Data-driven Numerical Invariant Synthesis with Automatic Generation of Attributes

Ahmed Bouajjani[1], Wael-Amine Boutglay[1,2(✉)], and Peter Habermehl[1]

[1] Université Paris Cité, IRIF, Paris, France
{abou,boutglay,haberm}@irif.fr
[2] Mohammed VI Polytechnic University, Ben Guerir, Morocco

Abstract. We propose a data-driven algorithm for numerical invariant synthesis and verification. The algorithm is based on the ICE-DT schema for learning decision trees from samples of positive and negative states and implications corresponding to program transitions. The main issue we address is the discovery of relevant attributes to be used in the learning process of numerical invariants. We define a method for solving this problem guided by the data sample. It is based on the construction of a separator that covers positive states and excludes negative ones, consistent with the implications. The separator is constructed using an abstract domain representation of convex sets. The generalization mechanism of the decision tree learning from the constraints of the separator allows the inference of general invariants, accurate enough for proving the targeted property. We implemented our algorithm and showed its efficiency.

Keywords: Invariant synthesis · Data-driven program verification

1 Introduction

Invariant synthesis for program safety verification is a highly challenging problem. Many approaches exist for tackling this problem, including abstract interpretation, CEGAR-based symbolic reachability, property-directed reachability (PDR), etc. [3,5,6,8,10,14,17,19]. While those approaches are applicable to large classes of programs, they may have scalability limitations and fail to infer certain types of invariants, such as disjunctive invariants. Emerging data-driven approaches, following the active learning paradigm with various machine learning techniques, have shown their ability to solve efficiently complex instances of the invariant synthesis problem [12,15,16,20,26,30,31]. These approaches are based on the iterative interaction between a *learner* inferring candidate invariants from a *data sample*, i.e., a set of data classified either as positive examples, known to be reachable from the initial states and that therefore must be included in any solution, or negative examples, known to be predecessors of states violating the safety property and that therefore cannot be included in any solution,

This work was supported in part by the french ANR project AdeCoDS.

S. Shoham and Y. Vizel (Eds.): CAV 2022, LNCS 13371, pp. 282–303, 2022.
https://doi.org/10.1007/978-3-031-13185-1_14

and a *teacher* checking the validity of the proposed solutions and providing counterexamples as feedback in case of non-validity. One such data-driven approach is ICE [15] which has shown promising results with its instantiation ICE-DT [16] that uses decision trees for the learning component. ICE is a learning approach tailored for invariant synthesis, where the feedback provided by the teacher can be, in addition to positive and negative examples, implications of the form $p \rightarrow q$ expressing the fact that if p is in a solution, then necessarily q should also be included in the solution since there is a transition in the program from p to q.

The strength of data-driven approaches is the generalization mechanisms of their learning components, allowing them to find relevant abstractions from a number of examples without exploring the whole state space of the program. In the case of ICE-DT, this is done by a sophisticated construction of decision trees classifying correctly the known positive and negative examples at some point, and taking into account the information provided by the implications. These decision trees, where the tested attributes are predicates on the variables of the program, are interpreted as formulas corresponding to candidate invariants.

However, to apply data-driven methods such as ICE-DT, one needs to have a pool of attributes that are potentially relevant for the construction of the invariant. This is actually a crucial issue. In ICE-DT, as well as in most data-driven methods, finding the predicates involved in the invariant construction is based on systematic enumeration of formulas according to some pre-defined templates or grammars. For instance, in the case of numerical programs, the considered patterns are some special types of linear constraints, and candidate attributes are generated by enumerating all possible values for the coefficients under some fixed bound. While such a brute-force enumeration can be effective in many cases, it represents, in general, an obstacle for both scalability and finding sufficiently accurate inductive invariants in complex cases.

In this paper, we provide an algorithmic method for efficient generation of attributes for data-driven invariant synthesis for numerical programs manipulating integer variables. While enumerative approaches are purely syntactic and do not take into account the data sample, our method is guided by it. We show that this method, when integrated in the ICE-DT schema, leads to a new invariant synthesis algorithm outperforming state-of-the-art methods and tools.

Our method for attributes discovery is based on, given an ICE data sample, computing a *separator* of it as a union of convex sets i.e., (1) it covers all the positive examples, (2) it does not contain any negative example, and (3) it is consistent with the implications (for every $p \rightarrow q$ in the sample, if the separator contains p, then it should also contain q). Then, the set of attributes generated is the set of all constraints defining the separator. However, as for a given sample there might be several possible separators, a question is which separators to consider. Our approach is guided by two requirements: (1) we need to avoid big pools of attributes in order to reduce the complexity of the invariant construction process, and (2) we need to avoid having in the pool constraints that are (visibly) unnecessary, e.g. separating positive examples in a region without any negative ones. Therefore, we consider separators that satisfy the property that, whenever

they contain two convex sets, it is impossible to take their convex union (smallest convex set containing the union) without including a negative example.

To represent and manipulate algorithmically convex sets, we consider abstract domains, e.g., intervals, octagons, and polyhedra, as they are defined in the abstract interpretation framework and implemented in tools such as APRON [18]. These domains correspond to particular classes of convex sets, defined by specific types of linear constraints. In these domains, the union operation is naturally over-approximated by the *join* operation that computes the best over-approximation of the union in the considered class of convex sets. Then, constructing separators as explained above can be done by iterative application of the join operation while it does not include negative examples.

Then, this method for generating candidate attributes can be integrated into the ICE-DT schema: in each iteration of ICE loop, given a sample, the learner (1) generates a set of candidate attributes from a separator of the sample, (2) builds a decision tree from these attributes and proposes it as a candidate invariant to the teacher. Then, the teacher (1) checks that the proposed solution is an inductive invariant, and if it is not (2) provides a counterexample to the learner, extending the sample that will be used in the next iteration.

Here a question might be asked: why do we need to construct a decision tree from the constraints of the separator and do not propose directly the formula defining the separator as a candidate invariant to the teacher. The answer is that the decision tree construction is crucial for generalization. Indeed, given a sample, the constructed separator might be too specialized to that sample and does not provide a useful inductive invariant (except for some simple cases). For instance, the constructed separator is a union of *bounded* convex sets (polytopes), while invariants are very often unbounded convex sets (polyhedra). The effect of using decision trees, in this case, is to select the relevant constraints and discard the unnecessary bounds, leading very quickly to an unbounded solution that is general enough to be an inductive invariant. Without this generalization mechanisms, the ICE loop will not terminate in such (quite common) cases.

The integration of our method can be made tighter and more efficient by making the process of building separators incremental along the ICE iterations: at each step, after the extension of the sample by the teacher, instead of constructing a separator of the new sample from scratch, the parts of previously computed separators not affected by the last extension of the sample are reused.

We have implemented our algorithm and carried out experiments on the SyGuS-Comp'19 benchmarks. Our method solves significantly more cases than the tools LoopInvGen [25,26], CVC4 [1,27], and Spacer [19], as well as our implementation of the original ICE-DT [16] algorithm (with template-based enumeration of attributes), with very competitive time performances.

Related Work. Many learning-based approaches for the verification of numerical programs have been developed recently. One of the earliest approaches is Daikon [11]. Given a pool of formulas, it computes likely invariants from program executions. Later approaches were developed for the synthesis of sound invariants, for example [30] iteratively generates a set of reachable and bad states and

classifies them with a combination of half-spaces computed using SVM. In [29], the problem is reformulated as learning geometric concepts in machine learning. The first instantiation of the ICE framework was based on a constraint solver [15]. Later on, it was instantiated using the decision trees learning algorithm [16]. Both those instantiations require a fixed template for the invariants or the formulas appearing in them. LoopInvGen enumerates predicates on-demand using the approach introduced in [26]. This is extended to a mechanism with hybrid enumeration of several domains or grammars [25]. Continuous logic networks were also used to tackle the problem in CLN2INV [28]. Code2Inv [31], the first approach to introduce general deep learning methods to program verification, uses a graph neural network to capture the program structure and reinforcement learning to guide the search heuristic of a particular domain.

