Property-Directed Inference of Universal Invariants or Proving Their Absence

A. Karbyshev[1], N. Bjørner[2], S. Itzhaky[3], N. Rinetzky[1], and S. Shoham[4]

[1] Tel Aviv University, Tel Aviv, Israel
[2] Microsoft Research, USA
[3] Massachusetts Institute of Technology, USA
[4] The Academic College of Tel Aviv Yaffo

Abstract. We present *Universal Property Directed Reachability* ($\mathsf{PDR}^\forall$), a property-directed procedure for automatic inference of invariants in a universal fragment of first-order logic. $\mathsf{PDR}^\forall$ is an extension of Bradley's PDR/IC3 algorithm for inference of propositional invariants. $\mathsf{PDR}^\forall$ terminates when it either discovers a concrete counterexample, infers an inductive universal invariant strong enough to establish the desired safety property, or finds a *proof that such an invariant does not exist*. We implemented an analyzer based on $\mathsf{PDR}^\forall$, and applied it to a collection of list-manipulating programs. Our analyzer was able to automatically infer universal invariants strong enough to establish memory safety and certain functional correctness properties, show the absence of such invariants for certain natural programs and specifications, and detect bugs. All this, without the need for user-supplied abstraction predicates.

1 Introduction

We present *Universal Property Directed Reachability* ($\mathsf{PDR}^\forall$), a procedure for automatic inference of quantified inductive invariants, and its application for the analysis of programs that manipulate unbounded data structures such as singly-linked and doubly-linked list data structures. For a correct program, the inductive invariant generated ensures that the program satisfies its specification. For an erroneous program, $\mathsf{PDR}^\forall$ produces a concrete counterexample. Historically, this has been addressed by abstract interpretation [17] algorithms, which automatically infer sound inductive invariants, and bounded model checking algorithms, which explore a limited number of loop iterations in order to systematically look for bugs [6, 13]. We continue the line of recent works [2, 32] which simultaneously search for invariants and counterexamples. We follow Bradley's PDR/IC3 algorithm [9] by repeatedly strengthening a candidate invariant until it either becomes inductive, or a counterexample is found.

In our experience, the correctness of many programs can be proven using universal invariants. Hence, we simplify matters by focusing on inferring universal first-order invariants. When $\mathsf{PDR}^\forall$ terminates, it yields one of the following outcomes: (i) a universal inductive invariant strong enough to show that the program respects the property, (ii) a concrete counterexample which shows that the program violates the desired safety property, or (iii) a *proof that the program cannot be proven correct using a universal invariant* in a given vocabulary.

```
void split(h, g){
i:=h; j:=null; k:=null;
while (i ≠ null){
 if ¬C(i) then {
   if i = h then h:=i.n
   else j.n:=i.n;
   if g = null then g:=i
   else k.n:=i;
   k:=i; i:=i.n;
   k.n:=null;}
 else {j:=i; i:=i.n}
}}
```

requires:
$g = null \wedge H = h \wedge (\forall x, y.\, n^*(x, y) \leftrightarrow L(x, y))$
ensures:
$(\forall z.\, h \neq null \wedge n^*(h, z) \rightarrow C(z)) \wedge$
$(\forall z.\, g \neq null \wedge n^*(g, z) \rightarrow \neg C(z)) \wedge$
$(\forall z.\, z \neq null \rightarrow (L(H, z) \leftrightarrow n^*(h, z) \vee n^*(g, z))) \wedge$
$(\forall x, y.\, L(H, x) \wedge L(x, y) \wedge C(x) \wedge C(y) \rightarrow n^*(x, y)) \wedge$
$(\forall x, y.\, L(H, x) \wedge L(x, y) \wedge \neg C(x) \wedge \neg C(y) \rightarrow n^*(x, y))$

(a) A procedure that moves all the elements not satisfying $C(\cdot)$ from list h to list g and its specification. The ghost variables H and L record the head and the order of elements, respectively, of the original list.

```
void filter(h){
 i:=h; j:=null;
 while (i ≠ null){
   if ¬C(i) then
     if i = h then h:=i.n
     else j.n:=i.n;
   else j:=i;
   i:=i.n
}}
```

$I = L_1 \wedge L_2 \wedge L_3 \wedge L_4 \wedge L_5 \wedge L_6 \wedge L_7$, where
$L_1 = i \neq h \wedge i \neq null \rightarrow n^*(j, i)$
$L_2 = i \neq h \rightarrow C(h)$
$L_3 = n^*(h, j) \vee i \neq j$
$L_4 = \forall x_1.\, i \neq h \wedge n^*(j, x_1) \wedge x_1 \neq j \rightarrow n^*(i, x_1)$
$L_5 = i \neq h \rightarrow C(j)$
$L_6 = \forall x_2.\, z = h \vee j = null \vee$
$\quad \neg n^*(h, x_2) \vee n^*(h, j) \vee \neg C(j)$
$L_7 = \forall x_3.\, j \neq null \wedge n^*(h, x_3) \wedge$
$\quad x_3 \neq h \wedge \neg C(x_3) \rightarrow n^*(j, x_3)$

(b) A procedure that deletes all the elements not satisfying $C(\cdot)$ from list h and its inferred loop invariant.

Fig. 1. Motivating examples. $n^*(x, y)$ means a (possibly empty) path of n-fields from x to y.

Diagram Based Abstraction. Unlike previous work [2, 32], we neither assume that the predicates which constitute the invariants are known, nor apriori bound the number of universal quantifiers. Instead, we rely on first-order theories with a *finite model property*: for such theories, SMT-based tools are able to either return UNSAT, indicating that the negation of a formula φ is valid, or construct a *finite* model σ of φ. We then translate σ into a *diagram* [10]—a formula describing the set of models that extend σ—and use the diagram to construct a *universal* clause to strengthen a candidate invariant.

Property-Directed Invariant Inference. Similarly to IC3, PDR$^\forall$ iteratively constructs an increasing sequence of candidate inductive invariants $F_0, \ldots, F_N$. Every F_i over-approximates the set $\mathcal{R}_i$ of states that can be reached by up to i execution steps from a given set of *initial* states. In every iteration, PDR$^\forall$ uses SMT to check whether one of the candidate invariants became inductive. If so, then the program respects the desired property. If not, PDR$^\forall$ iteratively strengthens the candidate invariants and adds new ones, guided by the considered property. Specifically, it checks if there exists a *bad* state σ which satisfies F_N but not the property. If so, we use SMT again to check whether there is a state σ_a in F_{N-1} that can lead to a state in the *diagram* φ of σ in one execution step. If no such state exists, the candidate invariant F_N can be strengthened by conjoining

it with the negation of φ. Otherwise, we recursively strengthen F_{i-1} to exclude σ_a from its over-approximation of $\mathcal{R}_{i-1}$. If the recursive process tries to strengthen F_0, we stop and use a bounded model checker to look for a counterexample of length N. If no counterexample is found, ***PDR***$^\forall$ *determines that no universal invariant strong enough to prove the desired property exists* (see Lem. 1). We note that PDR$^\forall$ is not guaranteed to terminate, although in our experience it often does.

