ¬C(Q), F_i, N − i − 1, i + 1);
89              if ε = ⊥ then (F_{i+1} ← F_{i+1} ∧ c ; break);
90              else
91                  Q ← Q ∪ {ε};
92                  if ε is a C-example then return false;
93                  else if ε is an I-example (s, t) then
94                      if FINDREACHORSTRENGTHEN(c, F, Q, i − 1) = reach
                        then
95                          Add t to Q as an E-example;
96                          r ← false;
97                          break;
98          until ∞;
99          if r then
100             ε ← TEACHER.ISRELIND(F_i, true);
101             if ε = ⊥ then return true;
102             Q ← Q ∪ {ε};
103 return false
```

Algorithm 11: QUIPPUSH.

I-examples: In IC3, when a lemma $c \in F_i$ cannot be pushed due to an I-example, it is skipped and not added to F_{i+1}. Hence, F_{i+1} is weaker than F_i. In contrast, QUIP tries to strengthen F_i. For example, let (s,t) be an I-example. QUIP treats s as a bad state and tries to remove it from F_i by calling FINDREACHORSTRENGTHEN (line 94). If it succeeds it can try and push c again. Otherwise, QUIP discovers that s is reachable in i steps and, therefore, t is reachable in $i+1$ steps. In this case, t is an E-example.

Note that if the Teacher happens to prioritize E-examples over I-examples, then the QUIP-Learner does not extract reachable states from I-examples, but does still force pushing by strengthening.

The above description is not confined to pushing only. I- and E-examples can also be used during generalization. Note that currently the function GENERALIZE calls IE-CONSISTENT and thus takes into account every E-example in Q.

In addition, *weakening* can be applied differently. Currently, if a given lemma $c \in F_i$ is non-pushable, instead of giving up on that lemma and weakening by not adding it to F_{i+1}, the IC3-Learner can choose to weaken the lemma itself such that it becomes pushable.

6 Conclusions

In this paper, we introduced the RICE framework for Machine Learning-based Invariant Synthesis (MLIS). RICE is an extension of the ICE MLIS framework [3, 4]. It extends the MLIS-Teacher to support relative-induction queries. This improves the communication between the MLIS-Teacher and the MLIS-Learner. In particular, it allows the Learner to be statefull (i.e., *active* in Machine Learning terminology) and better guide the Teacher to desired responses.

We showed that a well-known SAT-based Model Checking (SAT-MC) algorithm IC3 [1] can be formulated as an instance of RICE. We do so, by designing a specific IC3-Learner and a corresponding IC3-Teacher in the MLIS framework. We showed how ICE-examples are handled by the IC3-Learner, and explicated sufficient requirements on the IC3-Teacher for the combination to simulate the IC3 algorithm. Furthermore, we showed that different variants of IC3 correspond to different instances of the IC3-Teacher. In particular, we show that a recent SAT-MC algorithm QUIP [5] is an instance of the RICE MLIS framework in which the Teacher does not restrict E-examples to the set of initial states *Init*, and the Learner guides the Teacher to discover useful E-examples.

Our work opens many avenues for future exploration in similarities and differences between MLIS and SAT-MC frameworks for invariant synthesis. From the MLIS perspective, an interesting question is how to include other active learning algorithms effectively into the framework and how to make the Learner incremental. From the SAT-MC perspective, an interesting question is how to incorporate new forms of weakening other than "pushing". For example, individual lemmas can be weakened by adding literals to make the frames consistent with all (implicit) ICE-examples generated by the run of a SAT-MC algorithm so far. We hope to explore these fruitful directions in the future.

Acknowledgments. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement No. [321174].

References

1. A. R. Bradley. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 70–87, 2011.
2. N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation of property directed reachability. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 125–134, 2011.
3. P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 69–87, 2014.

4. P. Garg, D. Neider, P. Madhusudan, and D. Roth. 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, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 499–512, 2016.
5. A. Gurfinkel and A. Ivrii. Pushing to the Top. In *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pages 65–72, 2015.
6. C. Löding, P. Madhusudan, and D. Neider. Abstract learning frameworks for synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 167–185, 2016.
7. K. L. McMillan. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, pages 1–13, 2003.
8. R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 574–592, 2013.
9. Y. Vizel and A. Gurfinkel. Interpolating property directed reachability. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 260–276, 2014.