The learning approach of ICE and ICE-DT has been generalized to solve problems given as constrained horn clauses (CHC) in Horn-ICE [12] and HoICE [4]. Outside the ICE framework, [33] proposed a learning approach for solving CHC using decision trees and SVM for the synthesis of candidate predicates from a set of reachable and bad states of the program. The limitation of the non-ICE-based approach is that when the invariant is not inductive, the program has to be rerun, forward and backward, to generate more reachable and bad states.

In more theoretical work, an abstract learning framework for synthesis, introduced in [21], incorporates the principle of CEGIS (counterexample-guided inductive synthesis). A study of overfitting in invariant synthesis was conducted in [25]. ICE was compared with IC3/PDR in terms of complexity in [13]. A generalization of ICE with relative inductiveness [32] can implement IC3/PDR following the paradigm of active learning with a learner and a teacher.

Automatic invariant synthesis and verification has been addressed by many other techniques based on exploring and computing various types of abstract representations of reachable states (e.g., [3,5,6,8,10,14,17,19]). Notice that, although we use abstract domains for representation and manipulation of convex sets, our strategy for exploring the set of potential invariants is different from the ones used typically in abstract interpretation analysis algorithms [8].

2 Safety Verification Using Learning of Invariants

This section presents the approach we use for solving the safety verification problem. It is built upon the ICE framework [15] and in particular its instantiation with the learning of decision trees [16]. We first define the verification problem.

2.1 Linear Constraints and Safety Verification

Let X be a set of variables. Linear formulas over X are boolean combinations of linear constraints of the form $\sum_{i=1}^{n} a_i x_i \leq b$ where the x_i's are variables in X, the a_i's are integer constants, and $b \in \mathbb{Z} \cup \{+\infty\}$. We use linear formulas to reason symbolically about programs with integer variables. Assume we have a program with a set of variables V and let $n = |V|$. A state of the program is a

vector of integers in $\mathbb{Z}^n$. Primed versions of these variables are used to encode the transition relation T of the program: for each $v \in V$, we consider a variable v' to represent the value of v after the transition. Let V' be the set of primed variables, and consider linear formulas over $V \cup V'$ to define the relation T.

The *safety verification problem* consists in, given a set of safe states *Good*, deciding whether, starting from a set of initial states *Init*, all the reachable states by iterative application of T are in *Good*. Dually, this is equivalent to decide if starting from *Init*, it is possible to reach a state in *Bad* which is the set of unsafe states (the complement of *Good*). Assuming that the sets *Init* and *Good* can be defined using linear formulas, the safety verification problem amounts to find an adequate *inductive invariant* I, such that the three following formulas are valid:

$$Init(V) \Rightarrow I(V) \quad (1)$$

$$I(V) \Rightarrow Good(V) \quad (2)$$

$$I(V) \wedge T(V, V') \Rightarrow I(V') \quad (3)$$

We are looking for inductive invariants which can be expressed as a linear formula. In that case, the validity of the three formulas is decidable and can be checked with a standard SMT solver.

2.2 The ICE Learning Framework

ICE [15] follows the active learning paradigm to learn adequate inductive invariants of a given program and a given safety property. It consists of an iteratively communicating *learner* and a *teacher* (see Algorithm 1).

Input : A transition system and a property: $(Init, T, Good)$
Output: An adequate invariant or error
1 initialize ICE-sample $S = (S^+, S^-, S^{\rightarrow})$;
2 **while** *true* **do**
3 $\quad J \leftarrow$ LEARN(S);
4 $\quad (success, counterexample) \leftarrow$ IS_INDUCTIVE(J);
5 $\quad$ **if** *success* **then return** J ;
6 $\quad$ **else**
7 $\quad\quad S \leftarrow$ UPDATE$(S, counterexample)$;
8 $\quad\quad$ **if** *contradictory*(S) **then return** *error*;

Algorithm 1: The main loop of ICE.

In each iteration, in line 3, the *learner*, which does not know anything about the program, synthesizes a candidate invariant (as a formula over the program variables) from a *sample* S (containing information about program states) which is enriched during the learning process. Contrary to other learning methods, the sample S not only contains a set of *positive* states S^+ which should satisfy the invariant, and a set of *negative* states S^- which should not satisfy the invariant,

but it contains also a set of *implications* $S^{\rightarrow}$ of the form $s \rightarrow s'$ meaning that if s satisfies the invariant, then s' should satisfy it as well (because there is a transition from s to s' in the transition relation of the program). Therefore, an ICE-sample S is a triple $(S^+, S^-, S^{\rightarrow})$, where to account for the information contained in implications, it is imposed additionally that

$$\forall s \rightarrow s' \in S^{\rightarrow} : \text{ if } s \in S^+, \text{ then } s' \in S^+, \text{ and if } s' \in S^-, \text{ then } s \in S^- \qquad (4)$$

The sample is initially empty (or containing some states whose status, positive or negative, is known). It is assumed that a candidate invariant J proposed by the learner is *consistent* with the sample, i.e. states in S^+ satisfy the invariant J, the states in S^- falsify it, and for implications $s \rightarrow s' \in S^{\rightarrow}$ it is not the case that s satisfies J but not s'. Given a candidate invariant J provided by the *learner* in line 3, the *teacher* who knows the transition relation T, checks if J is an inductive invariant in line 4; if yes, the process stops, an invariant has been found; otherwise a counterexample is provided and used in line 7 to update the sample for the next iteration. The teacher checks the three conditions an inductive invariant must satisfy (see Sect. 2.1). If (1) is violated the counterexample is a state s which should be in the invariant because it is in *Init*. Therefore s is added to S^+. If (2) is violated the counterexample is a state s which should not be in the invariant because it is not in *Good* and s is added to S^-. If (3) is violated the counterexample is an implication $s \rightarrow s'$ where if s is in the invariant, s' should also be in it. Therefore $s \rightarrow s'$ is added to $S^{\rightarrow}$. In all three cases, the sample is updated to satisfy property 4. If this leads to a contradictory sample, i.e. $S^+ \cap S^- \neq \emptyset$, the program is incorrect and an error is returned. Notice that obviously, in general, the loop is not guaranteed to terminate.

2.3 ICE-DT: Invariant Learning Using Decision Trees

In [16], the ICE learning framework is instantiated with a learn method, which extends classical decision tree learning algorithms with the handling of implications. In the context of invariant synthesis, *decision trees* are used to classify points from a universe, which is the set of program states. They are binary trees whose inner nodes are labeled by predicates from a set of attributes and whose leaves are either $+$ or $-$. Attributes are (atomic) formulas over the variables of the program. They can be seen as boolean functions that the decision tree learning algorithm will compose to construct a classifier of the given ICE sample. In our case of numerical programs manipulating integer variables, attributes are linear inequalities. Then, a decision tree can be seen naturally as a quantifier-free formula over program variables.