Example 1. Procedure `split()`, shown in Fig. 1(a), moves the elements not satisfying the condition C from the list pointed to by `h` to the list pointed by `g`. PDR$^\forall$ can infer tricky inductive invariants strong enough to prove several interesting properties: (i) memory safety, i.e., no null dereference and no memory leaks; (ii) all the elements satisfying C are kept in h; (iii) all the elements which do not satisfy C are moved to g; (iv) no new elements are introduced; and (v) stability, i.e., the order between the element satisfying C is not changed. Our implementation verified that `split()` satisfies all the above properties fully automatically by inferring an inductive loop invariant consisting of 44 clauses (among them 27 are universal formulae) in 838 sec.

Example 2. Procedure `filter()`, shown in Fig. 1(b), removes and deallocates the elements not satisfying the condition C from the list pointed to by `h`. The figure also shows the loop invariant inferred by PDR$^\forall$ when it was asked to verify property (iii), shown above. The invariant highlights certain interesting properties of `filter()`. For example, clause L_4 says that if the head element of the list was processed and kept in the list (this is the only way $i \neq h$ can hold), then j becomes an immediate predecessor of i. Clause L_7 says that all the elements x_3 reachable from h and not satisfying C must occur after j.

Experimental Evaluation. We implemented PDR$^\forall$ on top of the decision procedure of [32], and applied it to a collection of procedures that manipulate (possibly sorted) singly linked lists, doubly-linked lists, and multi-linked lists. Our analysis successfully verified interesting specifications, detected bugs in incorrect programs, and established the absence of universal invariants for certain correct programs.

Main Contributions. The main contributions of this work can be summarized as follows.

- We present PDR$^\forall$, a pleasantly simple, yet surprisingly powerful, combination of PDR [9] with a strengthening technique based on diagrams [10]. PDR$^\forall$ enjoys a high-degree of automation because it does *not* require pre-defined abstraction predicates.
- The diagram-based abstraction is particularly interesting as it is determined "on-the-fly" according to the structural properties of the bad states discovered in PDR's traversal of the state space.
- We prove that the diagram-based abstraction is precise in the sense that if PDR$^\forall$ finds a spurious counterexample then the program cannot be proven correct using a universal invariant. We believe that this is a unique feature of our approach.
- We implemented PDR$^\forall$ on top of a decision procedure for logic AE^R [31], and applied it successfully to verify a collection of list-manipulating programs, detect bug, and prove the absence of universal invariants. We show that our technique outperforms an existing state-of-the-art less-automatic PDR-based verification technique [32] which uses the same decision procedure.

2 Preliminaries

This section formalizes the verification problem of programs and sets terminology and notation used in the rest of the paper.

Programs. We assume that the input is a program comprised of a single loop, i.e., it has the form `while` $Cond$ `do` Cmd, where Cmd is loop-free. We handle programs with a more complicated control structures, e.g., nested or multiple loops, by explicitly encoding the program counter.

From Programs to Transition Systems. The semantics of a program is described by a *transition system*, which consists of a set of states and transitions between states.

Program States. We consider the states of the program at the beginning of each iteration of the loop. A program state is represented by a first-order model $\sigma = (\mathcal{U}, \mathcal{I})$ over a vocabulary $\mathcal{V}$ which consists of constants and relation symbols, where $\mathcal{U}$ is the *universe* of the model, and $\mathcal{I}$ is the interpretation function of the symbols in $\mathcal{V}$. For example, to represent memory states of list manipulating programs, we use a vocabulary $\mathcal{V}$ which associates every program variable `x` with a constant x, every boolean field `C` with a unary predicate $C(\cdot)$, and every pointer field `n` with a binary predicate $n^*(\cdot, \cdot)$ which represents its reflexive transitive closure.[5] We use a special constant $null$ to denote the `null` value. We depict memory states $\sigma = (\mathcal{U}, \mathcal{I})$ as directed graphs (see Figure 2). Individuals in $\mathcal{U}$, representing heap locations, are depicted as circles labeled by their name. We draw an edge from the name of constant x and of a unary predicate C to an individual v if $\sigma \models x = v$ or $\sigma \models C(v)$, respectively. We draw an n^*-annotated edge between v and u if $\sigma \models n^*(v, u)$. For clarity, we do not show the edge from v to u if the edges from v to w and from w to u are drawn.

Transition Relation. The set of transitions of a program is defined using a *transition relation*. A transition relation is a set of models of a *double vocabulary* $\hat{\mathcal{V}} = \mathcal{V} \uplus \mathcal{V}'$, where vocabulary $\mathcal{V}$ is used to describe the *source* state of the transition and vocabulary $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$ is used to describe its *target* state: A model $\sigma' = (\mathcal{U}, \mathcal{I}')$ over $\mathcal{V}'$ describes a program state $\sigma = (\mathcal{U}, \mathcal{I})$, where $\mathcal{I}(v) = \mathcal{I}'(v')$ for every symbol $v \in \mathcal{V}$.

Definition 1 (Reduct). *Let $\hat{\sigma} = (\mathcal{U}, \mathcal{I})$ be a model of $\hat{\mathcal{V}}$, and let $\Sigma \subseteq \hat{\mathcal{V}}$. The* reduct of $\hat{\sigma}$ to Σ*, denoted by $\hat{\sigma}[\Sigma]$, is the model $(\mathcal{U}, \mathcal{I}_i)$ of Σ where for every symbol $v \in \Sigma$, $\mathcal{I}_i(v) = \mathcal{I}(v)$.*

We often write a transition $\hat{\sigma}$ as a pair of states (σ_1, σ_2), such that σ_1 is the *reduct* of $\hat{\sigma}$ to vocabulary $\mathcal{V}$, and σ_2 is the state described by the *reduct* to $\mathcal{V}'$. Each transition (σ_1, σ_2) describes one possible execution of the loop body, Cmd, i.e., it relates the state σ_1 at the beginning of an iteration of the loop to the state σ_2 at the end of the iteration. We say that σ_2 is a successor of σ_1, and σ_1 is a predecessor of σ_2.

Properties and Assertions. *Properties* are sets of states. We express properties using logical formulae over $\mathcal{V}$. For example, we express properties of list-manipulation programs, e.g., their pre- and post-conditions, *Pre* and *Post*, respectively, using assertions written in a fragment of first-order logic with transitive closure. In our analysis, these

[5] We reason about list-manipulating programs using logic EA^R [32]. Hence, values of pointer fields n are defined indirectly by a formula over n^*, but n is not included in the vocabulary.

assertions are translated into equisatisfiable first-order logic formulae [31]. We use $(\varphi)'$ to denote the formula obtained by replacing every constant and relation symbol in formula φ with its primed version.