The main idea of the ICE-DT learner (see Algorithm 2) is as follows. Initially, the learner fixes a set of attributes (possibly empty) which is kept in a global variable and updated in successive executions of LEARN(S). In line 2, given a sample, the learner checks whether the current set of attributes is sufficient to produce a decision tree corresponding to a formula consistent with the sample. If the check is successful the sample S is changed to S_{Attr} taking

Input : An ICE sample $S = (S^+, S^-, S^{\rightarrow})$
Output: A formula
Global : *Attributes* initialized with *InitialAttributes*
1 **Proc** LEARN(S)
2 $(success, S_{Attr}) \leftarrow$ SUFFICIENT($Attributes, S$);
3 **while** $\neg success$ **do**
4 $Attributes \leftarrow$ GENERATEATTRIBUTES($Attributes, S$);
5 $(success, S_{Attr}) \leftarrow$ SUFFICIENT($Attributes, S$);
6 **return** $tree\_to\_formula$(CONSTRUCT-TREE($S_{Attr}, Attributes$))

Algorithm 2: The ICE-DT learner LEARN(S) procedure.

into account information gathered during the check (see below for the details of SUFFICIENT($Attributes, S$)). If the check fails new attributes are generated with GENERATEATTRIBUTES($Attributes, S$) until success. Then, a decision tree is constructed in line 6 from the sample S_{Attr} by CONSTRUCT-TREE($S_{Attr}, Attributes$) which we present below (Algorithm 3). It is transformed into a formula and returned as a potential invariant. Notice that in the main ICE loop of Algorithm 1 the teacher then checks if this invariant is inductive or not. If not, the original sample S is updated and in the next iteration the learner checks if the attributes are still sufficient for the updated sample. If not, the learner generates new attributes and proceeds with constructing another decision tree and so on.

An important question is how to choose *InitialAttributes* and how to generate new attributes when needed. In [16], the set *InitialAttributes* is for example the set of octagons over program variables with absolute values of constants bounded by $c \in \mathbb{N}$. If these attributes are not sufficient to classify the sample, then new attributes are generated simply by increasing the bound c by 1. We use a different method described in detail in Sect. 4. We now describe how a decision tree can be constructed from an ICE sample and a set of attributes.

Decision Tree Learning Algorithms. The well-known standard decision tree learning algorithms like ID3 [23] take as an input a sample containing points marked as positive or negative of some universe and a fixed set *Attributes*. They construct a decision tree by choosing as the root an attribute, splitting the sample in two (one with all points satisfying the attribute and one with the other points) and recursively constructing trees for the two subsamples. At each step the attribute maximizing the information gain computed using the entropy of subsamples is chosen. Intuitively this means that at each step, the attribute which separates the "best" positive and negative points is chosen. In the context of verification, exact classification is needed, and therefore, all points in a leaf must be classified in a way consistent with the sample.

In [16] this idea is extended to handle also implications which is essential for an ICE learner. The basic algorithm to construct a tree (given as Algorithm 3 below) gets as input an ICE sample $S = (S^+, S^-, S^{\rightarrow})$ and a set of *Attributes* and produces a decision tree *consistent* with the sample, which means that each point in S^+ (resp. S^-) is classified as positive (resp. negative) and for each

implication $(s, s') \in S^{\rightarrow}$ it is not the case that s is classified as positive and s' as negative. The initial sample S is supposed to be consistent.

```
   Input  : An ICE sample S = (S^+, S^-, S^→) and a set of Attributes.
   Output: A tree
 1 Proc ConstructTree(S, Attributes)
 2     Set G (partial mapping of end-points of impl. to {Positive, Negative}) to empty ;
 3     Let Unclass be the set of all end-points of implications in S^→;
 4     Compute the implication closure of G w.r.t. S;
 5     return DecisionTreeICE(⟨S^+, S^-, Unclass⟩, Attributes);
 6 Proc DecisionTreeICE(Examples = ⟨Pos, Neg, Unclass⟩, Attributes)
 7     Move all points of Unclass classified as Positive (resp. Negative) to Pos (resp. Neg);
 8     if Neg = ∅ then
 9         Mark all points of Unclass in G as Positive;
10         Compute the implication closure of G w.r.t. S;
11         return Leaf(+);
12     else if Pos = ∅ then
13         Mark all points of Unclass in G as Negative;
14         Compute the implication closure of G w.r.t. S;
15         return Leaf(−);
16     else
17         a ← choose(Attributes, Examples);
18         Divide Examples into two: Examples_a with all points satisfying a and
             Examples_¬a the others;
19         T_left ← DecisionTreeICE(Examples_a, Attributes \ {a});
20         T_right ← DecisionTreeICE(Examples_¬a, Attributes \ {a});
21         return Tree(a, T_left, T_right);
```

Algorithm 3: The ICE-DT decision-tree learning procedures.

The learner is similar to the classical decision tree learning algorithms. However, it has to take care of implications. To this end, the learner also considers the set of points appearing as end-points in the implications but not in S^+ and S^-. These points are considered in the beginning as unclassified, and the learner will either mark them Positive or Negative during the construction as follows: if in the construction of the tree a subsample is reached containing only positive (resp. negative) points and unclassified points (lines 8 and 12 resp.), *all* these points are classified as positive (resp. negative). To make sure that implications are still consistent, the *implication closure* with the newly classified points is computed and stored in the global variable G, a (partial mapping) of end-points in $S^{\rightarrow}$ to {Positive, Negative}. The implication closure of G w.r.t. S is defined as: If $G(s) =$ Positive or $s \in S^+$ and $(s, s') \in S^{\rightarrow}$ then also $G(s') =$ Positive. If $G(s') =$ Negative or $s' \in S^-$ and $(s, s') \in S^{\rightarrow}$ then also $G(s) =$ Negative.

The set *Attributes* is such that a consistent decision tree will always be found, i.e. the set *Attributes* in line 17 is never empty (see below). An attribute in a node is chosen with choose$(Attributes, Examples)$ returning an attribute $a \in Attributes$ with the highest *gain* according to *Examples*. We do not give the details of this function. In [16] several gain functions are defined extending the classical gain function based on entropy with the treatment of implications. We use the one which penalizes cutting implications (like ICE-DT-penalty).

Checking if the Set of Attributes is Sufficient. Here we show how the function sufficient$(Attributes, S)$ of Algorithm 2 is implemented in [16]. Two

states s and s' are considered equivalent (denoted by $\equiv_{Attributes}$), if they satisfy the same attributes of *Attributes*. One has to make sure that two equivalent states are never classified in different ways by the tree construction algorithm. This is done by the following procedure: For any two states s, s' with $s \equiv_{Attributes} s'$ which appear in the sample (as positive or negative or end-points of the implications) two implications $s \rightarrow s'$ and $s' \rightarrow s$ are added to $S^{\rightarrow}$ of S.

Then, the implication closure of the sample is computed starting from an empty mapping G (all end-points are initially unclassified). If during the computation of the implication closure one end-point is classified as both POSITIVE and NEGATIVE, then SUFFICIENT$(Attributes, S)$ returns $(false, S)$ else it returns $(true, S_{Attr})$ where S_{Attr} is obtained from $S = (S^+, S^-, S^{\rightarrow})$ by adding to S^+ the end-points of implications classified as POSITIVE and to S^- the end-points classified as NEGATIVE.

In [16] it is shown that this guarantees in general that a tree consistent with the sample will always be constructed regardless of the order in which attributes are chosen. We illustrate now the ICE-DT learner on a simple example.

Example 1. Let $S = (S^+, S^-, S^{\rightarrow})$ be a sample (illustrated in Fig. 1) with two-dimensional states (variables x and y): $S^+ = \{(1,1), (1,4), (3,1), (5,1), (5,4), (6,1), (6,4)\}$, $S^- = \{(4,1), (4,2), (4,3), (4,4)\}$, $S^{\rightarrow} = \{(2,2) \rightarrow (2,3), (0,2) \rightarrow (4,0)\}$. We suppose that $Attributes = \{x \geq 1, x \leq 3, y \geq 1, y \leq 4, x \geq 5, x \leq 6\}$ is given. In Sect. 4 we show how to obtain this set from the sample. The learner first checks that the set *Attributes* is sufficient to construct a formula consistent with S. The check succeeds and we have among others that $(2,2)$ and $(2,3)$ and the surrounding positive states on the left are all equivalent w.r.t. $\equiv_{Attributes}$. Therefore after adding implications (which we omit for clarity in the following) and the computation of the implication closure both $(2,2)$ and $(2,3)$ are added to S^+. Then, the construction of the tree is started with *Examples* containing 9 positive, 4 negative and 2 unclassified states. Depending on the gain function an attribute is chosen. Here, it is $x \geq 5$, since it separates all the positive states on the right from the rest and does not cut the implication. The set *Examples* is split into the states satisfying $x \geq 5$ and those which don't: $Examples_{x \geq 5}$ and $Examples_{x<5}$. $Examples_{x \geq 5}$ contains only positive states $\{(5,1), (5,4), (6,1), (6,4)\}$ and the branch is finished whereas $Examples_{x<5}$ contains the remaining positive, negative and unclassified states and the construction continues. The attribute $x \leq 3$ is chosen and $Examples_{x<5}$ split in two. $Examples_{x<5 \wedge x \leq 3}$ contains the positive states $\{(1,1), (1,4), (3,1), (2,2), (2,3)\}$ and one unclassified state $(0,2)$. Therefore, the algorithm marks $(0,2)$ as positive and as there is an implication $(0,2) \rightarrow (4,0)$, the state $(4,0)$ is marked positive as well and a leaf node is returned. The other branch $Examples_{x<5 \wedge x>3}$ now contains negative states $\{(4,1), (4,2), (4,3), (4,4)\}$ and a positive state $(4,0)$. Therefore another attribute is needed. Finally, the algorithm returns a tree corresponding to the formula $x \geq 5 \vee (x < 5 \wedge x \leq 3) \vee (x < 5 \wedge x > 3 \wedge y < 1)$.

3 Linear Formulas as Abstract Objects

Algorithm 2 requires a set of attributes as input. In Sect. 4, we show how to generate these attributes from the sample. For that purpose, we use numerical abstract domains to represent and manipulate algorithmically sets of integer vectors representing program states. We consider standard numerical domains defined in [7,9,22] and implemented in tools such as APRON [18]: Intervals, Octagons, and Polyhedra.

Given a set of n variables X and a linear formula φ over X, let $[\![\varphi]\!] \subseteq \mathbb{Z}^n$ be the set of all integer points satisfying the formula. Now, a subset of $\mathbb{Z}^n$ is called

- an *interval*, iff it is equal to $[\![\varphi]\!]$ where φ is a conjunction of constraints of the form $\alpha \leq x \leq \beta$, where $x \in X$, $\alpha \in \mathbb{Z} \cup \{-\infty\}$ and $\beta \in \mathbb{Z} \cup \{+\infty\}$.
- an *octagon*, iff it is equal to $[\![\varphi]\!]$ where φ is a conjunction of constraints of the form $\pm x \pm y \leq \alpha$ where $x, y \in X$ and $\alpha \in \mathbb{Z} \cup \{+\infty\}$.
- a *polyhedra*, iff it is equal to $[\![\varphi]\!]$ where φ is a conjunction of linear constraints of the form $\sum_{i=1}^{n} a_i x_i \leq b$ where $X = \{x_1, \ldots, x_n\}$ and for every i, $a_i \in \mathbb{Z}$, and $b \in \mathbb{Z} \cup \{+\infty\}$.

Now, we can define several abstract domains as complete lattices $A_X^{type} = \langle D_X^{type}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$, where *type* is either *int*, *oct* or *poly* and D_X^{int} is the set of intervals, D_X^{oct} is the set of octagons and D_X^{poly} the set of polyhedra.

The relation $\sqsubseteq$ is set inclusion. The binary operation $\sqcup$ (resp. $\sqcap$) is the *join* (resp. *meet*) operation that defines the smallest (resp. greatest) element in D_X that contains (resp. contained in) the union (resp. the intersection) of the two composed elements. Finally $\bot$ (resp. $\top$) corresponds to the empty set (resp. $\mathbb{Z}^n$).

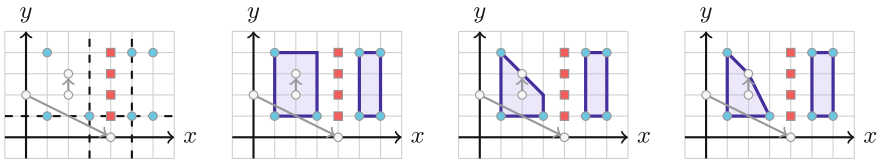
We suppose that we have a function $Form^{type}(d)$ which given an element $d \subseteq \mathbb{Z}^n$ of the lattice provides us a formula φ of the corresponding type such that $[\![\varphi]\!] = d$. There are many ways to describe the set d with a formula φ. Therefore the function $Form^{type}(d)$ depends on the particular implementation of the abstract domains. We furthermore define $Constr^{type}(d)$ to be the set of linear constraints of $Form^{type}(d)$.

We drop the superscript *type* from all preceding definitions, when it is clear from the context or when we define notions for all types.

All singleton subsets of $\mathbb{Z}^n$ are elements of the lattices and for example, if $p = (x = 1, y = 2)$, then, for the domains of Intervals, Octagons, and Polyhedra as implemented in APRON we have: $Constr^{int}(\{p\}) = \{x \leq 1, x \geq 1, y \leq 2, y \geq 2\}$, $Constr^{oct}(\{p\}) = \{x \geq 1, x \leq 1, y - x \geq 1, x + y \geq 3, y \geq 2, y \leq 2, x + y \leq 3, x - y \geq -1\}$ and $Constr^{poly}(\{p\}) = \{x = 1, y = 2\}$.

Notice, that in APRON while equality constraints are used in the Polyhedra domain, these constraints are not explicit in the Interval and Octagon domains.

An important fact about the three domains mentioned above is that, each element of the lattice is the intersection of a convex subset of $\mathbb{Q}^n$ with $\mathbb{Z}^n$. To be able to reason about integer points from *nonconvex* sets, we will use in the next section sets of sets.



(a) An ICE sample (b) Intervals (`int`) (c) Octagons (`oct`) (d) Polyhedra (`poly`)

Fig. 1. An ICE sample and its separators using different abstract domains.

4 Generating Attributes from Sample Separators

We define in this section algorithms for generating a set of attributes that can be used for constructing decision trees representing candidate invariants. Given an ICE sample, these algorithms are based on constructing separators of the two sets of positive and negative states that are consistent with the implications in the sample. These separators are sets of intervals, octagons or polyhedra. The set of all constraints that define these sets are collected as a set of attributes.

4.1 Abstract Sample Separators

Let $S = (S^+, S^-, S^{\rightarrow})$ be an ICE sample, and let $A_X = \langle D_X, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ be an abstract domain. Intuitively, a separator has sets containing all positive states, not containing any negative state and is consistent with implications. Formally, an A_X-separator of S is a set $\mathbb{S} \in 2^{D_X}$ such that $\forall p \in S^+.\ \exists d \in \mathbb{S}.\ p \in d$ and $\forall p \in S^-.\ \forall d \in \mathbb{S}.\ p \notin d$ and $\forall p \rightarrow q \in S^{\rightarrow}.\ \forall d \in \mathbb{S}.\ (p \in d \implies (\exists d' \in \mathbb{S}.\ q \in d'))$.