Verification Problem. The *transition system* of a program is represented by a pair $TS = (Init, \rho)$, where $Init$ is a first-order formula over $\mathcal{V}$ used to denote the *initial states* of the program, and ρ is a formula over $\hat{\mathcal{V}}$ used to denote its *transition relation*. A state σ is initial if $\sigma \models Init$, and a pair of states (σ_1, σ_2) is a transition if $(\sigma_1, \sigma_2) \models \rho$. We say that a state is *reachable by at most* i steps of ρ (or i*-reachable* for short, when ρ is clear from the context) if it can be reached by at most i applications of ρ starting from some initial state. We denote the set of i-reachable states by $\mathcal{R}_i$. We say that a state is *reachable* if it is i-reachable for some i. We say that TS *satisfies* a *safety property* $\mathcal{P}$ if all reachable states satisfy $\mathcal{P}$. We often define *Bad* $\stackrel{\text{def}}{=} \neg\mathcal{P}$, and refer to states satisfying *Bad* as *bad states*. We define $\rho \stackrel{\text{def}}{=} Cond \wedge wlp(Cmd, Id)$, where $wlp(Cmd, Id)$ denotes the weakest liberal precondition of the loop body and *Id* is a conjunction of equalities between $\mathcal{V}$ and $\mathcal{V}'$ (see [31] for more details). We define $Init$ and *Bad* using the programs pre- and post- conditions: $Init \stackrel{\text{def}}{=}$ *Pre* and *Bad* $\stackrel{\text{def}}{=} \neg Cond \wedge \neg$*Post*. That is, a state is initial if it satisfies the pre-condition, and it is bad if it satisfies the negation of the loop condition (which indicates termination of the loop) but does not satisfy the post-condition. This captures the requirement that when the loop terminates the post-condition needs to hold.

Example 3. In Exa. 2, $Init \stackrel{\text{def}}{=} (i = h) \wedge (j = null)$ and $Bad \stackrel{\text{def}}{=} (i = null) \wedge \neg((h \neq null) \rightarrow (\forall v.\, n^*(h, v) \rightarrow C(v)))$. Note that these refer to the pre- and post-conditions that should hold right before the loop begins and right after it terminates, respectively. Here, a state is bad if $i = null$ (i.e., it occurs when the loop terminates) and $\mathtt{h}$ points to a non-empty list that contains an element not having the property C.

Invariants. An *invariant* of a program is a property that should hold for all reachable states. It is *inductive* if it is closed under application of ρ.

Definition 2 (Invariants). *Let $TS = (Init, \rho)$ be a transition system and $\mathcal{P}$ a safety property over $\mathcal{V}$. A formula $\mathcal{I}$ is a* safety inductive invariant *(*invariant*, in short) for TS and $\mathcal{P}$ if (i) $Init \Rightarrow \mathcal{I}$, and (ii) $\mathcal{I} \wedge \rho \Rightarrow (\mathcal{I})'$, and (iii) $\mathcal{I} \Rightarrow \mathcal{P}$.*

If there exists an invariant for TS and $\mathcal{P}$, then TS satisfies $\mathcal{P}$. An invariant is *universal* if it is equivalent to a universal formula in prenex normal form. We note that the invariants inferred by $\mathsf{PDR}^\forall$ are conjunctions of *universal clauses*, where a universal clause is a universally quantified disjunction of literals (positive or negative atomic formulae).

3 Universal-Property-Directed Reachability

In this section, we present *Universal Property Directed Reachability* ($\mathsf{PDR}^\forall$), an algorithm for checking if a transition system TS satisfies a safety property $\mathcal{P}$. $\mathsf{PDR}^\forall$ is an adaptation of Bradley's *property-directed reachability* (IC3) algorithm [9] that uses universal formulae instead of propositional predicates [9,22,29] or predicate abstraction [32]. We use Exa. 2 as a running example throughout this section.

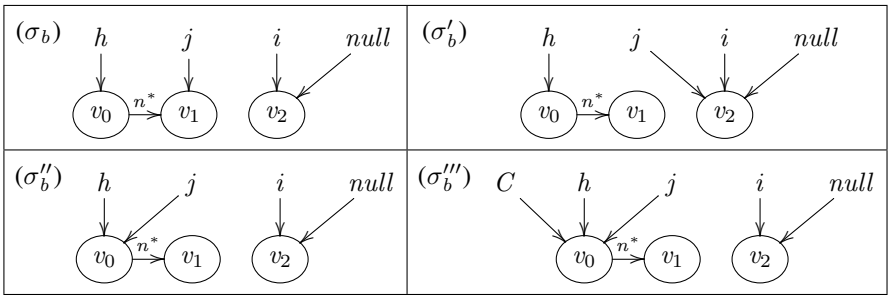

Fig. 2. Graphical depiction of models found during the analysis of the running example

Requirements. We require that the transition relation ρ, as well as the *Init* and *Bad* conditions, are expressible in a first-order logic $\mathcal{L}$ (We can partly handle transitive closure using the approach of [31]. See Section 5.) We require that every satisfiable formula in $\mathcal{L}$ has a finite model, and assume to have a decision procedure $SAT(\psi)$, which checks if a formula ψ in $\mathcal{L}$ is satisfiable, and a function $model(\psi)$, which returns a *finite* model σ of ψ if such a model exists and None otherwise.

3.1 Diagrams as Structural Abstractions

PDR$^\forall$ iteratively strengthens a candidate invariant by retrieving program states that lead to bad states and checking whether the retrieved states are reachable. In that sense, PDR$^\forall$ is similar to IC3. The novel aspect of our approach is the use of *diagrams* [10] to generalize individual states into sets of states before checking for reachability. Diagrams provide a *structural abstraction* of states by existential formulae: The *diagram* of a *finite* model σ, denoted by $Diag(\sigma)$, is an existential cube which describes explicitly the relations between all the elements of the model.[6]

Definition 3 (Diagrams). *Given a* finite *model $\sigma = (\mathcal{U}, \mathcal{I})$ over alphabet $\mathcal{V}$, the* diagram *of σ, denoted by $Diag(\sigma)$, is a formula over alphabet $\mathcal{V}$ which denotes the set of models in which σ can be isomorphically embedded. $Diag(\sigma)$ is constructed as follows.*

- *For every element $e_i \in \mathcal{U}$, a fresh variable x_{e_i} is introduced.*
- *$\varphi_{distinct}$ is a conjunction of inequalities of the form $x_{e_i} \neq x_{e_j}$ for every pair of distinct elements $e_i \neq e_j$ in the model.*
- *$\varphi_{constants}$ is a conjunction of equalities of the form $c = x_e$ for every constant symbol c such that $\sigma \models c = e$.*
- *φ_{atomic} is a conjunction of atomic formulae which include for every predicate $p \in \mathcal{V}$ the atomic formula $p(\bar{x_e})$ if $\sigma \models p(\bar{e})$, and $\neg p(\bar{x_e})$ otherwise.*

Then: $Diag(\sigma) \stackrel{\text{def}}{=} \exists x_{e_1} \ldots x_{e_{|\mathcal{U}|}} . \varphi_{distinct} \wedge \varphi_{constants} \wedge \varphi_{atomic}$.

Intuitively, one can think of $Diag(\sigma)$ as the formula produced by treating individuals in σ as existentially quantified variables and explicitly encoding the interpretation of every

[6] Def. 3, as well as the property formulated by Lem. 1, are an adaptation of the standard model-theoretic notion of a diagram [10].

constant and every predicate using a conjunction of equalities, inequalities, and atomic formulae. For example, the diagram of σ_b, depicted in Figure 2(σ_b), is

$$\begin{aligned} \mathit{Diag}(\sigma_b) \stackrel{\text{def}}{=} & \exists x_0, x_1, x_2.\ x_0 \neq x_1 \wedge x_0 \neq x_2 \wedge x_1 \neq x_2 \wedge \\ & h = x_0 \wedge j = x_1 \wedge i = x_2 \wedge \mathit{null} = x_2 \wedge \\ & \neg C(x_0) \wedge \neg C(x_1) \wedge \neg C(x_2) \wedge \\ & n^*(x_0, x_0) \wedge n^*(x_1, x_1) \wedge n^*(x_2, x_2) \wedge n^*(x_0, x_1) \wedge \\ & \neg n^*(x_0, x_2) \wedge \neg n^*(x_1, x_0) \wedge \neg n^*(x_1, x_2) \wedge \neg n^*(x_2, x_0) \wedge \neg n^*(x_2, x_1) \end{aligned}$$

The first line records the fact that the universe of σ_b consists of three elements. The second line characterizes the interpretations of all the constant symbols in σ_b. The other lines capture precisely the interpretation of predicates C and n^* in σ_b.