Given a set of positive states S^+, we define the basic separator $\mathbb{S}_{basic}$ as $\{\{p\} \mid p \in S^+\}$ where each state is alone in its set. Our method for generating attributes for the learning process is based on computing a special type of separators called *join-maximal*. An A_X-separator $\mathbb{S}$ is join-maximal if is not possible to take the join of two of its elements without including a negative state: $\forall d_1, d_2 \in \mathbb{S}.\ d_1 \neq d_2 \implies (\exists n \in S^-.\ n \in d_1 \sqcup d_2)$.

Example 2. Let us consider again the ICE sample S given in Example 1. Figure 1 shows the borders of join-maximal A_X-separators for S for different abstract domains (Intervals `int`, Octagons `oct`, and Polyhedra `poly`).

Remark 1. An ICE sample may have multiple join-maximal separators as Fig. 2 shows for the polyhedra domain. The method presented in the next section computes one of them non-deterministically.

4.2 Computing a Join-Maximal Abstract Separator

We present in this section a basic algorithm for computing a join-maximal A_X-separator for a given sample S. Computing such a separator can be done iteratively starting from $\mathbb{S}_{basic}$, and at each step, choosing two elements d_1 and d_2

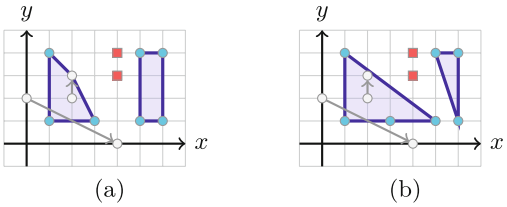


Fig. 2. Different join-maximal separators for a same sample.

in the current separator such that $d_1 \sqcup d_2$ does not contain a negative state in S^- (This can be checked using the meet operation $\sqcap$), and replacing d_1 and d_2 by $d_1 \sqcup d_2$. Then, if any element of the separator contains the source p of an implication $p \to q$, which means that p is considered now as a positive state, then since q must also be considered as positive, the element $\{q\}$ must be added to the separator if q is not already in some element of the current separator. When no new join operations (without including negative states) can be done, the obtained set is necessarily a join-maximal A_X-separator of S. This procedure corresponds to Algorithm 4.

Input : An ICE sample $S = (S^+, S^-, S^{\to})$ and an abstract domain $A_X = \langle D_X, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$.
Output: $\mathbb{S}$ a join-maximal A_X-separator of S.

1 **Proc** CONSTRUCTSEPARATOR(S, A_X)
2 $\quad \mathbb{S} \leftarrow \mathbb{S}_{basic}$ (* $= \{\{s\} \mid s \in S^+\}$ *) ;
3 $\quad$ **while** *true* **do**
4 $\quad\quad$ **if** $\exists a, b \in \mathbb{S}.\, a \neq b \wedge \forall n \in S^-.\, n \notin a \sqcup b$ **then**
5 $\quad\quad\quad \mathbb{S} \leftarrow (\mathbb{S} \setminus \{a, b\}) \cup \{a \sqcup b\}$;
6 $\quad\quad\quad$ **while** $\exists p \to q \in S^{\to}.\, \exists d \in \mathbb{S}.\, p \in d \wedge \forall d' \in \mathbb{S}.\, q \notin d'$ **do**
7 $\quad\quad\quad\quad \mathbb{S} \leftarrow \mathbb{S} \cup \{\{q\}\}$;
8 $\quad\quad$ **else break**;

Algorithm 4: Computing a join-maximal A_X-separator.

Notice that instead of starting with the basic separator $\mathbb{S}_{basic}$ defined as above one can start with any separator $\mathbb{S}_{init} \supseteq \mathbb{S}_{basic}$ whose additional sets contain only states which are known to be positive (for example the initial states).

Example 3. Consider again the sample S of Example 2. We show how the separators of S in Fig. 1 are constructed using Algorithm 4. The algorithm starts from the basic separator $\mathbb{S}_{basic}$ where every positive state in S is alone (Fig. 3(a)). It picks two elements in that separator, e.g. $\{d_1\}$ and $\{d_2\}$. As their join does not include negative states, $\{d_1\}$ and $\{d_2\}$ are replaced by $j_1 = \{d_1\} \sqcup \{d_2\}$ to get a new separator (Fig. 3(b)). Then, depending on the considered domain, different separators are obtained. For Intervals, the join of j_1 and $\{d_3\}$ leads to the separator in Fig. 1(a). Notice that both ends of the implication $(2, 2) \to (2, 3)$ are included in $j_1 \sqcup \{d_3\}$. In the case of Octagons, the join of j_1 and $\{d_3\}$ is the set

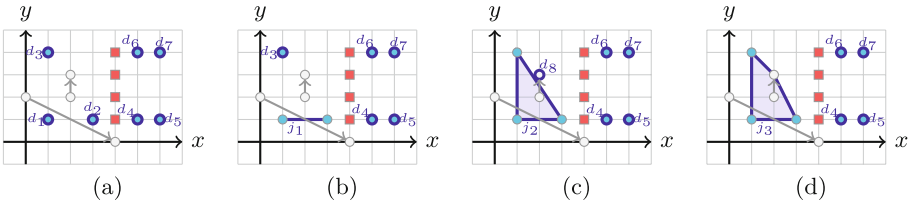


Fig. 3. The first iterations of Algorithm 4 on the sample S of Fig. 1

on the left of Fig. 1(b). Again, both ends of the implication $(2,2) \rightarrow (2,3)$ are included in $j_1 \sqcup \{d_3\}$. In the case of Polyhedra, $j_2 = j_1 \sqcup \{d_3\}$ is the triangle shown in Fig. 3(c). Since $(2,2)$ is included in j_2 but not $(2,3)$, the element $\{(2,3)\}$ is added to the separator, leading to the separator represented in Fig. 3(c). In the next iteration, j_2 is joined with $\{d_8\}$ leading to the separator shown in Fig. 3(d). Finally, a similar iteration of join operations leads to the rectangle including the four points, and this leads to the join-maximal separator of Fig. 1.

Remark 2. In the best case Algorithm 4 performs $|S^+|$ join and $|S^+|(|S^-|+|S^{\rightarrow}|)$ meet operations (all pairs of points can be joined and all left end-points of implications are not in the new joined convex sets). In the worst case, it performs $O\big((|S^+| + |S^{\rightarrow}|)^2\big)$ join and $O\big((|S^+| + |S^{\rightarrow}|)^2(|S^-| + |S^{\rightarrow}|)\big)$ meet operations (at most $|S^-| + |S^{\rightarrow}|$ meets are needed to check if two sets can be joined and implications might add new points to S^+). The cost of meet and join depends on the used abstract domain; it is polynomial for intervals and octagons, and exponential for polyhedra, in the number of variables. Algorithm 4 is not designed to compute a join-max separator with a minimal number of convex sets as this would require a potentially exponential number of meet and join operations.