Lemma 1. *Let σ be a model over $\mathcal{V}$, and let φ be a closed existential first-order formula over $\mathcal{V}$. If $\sigma \models \varphi$ then $\mathit{Diag}(\sigma) \Rightarrow \varphi$.*

Semantically, Lem. 1 means that for any models σ and σ_i such that $\sigma_i \models \mathit{Diag}(\sigma)$ if $\sigma \models \varphi$ then $\sigma_i \models \varphi$. This implies that if a bad state is reachable from σ *and* the program can be proven correct using an inductive universal invariant $\mathcal{I}$ then *all* the states in σ's diagram are unreachable too: $\mathcal{I}$ is an inductive invariant, thus any state σ leading to a bad state must satisfy (closed existential) formula $\neg\mathcal{I}$. Hence, $\mathit{Diag}(\sigma) \Rightarrow \neg\mathcal{I}$, which means that all states satisfying $\mathit{Diag}(\sigma)$ are unreachable. In this sense, the abstraction based on diagrams is precise for programs with universal invariants.

3.2 Data Structures and Frames

PDR$^\forall$ is shown in Algorithm 1. It uses procedures *block*() and *analyzeCEX*(), shown in Algorithm 2 and Algorithm 3, respectively, as subroutines. The algorithm uses an array F of *frames*, where a frame is a conjunction of universal clauses. For clarity, we refer to the ith entry of the array using subscript notation, i.e., F_i instead of $F[i]$. Intuitively, frame F_i over-approximates $\mathcal{R}_i$, the set of i-reachable states. The algorithm also maintains a *frame counter* N which records the number of frames it developed. We refer to F_0 as the *initial* frame, to F_N as the *frontier* frame, and to any F_i, where $0 \leq i < N$, as a *back* frame.

PDR$^\forall$ maintains several invariants which ensure that every frame F_i is an over approximation of $\mathcal{R}_i$, and hence that the sequence of developed frames is an over approximation of all the traces of the program of length $N + 1$ or less. Technically, this means that the algorithm constructs an *approximate reachability sequence*.

Definition 4. *Let $TS = (\mathit{Init}, \rho)$ be a transition system and $\mathcal{P}$ a safety property. A sequence $\langle F_0, F_1, \ldots, F_N \rangle$ is an* approximate reachability sequence *for TS and $\mathcal{P}$ if:*

(i) $\mathit{Init} \Rightarrow F_0$.
(ii) $F_i \Rightarrow F_{i+1}$, *for all* $0 \leq i < N$, *i.e., for every state σ, if $\sigma \models F_i$ then $\sigma \models F_{i+1}$.*
(iii) $F_i \wedge \rho \Rightarrow (F_{i+1})'$, *for all* $0 \leq i < N$, *i.e., for every transition $(\sigma_1, \sigma_2) \models \rho$, if $\sigma_1 \models F_i$ then $\sigma_2 \models F_{i+1}$.*
(iv) $F_i \Rightarrow \mathcal{P}$, *for all* $0 \leq i \leq N$.

Algorithm 1: $\mathsf{PDR}^{\forall}$ $(\mathit{Init}, \rho, \mathit{Bad})$

```
1  if SAT(Init ∧ Bad) then
2      exit invalid: model(Init ∧ Bad)
3  F_0 := Init
4  F_1 := true
5  N := 1
6  while true do
7      if there exists 0 ≤ j < N
         such that F_{j+1} ⇒ F_j then
8          return valid
9      if ¬SAT(F_i ∧ Bad) then
10         F_{N+1} := true
11         N := N + 1
12     else
13         σ_b := model(F_N ∧ Bad)
14         block(N, σ_b)
```

Algorithm 2: $\mathit{block}(j, \sigma)$

```
21 φ = Diag(σ)
22 while SAT(F_{j-1} ∧ ρ ∧ (φ)') do
23     if 1 < j then
24         σ_a = reduct(model(F_{j-1} ∧ ρ ∧ (φ)'))
25         block(j − 1, σ_a)
26     else
27         analyzeCEX(N)
28 for i = 0 ... j do
29     F_i := F_i ∧ ¬φ
```

Algorithm 3: $\mathit{analyzeCEX}(N)$

```
31 if there exists σ_0, ..., σ_N such that
32    σ_0 ⊨ Init
33    (σ_i, σ_{i+1}) ⊨ ρ for every 0 ≤ i < N, and
34    σ_N ⊨ Bad
35 then exit invalid: σ_0, ..., σ_N
36 else exit No Universal Invariant Exists
```

Items (ii) and (iii) ensure that every frame includes the states of the previous frame and their successors, respectively. Together with item (i), it follows by induction that for every $0 < i \leq N$ the set of states (models) that satisfy F_i is a superset of the set $\mathcal{R}_i$. Furthermore, by item (iv) no frame includes a bad state.

3.3 Iterative Construction of an Approximate Reachability Sequence

$\mathsf{PDR}^{\forall}$ is an iterative algorithm. At every iteration, the algorithm either strengthens the Nth frame, if it contains a bad state, or starts to develop the $N+1$th frame, otherwise. In addition, in every iteration, it might also strengthen some of the back frames. Each strengthening of frame F_i is performed by determining a universal clause φ_i which holds for any i-reachable state, and then conjoining F_i with φ_i.

Initialization. The algorithm first checks that the initial states and the bad states do not intersect. If so, it exits and returns the state that satisfies both Init and Bad as a counterexample (line 2). Otherwise, it sets F_0 to represent the set of initial states (line 3), F_1 to represent all possible states (line 4), and the frame counter to 1. Note that at this point, F_1 is a trivial over-approximation of the set of initial states and their successors, but it might contain bad states.

Iterative Construction. The algorithm then starts its iterative search for an inductive invariant (line 6). Recall that when the algorithm develops the Nth frame, it has already managed to determine an approximate reachability sequence $\langle F_0, \ldots, F_{N-1} \rangle$. Hence, every iteration starts by checking whether a fixpoint has been reached (line 7). If true, then an inductive invariant proving unreachability of *Bad* has been found, and the algorithm returns valid (line 8). Otherwise, the algorithm keeps on strengthening the frontier frame F_N by searching for a *bad witness*, a bad state in the frontier frame

(line 9). If no such state exists, it means that no bad state is N-reachable. Moreover, at this point $\langle F_0, \ldots, F_N \rangle$ is an approximate reachability sequence. Therefore, the iterative strengthening of F_N terminates and the algorithm initializes a new frontier frame to $true$ (lines 10 and 11).