4.3 Integrating Separator Computation in ICE-DT

We use the computation of a join-maximal separator to provide an instance of the function GENERATEATTRIBUTES of ICE-DT in Algorithm 2. Given a sample S, let $\mathbb{S}$ be the A_X-separator of S computed by CONSTRUCTSEPARATOR(S, A_X) defined by Algorithm 4. We consider the set *InitialAttributes* containing all the predicates that constitute the specification (*Init* and *Good*) and those that appear in the programs (as tests in the conditional statements and while loops). Then, we define: GENERATEATTRIBUTES(S) = *InitialAttributes* $\cup$ $\bigcup_{d \in \mathbb{S}}$ *Constr*(d)

Remark 3. Several convex sets of the separator $\mathbb{S}$ might generate the same constraint and the set of attributes generated in this way might contain attributes which partition the state space in the same way (e.g. $x \leq 0$ and $x \geq 1$, equivalent to $x > 0$ over the integers). We keep only one of them. The number of attributes generated is at most linear in the number of positive states in the sample S.

```
1 int j, k, t;
2 assume(j = 2 ∧ k = 0);
3 while true do
4 |  if t = 0 then j ← j + 4 else j ← j + 2; k ← k + 1;
5 assert(k = 0 ∨ j = 2k + 2);
```

Fig. 4. Example program

Notice that our function GENERATEATTRIBUTES(S), contrary to the one used in the original ICE-DT (Algorithm 2), does not expand a set of existing attributes, and therefore it only need the sample S as argument. In fact, with our method for computing attributes, the ICE-DT schema can be simplified: the while loop in Algorithm 2 can be replaced by one single initial test on the condition of success. Indeed, each time the learner is called, it checks whether the set of attributes computed for the previous sample is sufficient to build a separator for the new sample. Only when it is not sufficient that the generation of a separator is performed. Then, the call of the SUFFICIENT function afterward is needed to extend the sample so that the construction of a decision tree can be done (see explanation in Sect. 2.3), but it will necessarily succeed since in our case the set of attributes defines by construction a separator of the sample.

Example 4. Consider the program in Fig. 4 whose set of variables is $X = \{j, k, t\}$. We use Polyhedra. First, starting from an empty ICE-sample, regardless of the attributes, the learner proposes *true* as an invariant and $(5, 1, 0)$ is returned as a negative counterexample. Then, it proposes $false$ and $(2, 0, 0)$ is returned as a positive counterexample.

Now, Algorithm 4 is called to compute a separator for $S = (S^+ = \{(2, 0, 0)\}, S^- = \{(5, 1, 0)\}, S^{\rightarrow} = \emptyset)$. Here, we use initially a separator $\mathbb{S}_{init}$ containing the set of states satisfying the initial condition $j = 2 \wedge k = 0$ denoted by d_1 in addition to d_0 where $d_0 = \{(2, 0, 0)\}$. Since $d_0 \subseteq d_1$, the algorithm returns the join-maximal separator $\mathbb{S} = \{d_1\}$ with $Constr^{poly}(d_1) = \{j = 2, k = 0\}$.

Using constraints from $\mathbb{S}$ as attributes, the learner constructs the candidate invariant $k = 0$. Then, the teacher provides an implication counterexample $(0, 0, 1) \rightarrow (2, 1, 1)$. Now, without computing another separator (as the one it has is sufficient for the new sample), the learner proposes $j = 2 \wedge k = 0$ as an invariant, and the implication counterexample $(2, 0, 1) \rightarrow (4, 1, 1)$ is returned (and since $(2, 0, 1)$ is an initial state, $(4, 1, 1)$ is also considered positive).

Then, Algorithm 4 is called again to construct a separator for the sample $S = (S^+ = \{(2, 0, 0), (4, 1, 1)\}, S^- = \{(5, 1, 0)\}, S^{\rightarrow} = \{(0, 0, 1) \rightarrow (2, 1, 1), (2, 0, 1) \rightarrow (4, 1, 1)\})$. Starting from a separator $\mathbb{S}_{init} = \{d_0, d_1, d_2\}$ with $d_2 = \{(4, 1, 1)\}$ it returns the join-maximal separator

$$\mathbb{S} = \{d_3\} \qquad Constr^{poly}(d_3) = \{2k + 2 = j, j \leq 4, j \geq 2\}$$

Based on this separator, the learner proposes $2k+2 = j$, $(2, 0, 0) \rightarrow (6, 0, 0)$ is given as a counterexample (and then, since $(2, 0, 0)$ is in S^+, $(6, 0, 0)$ is considered

positive). Then, from $\mathbb{S}_{init} = \{d_0, d_1, d_2, d_4\}$ with $d_4 = \{(6,0,0)\}$ a new separator $\mathbb{S}$ is constructed

$$\mathbb{S} = \{d_5\} \qquad Constr^{poly}(d_5) = \{j + 2k \leq 6, k \geq 0, j \geq 2k + 2\}$$

leading to a new candidate invariant: $j + 2k \leq 6 \wedge j \geq 2k + 2$. The teacher returns at this point the negative state $(0, -2, 0)$. The attributes of $\mathbb{S}$ are still sufficient to construct a decision tree for the sample. Then, the learner proposes $j + 2k \leq 6 \wedge k \geq 0 \wedge j \geq 2k + 2$, and the teacher returns the counterexample $(3,0,1) \rightarrow (5,1,1)$ (and since $(5,1,1)$ is a negative state, $(3,0,1)$ is considered negative). The current sample S is now ($S^+ = \{(2,0,0),(4,1,1),(6,0,0)\}$, $S^- = \{(5,1,0),(5,1,1),(3,0,1),(0,-2,0)\}$, $S^{\rightarrow} = \{(0,0,1) \rightarrow (2,1,1),\ (2,0,1) \rightarrow (4,1,1),\ (2,0,0) \rightarrow (6,0,0),(3,0,1) \rightarrow (5,1,1)\}$).

Then, from $\mathbb{S}_{init} = \{d_0, d_1, d_2, d_4\}$, a join-maximal separator is constructed

$$\mathbb{S} = \{d_3, d_4\} \qquad Constr^{poly}(d_4) = \{j = 6, t = 0, k = 0\}$$

Some iterations later, using only the attributes of the last $\mathbb{S}$, the learner generates the inductive invariant $(t = 0 \wedge 2 \leq j \wedge k = 0) \vee (t \neq 0 \wedge 2 \leq j \wedge 2k + 2 = j)$

4.4 Computing Separators Incrementally

Algorithm 4 of Sect. 4.2 always starts from the initial separator, regardless of what has been done in the previous iterations of the ICE learning process. Here, we present an incremental approach to exploit the fact that adding a counterexample to the sample may modify the separator only locally allowing parts of separators computed in previous iterations to be reused. The basic idea is to store the history of the separator computation along the ICE iterations, and update it according to the new counterexamples discovered at each step.

The Algorithm. We use an abstract stack data structure to represent the history of separators. Along the iterations of the ICE learning algorithm, an increasing sequence of samples S_i's is considered (at each iteration it is enriched by the new counterexample provided by the teacher). Then, at each step i, a join-maximal separator $\mathbb{S}_i$ of the sample S_i is computed and stored in the stack. Notice that at a given step i, separators of index $j < i$ are not necessarily separators of S_i since they may not cover all positive points of S_i. Therefore, we introduce the following notion: a *partial A_X-separator* of a sample S is a set $\mathbb{S} \in 2^{D_X}$ such that $\forall p \in S^-. \forall d \in \mathbb{S}. p \notin d$.

Now, to compute the separator $\mathbb{S}_i$, we start from one of the partial separators in the stack, the most recent one that is not affected by the last update of the sample. When the sample at step i is extended with positive states, $\mathbb{S}_i$ can be computed directly from $\mathbb{S}_{i-1}$. However, when the sample is extended with negative states, this might require reconsidering several previous steps since some of the elements (convex sets) of their separators might contain states that are (discovered now to be) negative. In that case, we must return to the step of the greatest index $j < i$ (i.e., the last step before i) such that $\mathbb{S}_j$ is a partial

separator of S_i (i.e., the new knowledge about the negative states does not affect the computed separation at step j). By the fact that the sequence of samples is increasing, it is indeed correct to consider the biggest $j < i$ satisfying the property above. Therefore, the separator $\mathbb{S}_i$ is computed starting from $\mathbb{S}_j$ augmented with all the positive states in $S_i^+ \setminus S_j^+$.