If the frontier frame contains a bad witness, i.e. $F_N \wedge Bad$ is satisfiable, then there *might* be an N-reachable bad state. Due to our requirement for finite satisfiability of the logic, the bad witness is a *finite* model. Given a bad witness σ_b (line 13), the algorithm tries to determine whether it is indeed reachable, and thus the program does not satisfy its specification, or whether σ_b was discovered due to some over-approximation in one of the back frames. This check is done by invoking procedure *block*() with the index of the frontier frame and σ_b as parameters (line 14). The latter either returns a counterexample, determines that it is impossible to prove the specification using a universal invariant (in the given logic and vocabulary), or strengthens the frontier frame to exclude the *set of states in the diagram of* σ_b, and possibly strengthens some back frames too (see below). The iterative construction and strengthening of the frames continues until reaching a fixpoint, finding a counterexample, or determining the absence of a universal invariant.[7]

Example 4. When analyzing the running example, our algorithm discovers that state σ_b, shown in Fig. 2, is a bad witness when $F_1 = true$, and thus it invokes *block*$(1, \sigma_b)$. In this example, *block*() succeeds to block σ_b. Unfortunately, the strengthened frame F_1^1 (see below) still has bad models. Therefore the iterative strengthening continues and the next iterations find σ'_b, depicted in Figure 2, as a bad witness model for F_1^1, σ''_b as a bad witness model of F_1^2 and σ'''_b as a bad witness model of F_1^3. At that point, however, the algorithm determines that the strengthened frame F_1^4 does not have a bad witness. $\langle F_0, F_1^4 \rangle$ is now an approximate reachability sequence and $\mathsf{PDR}^\forall$ goes on and initializes a new frame, F_2, to $true$, and the search for an inductive invariant continues.

Diagram-Based Abstract Blocking. Procedure *block*(j, σ), shown in Algorithm 2, gets an index of a frame $j = 0, \ldots, N$ and a state σ which is included in the jth frame, i.e., $\sigma \models F_j$, and tries to determine whether σ is j-reachable. The unique aspect of our approach is the way in which it abstracts σ to a set of states in order to accelerate the strengthening routine. Namely, the use of diagrams. More specifically, $\mathsf{PDR}^\forall$ computes the diagram φ of σ (line 21) and then checks whether there is a j-reachable state satisfying φ. Importantly, due to Lem. 1, if a universal invariant exists then the generalization of σ to its diagram will not include any reachable state, hence the abstraction is precise in the sense that it maintains unreachability. In this case the strengthening of F_j is also guaranteed to succeed, excluding not only σ, but its entire diagram.

The check if the diagram φ of σ includes a j-reachable state is done conservatively by determining whether some state of φ has a predecessor in F_{j-1}. (Recall that F_{j-1} over-approximates $\mathcal{R}_{j-1}$.) Technically, this is done by checking whether the formula $\delta = F_{j-1} \wedge \rho \wedge (\varphi)'$ is satisfiable (line 22).[8]

Case I. If δ is unsatisfiable, no state represented by φ is j-reachable. Hence, F_j remains an over-approximation of $\mathcal{R}_j$ even if any state of φ is excluded. The exclusion is done

[7] To accelerate the iterative strengthening of frames, any clause φ in F_i that is *inductive* in F_i, i.e., $F_i \wedge \rho \Rightarrow (\varphi)'$ is also propagated forward to F_{i+1}. In particular, this allows to initialize a new frontier frame F_N, for $1 < N$, to a tighter over-approximation of $\mathcal{R}_N$ than $true$ (line 10) [22].

[8] Recall that $(\varphi)'$ is the primed version of φ and hence represents the post state of the transition.

by conjoining the jth frame with the universal formula $\neg\varphi$ (line 29), and results in a strengthening of F_j. In fact, $\neg\varphi$ is conjoined to any back frame (line 28). Strengthening all the back frames is not strictly necessary, it is done to make the fixpoint computation in line 7 cheap: We represent each frame as a set of clauses (with the meaning of conjunction) and check implication by checking inclusion of these sets. We refer to the exclusion of the states of φ as the *blocking* of (the diagram of) σ from frame F_j.

Example 5. In our running example, in the first iteration $\mathit{block}(1, \sigma_b)$ updates F_1^0 to $F_1^1 = \mathit{true} \wedge \neg \mathit{Diag}(\sigma_b)$, and in later iterations it updates $F_1^2 = F_1^1 \wedge \neg \mathit{Diag}(\sigma'_b)$, $F_1^3 = F_1^2 \wedge \neg \mathit{Diag}(\sigma''_b)$, and $F_1^4 = F_1^3 \wedge \neg \mathit{Diag}(\sigma'''_b)$.

Case II. If δ is satisfiable, then there exists an *adverse* state σ_a in frame F_{j-1}, a state which is the predecessor of some state of the diagram of σ that we try to block at frame F_j. Note that σ_a is not necessarily a predecessor of σ itself. The adverse state σ_a is found by taking the reduct of a (finite) model of δ (line 24). If an adverse model σ_a exists then the algorithm *recursively* tries to block it from F_{j-1} (line 25). The recursive procedure continues until the adverse state is either blocked or the algorithm finds an adverse state in the initial frame. Note that blocking an adverse state during the development of the Nth frame leads to a strengthening of some back frame F_i, and thus tightens its over-approximation of $\mathcal{R}_i$. If the algorithm finds an adverse state in the initial frame, then it invokes procedure *analyzeCEX*() to determine whether a bad state can be reached by N applications of the transition relation.

Finding concrete counterexamples and proving the absence of universal invariants. Procedure *analyzeCEX*(), shown in Algorithm 3, looks for a *counterexample*.

Definition 5 (Abstract and Spurious Counterexamples). *A sequence of formulae $\langle\varphi_0, \ldots, \varphi_N\rangle$ is an* abstract counterexample *if the formulas $\varphi_0 \wedge \mathit{Init}$, $\varphi_N \wedge \mathit{Bad}$, and $\varphi_i \wedge \rho \wedge (\varphi_{i+1})'$, for every $i = 0, \ldots, N-1$, are all satisfiable. The abstract counterexample is* spurious *if there exists no sequence of states $\langle\sigma_0, \ldots, \sigma_N\rangle$ such that $\sigma_0 \models \mathit{Init}$, $\sigma_N \not\models \mathit{Bad}$, and for every $0 \leq i < N$, $(\sigma_i, \sigma_{i+1}) \models \rho$.*

An abstract counterexample does not necessarily describe a real counterexample: In order to check if the abstract counterexample is real or spurious, the algorithm checks whether there is an N-reachable bad state (line 31). Technically, *analyzeCEX*() can be implemented using a symbolic bounded model checker [5]. If a real counterexample is found, the algorithm reports it (line 35). Otherwise, the obtained counterexample is *spurious*. Technically, this means that the property is neither verified nor falsified. In our case, the algorithm can determine that the verification effort is doomed: The spurious counterexample is in fact a proof for the absence of a universal invariant (see Prop. 1).

Generalization of blocked diagrams. Rather than blocking a diagram φ from frames $0, \ldots, j$ by conjoining them with the clause $\neg\varphi$ (line 29), our implementation uses a minimal UNSAT core of $F_{j-1} \wedge \rho \wedge (\varphi)'$ to define a clause L which implies $\neg\varphi$, and is also unreachable from F_{j-1}. Blocking is done by conjoining L with F_i for every $i \leq j$.