This leads to Algorithm 5. We use in its description a stack P supplied with the usual operations: $P.head()$ returns the top element of the stack, $P.\text{pop}()$ removes and returns the top element of the stack, and $P.\text{push}(e)$ inserts an element e at the top position of the stack. A refined version of Algorithm 5 is presented in the full paper [2] where the backtracking phase is made more effective: We attach information to each join-created object in order to track its join-predecessors (objects involved in its creation) in the stack.

```
   Global : P = {∅} a stack of partial separators.
1  Proc constructSeparatorInc(S_i = (S_i^+, S_i^-, S_i^→), A_X)
       // backtracking
2      while true do
3          if ∃n ∈ S_i^-. ∃d ∈ P.head(). n ∈ d then
4              P.pop();
5          else break;
       // expansion
6      𝕊 ← P.head();
7      add ← {p ∈ S_i^+ | ∀d ∈ 𝕊. p ∉ d} ∪ {q | ∃p → q ∈ S_i^→. ∃d' ∈ 𝕊. p ∈ d' ∧ ∀d'' ∈ 𝕊. q ∉ d''};
8      while ∃s ∈ add do
9          add ← add \ {s};
10         if ∃d ∈ 𝕊. ∀n ∈ S_i^-. n ∉ d ⊔ {s} then
11             let o = d ⊔ {s};
12             𝕊 ← (𝕊 \ {d}) ∪ {o};
13             for p → q ∈ S_i^→ s.t. p ∈ o ∧ ∀d' ∈ 𝕊. q ∉ d' do
14                 add ← add ∪ {q}
15         else
16             𝕊 ← 𝕊 ∪ {{s}};
17             for p → q ∈ S_i^→ s.t. p = s ∧ ∀d' ∈ 𝕊. q ∉ d' do
18                 add ← add ∪ {q}
19     P.push(𝕊);
20     return 𝕊;
```

Algorithm 5: Incremental computation of an A_X-separator of a sample S.

Integration to ICE-DT. The function constructSeparatorInc can be integrated to the ICE-DT algorithm just as the function constructSeparator in Sect. 4.3, by using it to implement the function *generateAttributes* of the learner. But this time, the learner is more efficient in computing the separator from which the attributes are extracted.

Example 5. Consider again the program in Fig. 4 of Example 4. The two first iterations are similar to the ones described in Example 4. Then, the obtained sample is $S = (S^+ = \{(2,0,0)\}, S^- = \{(5,1,0)\}, S^{\rightarrow} = \emptyset)$. Starting from the empty separator, Algorithm 5 computes the separator $\mathbb{S}_1 = \{d_1\}$ where $Constr^{poly}(d_1) = \{j = 2, k = 0\}$. Then, the learner proceeds as in the previous example to get the sample $S = (S^+ = \{(2,0,0),(4,1,1)\}, S^+ = \{(5,1,0)\}, S^{\rightarrow} = \{(0,0,1) \rightarrow (2,1,1), (2,0,1) \rightarrow (4,1,1)\})$. To build a separator of S, Algorithm 5 starts from $\mathbb{S}_1$ and produces $\mathbb{S}_2 = \{d_3\}$ where $d_3 = d_1 \sqcup \{(4,1,1)\}$.

Tool	Solved		Total
	safe	unsafe	
ICE-DT	111	11	122
LoopInvGen	130	8	138
CVC4	129	-	129
Spacer	118	18	136
NIS(`int`)	106	17	123
NIS(`oct`)	122	14	136
NIS(`poly`)	137	17	154
NIS(VB)	143	17	160

	NIS(`int`)	NIS(`oct`)	NIS(`poly`)
NIS(`int`)	-	7	2
NIS(`oct`)	13	-	5
NIS(`poly`)	31	18	-

Fig. 5. Benchmark results and comparison of NIS wrt. different abstract domains.

Similarly, when the counterexample $(2,0,0) \rightarrow (6,0,0)$ is obtained, the algorithm starts directly from $\mathbb{S}_2$ to produce $\mathbb{S}_3 = \{d_5\}$ where $d_5 = d_3 \sqcup \{(6,0,0)\}$.

After two more iterations, the sample is the same as S' in Example 4. At this point, $\mathbb{S}_3$ cannot be used to construct a separator for S since d_5 includes the negative state $(3,0,1)$. Then, the algorithm removes $\mathbb{S}_3$ from the stack. It checks that $\mathbb{S}_2$ is a partial separator of S, which is indeed the case. Then, it constructs a new separator $\mathbb{S}_4$ based on $\mathbb{S}_2$ by expanding it with the counterexamples received after the construction of $\mathbb{S}_2$ (the negative state $(0,-2,0)$ and the implications $(2,0,0) \rightarrow (6,0,0)$ and $(3,0,1) \rightarrow (5,1,1)$): $\mathbb{S}_4 = \{d_3, d_6\}$ where $Constr^{poly}(d_6) = \{t = 0, k = 0, j = 6\}$. The rest of the execution proceeds as with Algorithm 4. Here, the advantages of the incremental method are: (1) while positive examples are added the separators are simply expanded, and (2) when a negative example at step 4 is added, only one join operation has to be undone.

5 Experiments

We have implemented the prototype tool NIS (Numerical Invariant Synthesizer) using our method for attribute synthesis with the ICE-DT schema. NIS written in C++ is configurable with an abstract domain for the manipulation of abstract objects. It uses Z3 [24] for SMT queries and APRON's [18] abstract domains.

We compare our implementation with ICE-DT[1], LoopInvGen, CVC4, and Spacer[2]. LoopInvGen is a data-driven invariant inference tool based on a syntactic enumeration of candidate predicates [25,26]. It is written in OCaml and uses Z3 as an SMT solver. CVC4 uses an enumerative refutation-based approach [1,27]. It is written in C++ and it includes an SMT solver. Spacer is a PDR-based CHC solver [19], written in C++ and integrated in Z3.

[1] The original ICE-DT tool [16] does not support programs in the SyGuS format. Here we use our own implementation of ICE-DT. It shares with NIS all the components (teacher, decision tree learning algorithm with implications) except that attribute discovery is enumerative.

[2] Spacer does not support programs in the SyGuS format; a wrapper is written in C++ that converts a SyGuS program to a CHC problem and supplies it to Spacer via the Z3 FixedPoint API.

The evaluation was done on 164 linear integer arithmetic (LIA) programs[3] from SyGuS-Comp'19. They have a number of variables ranging from 2 to 10. The experiments were carried out using a timeout of 1800 s (30 min) for each example. They were conducted on a machine with 4 CPUs Intel(R) Xeon(R) 2,13 GHz, 16 cores, and 128 Go RAM running Linux CentOS 7.9.

Figure 5 shows the number of safe and unsafe solved programs by each tool. The instance of our approach using the Polyhedral abstract domain solves 154 programs out of 164, and the virtual best of our approach with the three abstract domains Intervals, Octagons, and Polyhedra, solves 160 programs out of 164. Two of the remaining examples require handling quantifiers, which cannot be done with the current implementation. The two others have not been solved with any of the four tools we considered.

These results show that globally our approach is powerful and is able to solve a significant number of cases that are not solvable by other tools. Interestingly, using different abstract domains leads to incomparable performances: although with polyhedra more cases are solvable, there are some cases that are uniquely solvable with intervals or octagons. Also, while operations on intervals and octagons have a lower complexity than on polyhedra, this is compensated with the fact that polyhedra are more expressive. Indeed, their expressiveness allows in many cases to find quickly invariants for which a less expressive domain requires much more iterations to be learned. Figure 5 shows the number of programs that can be solved using a particular abstract domain but not with another. Polyhedra are globally superior, but the three domains are complementary.