4 Correctness

In this section we formalize the correctness guarantees of $\mathsf{PDR}^\forall$. We recall that if $\mathsf{PDR}^\forall$ terminates it reports that either the program is safe, the program is not safe, providing a counterexample, or the program cannot be verified using a universal inductive invariant.

Lemma 2. *Let $TS = (Init, \rho)$ be a transition system and let $\mathcal{P}$ be a safety property. If $\mathsf{PDR}^\forall$ returns* valid *then TS satisfies $\mathcal{P}$. Further, if $\mathsf{PDR}^\forall$ returns a counterexample, then TS does not satisfy $\mathcal{P}$.*

Proof. $\mathsf{PDR}^\forall$ returns valid if there exists i such that $F_{i+1} \Rightarrow F_i$. Therefore, $F_i \wedge \rho \Rightarrow (F_{i+1})' \Rightarrow (F_i)'$. Recall that, by the properties of an approximate reachability sequence, $Init \Rightarrow F_0 \Rightarrow F_i$ and $F_i \Rightarrow \mathcal{P}$. Therefore, F_i is an inductive invariant, which ensures that TS satisfies $\mathcal{P}$. The second part of the claim follows immediately from the definition of a counterexample. □

Proposition 1. *Let $TS = (Init, \rho)$ be a transition system and let $\mathcal{P}$ be a safety property. If $\mathsf{PDR}^\forall$ obtains a spurious counterexample $\langle \varphi_0, \ldots, \varphi_N \rangle$ then there exists no universal safety inductive invariant $\mathcal{I}$ for TS and $\mathcal{P}$.*

Proof. Assume that there exists a universal safety inductive invariant $\mathcal{I}$ over $\mathcal{V}$. We show by induction on the distance $i = 0, \ldots, N$, of F_{N-i} from F_N that every state σ_i generated by $\mathsf{PDR}^\forall$ at frame F_i is such that $\sigma_i \models \neg\mathcal{I}$. Because $F_0 \equiv Init$ and by definition $Init \Rightarrow \mathcal{I}$, this implies that no such state is obtained for frame F_0, in contradiction to the existence of a spurious counterexample.

The base case of the induction pertains to F_N. It follows immediately from the property that a state σ_N generated at frame F_N is a model of the formula $F_N \wedge Bad$, and in particular is a model of $Bad = \neg\mathcal{P}$, i.e., $\sigma_N \models \neg\mathcal{P}$. Since $\mathcal{I} \Rightarrow \mathcal{P}$, or equivalently $\neg\mathcal{P} \Rightarrow \neg\mathcal{I}$, we conclude that $\sigma_N \models \neg\mathcal{I}$.

Consider a state generated at frame F_i. Then $\sigma_i = \sigma[\mathcal{V}]$ is the reduct of a model of the formula $F_i \wedge \rho \wedge (Diag(\sigma_{i+1}))'$. Moreover, by the induction hypothesis, $\sigma_{i+1} \models \neg\mathcal{I}$. Since $\neg\mathcal{I}$ is an existential formula, this means by Lem. 1 that $Diag(\sigma_{i+1}) \Rightarrow \neg\mathcal{I}$. We conclude that $F_i \wedge \rho \wedge (Diag(\sigma_{i+1}))' \Rightarrow F_i \wedge \rho \wedge (\neg\mathcal{I})'$. Therefore, σ_i is also (a reduct of) a model of the formula $F_i \wedge \rho \wedge \neg(\mathcal{I})'$. If we assume that $\sigma_i \models \mathcal{I}$, we would get that $\mathcal{I} \wedge \rho \wedge \neg(\mathcal{I})'$ is satisfiable, in contradiction $\mathcal{I}$ being inductive. Hence, $\sigma_i \models \neg\mathcal{I}$. □

Example 6. Procedure `traverseTwo()`, presented in Figure 3 together with its pre- and post-condition, traverses two lists until it finds their last elements. If the lists have a shared tail then `p` and `q` should point to the same element when the traversal terminates. The program indeed satisfies this property. However, this cannot be proven correct using an inductive universal invariant: Take, as usual, $Init$ to be the procedure's precondition and $\mathcal{P}$ to be the safety property whose negation is $Bad = (i = null \wedge j = null) \wedge \neg post$, where *post* is the procedure's postcondition. Consider the state σ_0 depicted in Figure 4. Clearly, this model satisfies *Init*. Therefore, if $\mathcal{I}$ exists, $\sigma_0 \models \mathcal{I}$. σ_0 is a predecessor of σ_1^t and hence it should be the case that $\sigma_1^t \models \mathcal{I}$. Now consider σ_1, which is a submodel of σ_1^t and interprets all constants as in σ_1. If $\mathcal{I}$ is universal, then $\sigma_1 \models \mathcal{I}$ as well. However, $\sigma_1 \not\models \mathcal{P}$, in contradiction to the property of a safety invariant. Indeed, when using

$\mathbf{pre}\colon p = null \wedge q = null \wedge i = g \wedge g \neq null \wedge j = h \wedge h \neq null \wedge$
$\exists v.n^*(g,v) \wedge n^*(h,v) \wedge v \neq null$
$\mathbf{post}\colon p = q \wedge p \neq null \wedge i = null \wedge j = null$

```
void traverseTwo(List g, List h) {
   while (i ≠ null ∨ j ≠ null){
      if i ≠ null then p := i; i := i.n;
      if j ≠ null then q := j; j := j.n}}
```

Fig. 3. A procedure that finds the last elements of two non-empty acyclic lists.

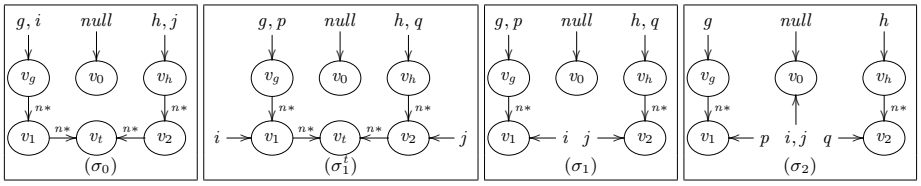

Fig. 4. A spurious counterexample found for procedure `traverseTwo()`, shown in Fig. 3.

PDR$^\forall$, the spurious counterexample $\langle \sigma_0, \sigma_1, \sigma_2 \rangle$ presented in Figure 4 is obtained. This indicates that no universal invariant for $\mathcal{P}$ exists. Note that state σ_1 is a predecessor of σ_2 and recall that σ_0 is a predecessor of σ_1^t. The spurious counterexample was obtained because σ_1^t satisfies the diagram of state σ_1.

5 Implementation and Empirical Evaluation

PDR$^\forall$ is parametric in the vocabulary, and can be implemented on top of any decision procedure for finite satisfiability of first order logic formulas. The language of these formulas should be expressive enough to capture the assertions, transition system, and space of candidate invariants. Our algorithm is not guaranteed to terminate, thus the underlying logic does not have to be decidable. Our implementation, however, uses EA^R which is a decidable logic [32].

EA^R allows for relational first-order formulas with a quantifier prefix of the form $\exists^*\forall^*$ and a deterministic transitive-closure operator *. The latter is used to construct reachability constraints over pointer fields such as `n` in Examples 1 and 2. This logic allows formulas such as $\forall\alpha.\ C(\alpha) \rightarrow n^*(h,\alpha)$ but forbids, e.g. $\forall\alpha\exists\beta.\ n^*(\alpha,\beta) \wedge C(\beta)$ (due to the nesting of an existential quantifier inside a universal one) and $\exists\alpha\forall\beta.\ f(\alpha) = \beta$ (due to the use of a function symbol f). EA^R satisfiability is reducible to effectively-propositional (EPR) satisfiability, also known as the Bernays-Schönfinkel-Ramsey class, which is in turn reducible to Boolean SAT, hence decidable [31, 32]. Moreover, theories in this fragment enjoy the *small model property*, meaning that a satisfiable formula is guaranteed to have a finite model of a size proportional to the depth of quantifier nesting.

Benchmarks. We implemented PDR$^\forall$ and applied it to a collection of procedures that manipulate singly-linked lists, doubly-linked lists, multi-linked lists, and implementations an insertion-sort algorithm [16], and a union-find algorithm [16]. Table 1 summarizes our experimental results.

(a) Verification. Our analyzer successfully verified memory safety, i.e., the absence of null-dereferences and of memory leaks, preservation of data-structure integrity, meaning

Table 1. Experimental results. Running time is measured in seconds. N denotes the highest index for a generated element $F[i]$. "# Z3" denotes the number of calls to Z3. **AF** denotes "Abstraction Failure" of [32]. **TO** means timeout ($>$ 1 hr). (a) Corrent programs; "# Cl. ($\forall$)" = number of ($\forall$-)clauses in the inferred invariant. (b) Correct programs for which there is no universal inductive invariant. (c) Incorrect program; "C.e. size" = size (|domain|) of a model in the counterexample trace. To verify the absence of memory leaks, we either used a unary predicate $alloc(\cdot)$ to record whether a node is allocated, or a ghost variable that keeps the original list. We used a 3.6GHz Intel Core i7 machine with 32GB of RAM, running Ubuntu 14.04 and the 64bit version of Z3 4.4 [19].

	Full				**Memory safety**				**Memory safety** [32]			
(a) Verification	Time	N	# Z3	# Cl. ($\forall$)	Time	N	# Z3	# Cl. ($\forall$)	Time	N	# Z3	# Cl. ($\forall$)
Singly-linked lists												
concat	2.2	3	64	8 (4)	0.9	3	39	4 (1)	**AF**			
delete	17	5	295	24 (11)	1.3	3	55	5	9.7	4	108	11
delete-all	7.9	4	172	16 (9)	0.6	3	34	3 (1)	2.7	3	60	6
filter	45	5	438	26 (18)	1.5	4	56	6 (1)	6.6	5	144	9
insert-at	2.5	3	79	9 (2)	1.5	3	55	8 (1)	7.8	5	157	10
insert	3.5	3	73	9 (2)	1.4	3	54	7 (1)	2.1	3	48	7
merge	431	8	1910	30 (19)	14	6	275	15 (4)	**AF**			
reverse	18	6	255	13 (7)	1.9	4	78	4 (1)	8.4	6	266	5
split	276	8	1245	32 (17)	5.2	5	129	11	24	6	186	10
uf-find	52	9	664	21 (13)	5.4	9	203	7 (2)	8.3	11	309	10
uf-union	117	7	821	30 (13)	64	8	696	21 (6)	**TO**			
Sorted singly-linked lists												
sorted-insert	8.0	3	101	13 (5)	1.8	3	56	8 (1)	26	3	63	10
sorted-merge	454	7	1379	34 (24)	36	6	430	12 (2)	**AF**			
bubble-sort	122	11	980	21 (7)	2.1	5	54	5 (1)	3.5	6	54	2
insertion-sort	2195	13	4829	44 (23)	186	13	1724	30 (5)	**TO**			
Doubly-linked lists												
create	16	7	225	10 (6)	3.6	4	89	7 (2)	47	3	43	6
delete	6.3	4	97	12 (5)	1.5	3	36	5 (2)	403	6	98	8
insert-at	6.2	4	95	14 (6)	3.2	3	64	10 (3)	439	5	208	16
Composite linked-list structures												
nested-flatten	675	17	2849	42 (28)	474	21	2937	22 (7)	**AF**			
nested-split	341	9	960	26 (18)	6.8	4	135	11 (2)	**AF**			
overlaid-delete	294	6	1282	31 (6)	93	7	698	23 (2)	**TO**			
ladder	188	7	793	25 (16)	9.2	6	148	8 (3)	25	3	84	13

(b) Absence of universal invariant	Description	Time	N	Z3
shared-tail	See Exa. 6	3.6	2	42
comb	See Sec. 5(b)	2	3	52

(c) Bug finding	Bug description	Time	N	Z3	C.e. size
insert-at	Precondition is too weak (omitted $e \neq$ `null`)	0.4	1	11	4
filter	Forgot a corner case where $\neg C(h)$	3	1	21	4
insertion-sort	Typo: typed j instead of i	5	4	68	4
sorted-merge	Forgot to link the two segments	7.5	1	49	4

that the procedure never creates cycles in the list, and functional correctness of several singly- and doubly-linked list manipulating procedures. The precondition says that the expected input is a (possibly empty) acyclic list, and the post-condition is the one expected from the procedure's name. For example, the post-condition of `reverse()` is that it returns a list comprised of the same elements as in its input, but in reversed order.

We also verified the correctness of several procedures that manipulate sorted lists: `sorted-insert()` inserts an element into its appropriate place in the sorted list,

`sorted-merge()` creates a sorted list by merging together two sorted ones, and `bubble-sort()` and `insertion-sort()` return sorted permutations of their inputs.

In addition, we verified several procedures that manipulate multi-linked lists. Procedure `overlaid-delete()` takes an overlaid list and deletes a given element. (Overlaid lists use multiple pointer fields to index the same set of elements in different orders.) Procedure `nested-filter()` moves all the elements not satisfying C into a sublist. Procedure `flat()` takes a nested list and flattens it by concatenating its sublists. Procedure `ladder()` creates a copy `t` of a list `h` and places a pointer `p` from every element in `h` to its counterpart in list `t`. We then verify that the `p` field of every element in `h` points to a distinct element in list `t`. This property indicates, indirectly, that both lists have the same length. Finally, we verify the union-find algorithm. E.g., for compressing `find()` operation, we prove the it maintains the reachability between every node and its root and preserves the elements.

We compared our results to [32], where EA^R was used to verify properties of list-manipulating programs with PDR, using human-supplied (universally-quantified) abstraction predicates as templates. We note that [32] can also establish certain functional correctness properties, but theirs are strictly weaker than ours. For example, they do not verify that a reversed list does not contain more elements than in its input list.

(b) Verifying the Absence of Universal Invariants. Our tool was also able to show that certain properties cannot be verified with a universal invariant. It proved that procedure `shared-tail()`, described in Exa. 6, does not have a universal invariant. We applied our tool to procedure `comb()`, which is a simplified version of `ladder()` where the newly allocated elements are not linked together, hence resulting in a heap shaped like a comb. The tool discovered that it is not possible to use a universal invariant to prove that when `comb()` terminates there is no null-valued `p`-field in the input list.