Compared to the other tools, the bottleneck of ICE-DT and also of LoopInvGen is the number of predicates that are generated using enumeration. Our approach avoids the explosion of the size of the attribute pool by guiding their discovery with the data sample, and reducing the size (by replacing objects by their join) of the computed separators from which constraints are extracted. Concerning CVC4, it uses enumerative refutation techniques, which are also subject to an explosion problem. Moreover, CVC4 does not allow to solve the cases of unsafe programs. The performances of Spacer depend on the ability to generalize the set of predecessors computed using the model-based projection and the interpolants used for separation from bad states in the context of IC3/PDR. While this is done efficiently in general, there are cases where this process can lead to fastidious computations while our technique can be much faster using a small number of join operations of positive states.

The scatter plots shown in Fig. 6 compare the execution times of our approach using Polyhedra abstract domain NIS(`poly`) with LoopInvGen, CVC4 and Spacer. (A timeout of 1800 s s is used for each example.) They show that NIS(`poly`) is in general faster than both LoopInvGen and CVC4, and that it has comparable performances in terms of execution time with Spacer. We have

[3] Other programs from SyGuS-Comp'19 have not been taken into account in our evaluations as they are boolean programs with integer variables for encoding nondeterminism or artificial programs augmented with useless variables and statements.

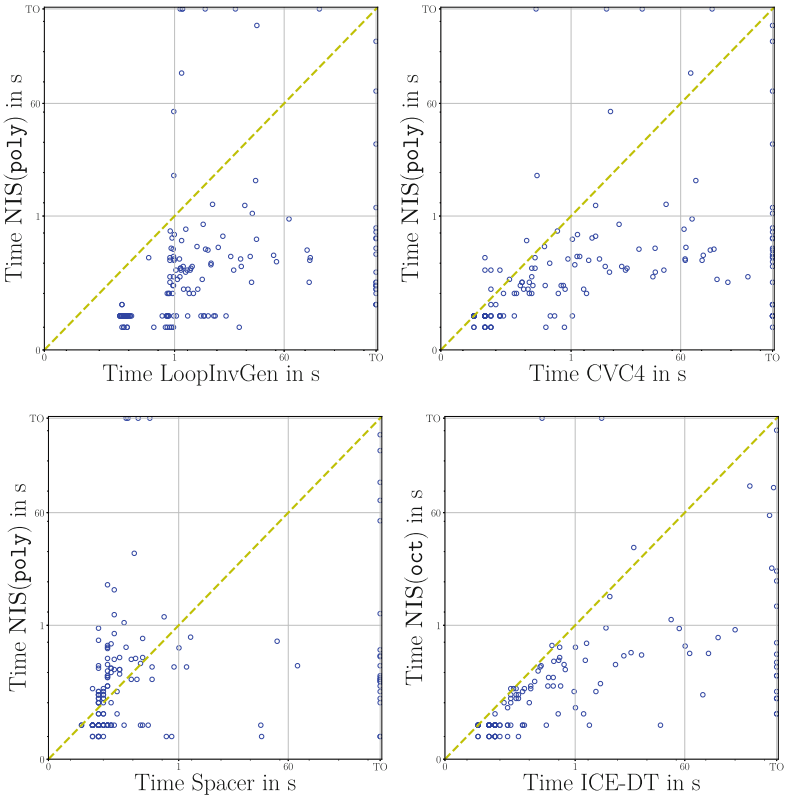


Fig. 6. Runtime of NIS(`poly`) vs. LoopInvGen, CVC4, and Spacer, and NIS(`oct`) vs. ICE-DT.

also compared the original ICE-DT, based on enumerative attribute generation using octagonal templates (as in [16]) with NIS(`oct`). The comparison shows that our tool is significantly faster (see the bottom right subfigure of Fig. 6).

6 Conclusion

We have defined an efficient method for generating relevant predicates for the learning process of numerical invariants. The approach is guided by the data sample built during the process and is based on constructing a separator of the sample. The construction consists of an iterative application of join operations in numerical abstract domains in order to cover positive states without including negative ones. Our method is tightly integrated to the ICE-DT schema, leading to an efficient data-driven invariant synthesis and verification algorithm.

Future work includes several directions. First, alternative methods for constructing separators should be investigated in order to reduce the size of the pool of attributes along the learning process while increasing their potential relevance.

Another issue to investigate is the control of the counterexamples provided by the teacher since they play an important role in the learning process. In our current implementation, their choice is totally dependent on the SMT solver used for implementing the teacher. Finally, we intend to extend this approach to other types of programs, in particular to programs with other data types, and programs with more general control structures such as procedural programs.

References

1. 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
2. Bouajjani, A., Boutglay, W.A., Habermehl, P.: Data-driven numerical invariant synthesis with automatic generation of attributes (2022). https://doi.org/10.48550/ARXIV.2205.14943, https://arxiv.org/abs/2205.14943
3. 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
4. Champion, A., Kobayashi, N., Sato, R.: HoIce: an ICE-based non-linear horn clause solver. In: Ryu, S. (ed.) APLAS 2018. LNCS, vol. 11275, pp. 146–156. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02768-1_8
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
6. 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
7. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming, pp. 106–130. Dunod (1976)
8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, pp. 238–252. ACM (1977)
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL 1978, pp. 84–96. ACM Press (1978)
10. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: FMCAD 2011, pp. 125–134. FMCAD Inc. (2011)
11. Ernst, M.D., et al.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
12. Ezudheen, P., Neider, D., D'Souza, D., Garg, P., Madhusudan, P.: Horn-ICE learning for synthesizing invariants and contracts. Proc. ACM Program. Lang. **2**(OOPSLA), 131:1-13125 (2018)
13. Feldman, Y.M.Y., Immerman, N., Sagiv, M., Shoham, S.: Complexity and information in invariant inference. Proc. ACM Program. Lang. **4**(POPL), 51–529 (2020)
14. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45251-6_29

15. 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
16. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: POPL 2016, pp. 499–512. ACM (2016)
17. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_13
18. Jeannet, B., Miné, A.: Apron: a library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_52
19. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 17–34. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_2
20. Li, J., Sun, J., Li, L., Le, Q.L., Lin, S.: Automatic loop-invariant generation and refinement through selective sampling. In: ASE 2017. pp. 782–792. IEEE Computer Society (2017)
21. Löding, C., Madhusudan, P., Neider, D.: Abstract learning frameworks for synthesis. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 167–185. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_10
22. Miné, A.: The octagon abstract domain. High. Order Symb. Comput. **19**(1), 31–100 (2006)
23. Mitchell, T.M.: Machine Learning, International Edition. McGraw-Hill Series in Computer Science, McGraw-Hill (1997)
24. 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
25. Padhi, S., Millstein, T., Nori, A., Sharma, R.: Overfitting in synthesis: theory and practice. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 315–334. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_17
26. Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: PLDI 2016, pp. 42–56. ACM (2016)
27. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C., Tinelli, C.: cvc4sy: smart and fast term enumeration for syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 74–83. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_5
28. Ryan, G., Wong, J., Yao, J., Gu, R., Jana, S.: CLN2INV: learning loop invariants with continuous logic networks. In: ICLR 2020. OpenReview.net (2020)
29. 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
30. 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
31. Si, X., Naik, A., Dai, H., Naik, M., Song, L.: Code2Inv: a deep learning framework for program verification. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 151–164. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_9

32. 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
33. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. In: PLDI 2018. pp. 707–721. ACM (2018)