(c) Bug Finding. We also ran our analysis on programs containing deliberate bugs. In all of the cases, the method was able to detect the bug and generate a concrete trace in which the safety or correctness properties are violated.

Remark 1. In our experiments, we noticed that sometimes the tool had to work harder to verify simple properties than when it was asked to verify complicated ones. In particular, verifying partial correctness properties was done faster when verified together with memory safety than without. In hindsight, this might not be surprising due to the property guided nature of the analysis.

6 Related Work

In this section, we briefly summarize the extensive volume of related work.

Synthesizing quantified invariants has received significant attention. Several works have considered discovery of quantified predicates, e.g., based on counterexamples [18] or by extension of predicate abstraction to support free variables [24, 33]. Our inferred invariants are comprised of universally quantified predicates, but unlike these approaches, our computation of the predicates is property directed and does *not* employ predicate abstraction. Additional works for generation of quantified invariants include using abstract domains of *quantified data automata* [25, 26] or ones tailored to Presburger

arithmetic with arrays [20], instantiating quantifier templates [8, 38], applying symbolic proof techniques [30], or using abstractions based on separation logic [4, 21].

Other works aim to identify loop invariants *given* a set of predicates as candidate ingredients. Houdini [23] is the first such algorithm of which we are aware. Santini [39, 40] is a recent algorithm which is based on full predicate abstraction. In the context of IC3, predicate abstraction was used in [7, 12, 32], the last of which specifically targeting shape analysis. In contrast to previous work, our algorithm does not require a pre-defined set of predicates, and is therefore more automatic: The diagrams provide an "on-the-fly" abstraction mechanism.

PDR has been shown to work extremely well in other domains, such as hardware verification [9, 22]. Subsequently, it was generalized to software model checking for program models that use linear real arithmetic [29] and linear rational arithmetic [11]. The latter employs a quantifier-elimination procedure for linear rational arithmetic to provide an approximate pre-image operation. In contrast, our use of diagrams allows us to obtain a natural approximation which is precise for programs that can be verified using universal invariants.

The reduction we use into EPR creates a parametrized array-based system (where the range of the arrays are Booleans). A number of tools have been developed for general array-based systems. The SAFARI [3] system is relevant. It is related to MCMT and Cubicle [14, 15, 27, 28], SAFARI uses symbolic pre-conditions to propagate symbolic states in the form of cubes that are conjunctions of literals over array constraints, and uses interpolants to synthesize universal invariants. Our method for propagating and inductively generalizing diagrams differs by being based on PDR.

The logic used by our implementation has limited capabilities to express properties of list segments that that are not pointed to by variables [32]. This is similar to the self-imposed limitations on expressibility used in a number of shape analysis algorithms [4, 21, 34–37, 41]. Past experience, as well as our own, has shown that despite these limitations it is still possible to successfully analyze a rich set of programs and properties.

7 Conclusions

$\mathsf{PDR}^{\forall}$ is a combination of PDR/IC3 [9] with the model-theoretic notion of diagrams [10]. The latter provide PDR an aggressive strengthening scheme in which the structural properties of a bad state are abstracted "on-the-fly" by a formula describing all of its possible extensions, which are then blocked together within the same iteration of PDR's main refinement loop. This obviates the need for user-supplied abstraction predicates. This form of automation is particularly important when one tries to verify tricky programs, e.g., programs that manipulate unbounded data structures, against a variety (of possibly changing) specifications. Indeed, our implementation successfully analyzed multiple specifications of tricky list-manipulating programs, discovered counterexamples, and, uniquely to our approach, showed that certain programs cannot be proven correct using a universal invariant. We are very pleased with the simplicity of our approach and believe that the notion of diagram-based abstractions is particularly useful for the verification of programs that manipulate unbounded state. In the future, we plan to apply it in other contexts too, e.g., for the verification of network programs [1].

References

1. The Open Networking Foundation. http://opennetworking.org.
2. A. Albarghouthi, J. Berdine, B. Cook, and Z. Kincaid. Spatial interpolants. *CoRR*, abs/1501.04100, 2015.
3. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: smt-based abstraction for arrays with interpolants. In *Computer Aided Verification*, pages 679–685, 2012.
4. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.
5. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.
6. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.
7. J. Birgmeier, A. R. Bradley, and G. Weissenbacher. Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR). In A. Biere and R. Bloem, editors, *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*, volume 8559 of *Lecture Notes in Computer Science*, pages 831–848. Springer, 2014.
8. N. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In F. Logozzo and M. Fähndrich, editors, *Static Analysis Symposium*, volume 7935 of *Lecture Notes in Computer Science*, pages 105–125. Springer, 2013.
9. A. Bradley. SAT-based model checking without unrolling. In *VMCAI*, 2011.
10. C. Chang and H. Keisler. *Model Theory*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1990.
11. A. Cimatti and A. Griggio. Software model checking via IC3. In *CAV*, 2012.
12. A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. IC3 modulo theories via implicit predicate abstraction. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2014.
13. E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pages 368–371, New York, NY, USA, 2003. ACM.
14. S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems - tool paper. In *Computer Aided Verification*, pages 718–724, 2012.
15. S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi. Invariants for finite instances and beyond. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 61–68. IEEE, 2013.
16. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. M.I.T. Press, 1990.
17. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
18. S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 19–32, 2002.
19. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

20. I. Dillig, T. Dillig, and A. Aiken. Symbolic heap abstraction with demand-driven axiomatization of memory invariants. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 397–410. ACM, 2010.
21. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
22. N. Eén, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, 2011.
23. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for Esc/Java. In *FME*, 2001.
24. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. *SIGPLAN Not.*, 37(1), 2002.
25. P. Garg, C. Löding, P. Madhusudan, and D. Neider. Learning universally quantified invariants of linear data structures. In *CAV*, 2013.
26. P. Garg, P. Madhusudan, and G. Parlato. Quantified data automata on skinny trees: An abstract domain for lists. In *SAS*, 2013.
27. S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *Logical Methods in Computer Science*, 6(4), 2010.
28. S. Ghilardi and S. Ranise. MCMT: A model checker modulo theories. In J. Giesl and R. Hähnle, editors, *Automated Reasoning (IJCAR)*, volume 6173 of *Lecture Notes in Computer Science*, pages 22–29. Springer, 2010.
29. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.
30. K. Hoder, L. Kovács, and A. Voronkov. Invariant generation in vampire. In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 60–64. Springer, 2011.
31. S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In *CAV*, 2013.
32. S. Itzhaky, N. Bjørner, T. Reps, M. Sagiv, and A. Thakur. Property-directed shape analysis. In *Computer Aided Verification*, volume 8559, pages 35–51. Springer, 2014.
33. S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Logic*, 9(1), 2007.
34. T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *CAV*, 2006.
35. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI*, 2005.
36. A. Podelski and T. Wies. Counterexample-guided focus. In *POPL*, 2010.
37. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.
38. S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, pages 223–234, 2009.
39. A. Thakur, A. Lal, J. Lim, and T. Reps. PostHat and all that: Attaining most-precise inductive invariants. TR-1790, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, Apr. 2013.
40. A. Thakur, A. Lal, J. Lim, and T. Reps. PostHat and all that: Automating abstract interpretation. *Electr. Notes Theor. Comp. Sci.*, 2013.
41. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, 2004.