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
karbyshev@post.tau.ac.il
[2] Microsoft Research, Redmond, USA
[3] Massachusetts Institute of Technology, Cambridge, USA
[4] The Academic College of Tel Aviv Yaffo, Tel Aviv, Israel

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:

D. Kroening and C.S. Păsăreanu (Eds.): CAV 2015, Part I, LNCS 9206, pp. 583–602, 2015.
DOI: 10.1007/978-3-319-21690-4_40

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

(a) A procedure that moves all the elements not satisfying $C(\cdot)$ from list `h` to list `g` and its specification. `h`,`g`,`i`,`j`, and `k` are pointers to list nodes, and `l.n` denotes the "next" field of node `l`. The ghost variables H and L record the head of the original list and the reachability order between its elements.

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

(b) A procedure that deletes all the nodes 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.

(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.

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 $F_0 \cdots F_N$. 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 Lemma 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 reachability order between the elements 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 36 clauses (among them 19 are universal formulae) in 206 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 a simplified version of property (iii): all the elements which do not satisfy C are removed from `h`. 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 $\mathsf{PDR}^{\forall}$ 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 $\mathsf{PDR}^{\forall}$ 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

Programs. We handle single loop programs, i.e., we assume that a program has the form `while` *Cond* `do` *Cmd*, where *Cmd* is loop-free. We encode more complicated control structures, e.g., nested or multiple loops, by explicitly recording the program counter. For clarity, in our examples we allow for a sequence of instructions preceding the loop. Technically, we encode their effect in the loop's pre-condition.

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.[1] We use a special constant *null* to denote the `null` value. We depict memory states $\sigma = (\mathcal{U}, \mathcal{I})$ as directed graphs (see Fig. 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 edges that can be inferred from the reflexive and transitive nature of n^*.

[1] 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.

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 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 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.

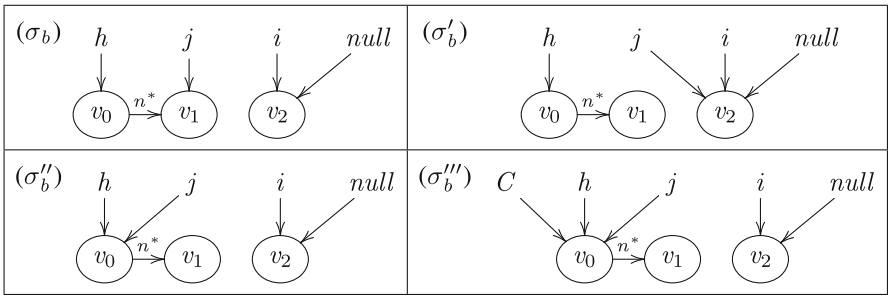

Fig. 2. Graphical depiction of models found during the analysis of the running example.

Example 3. In Example 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 `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 Example 2 as a running example throughout this section.

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 Sect. 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

$\mathsf{PDR}^{\forall}$ 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, $\mathsf{PDR}^{\forall}$ 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.[2]

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

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

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.

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

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

Semantically, Lemma 1 means that for any models σ and σ_i such that $\sigma_i \models Diag(\sigma)$ if $\sigma \models \phi$ then $\sigma_i \models \phi$. 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, $Diag(\sigma) \Rightarrow \neg\mathcal{I}$, which means that all states satisfying $Diag(\sigma)$ are unreachable. In this sense, the abstraction based on diagrams is precise for programs with universal invariants.

3.2 Data Structures and Frames

$\mathsf{PDR}^\forall$ is shown in Algorithm 1. It uses procedures *block*() and *analyzeCEX*(), shown in Algorithms 2 and 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.

$\mathsf{PDR}^\forall$ 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 $TS = (Init, \rho)$ 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) $Init \Rightarrow F_0$.
(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* $0 \leq i \leq N$.

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.

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

```
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_N ∧ 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. $block(j, \sigma)$

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

Algorithm 3. $analyzeCEX(j, N)$

```
31 if j = 0∧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
```

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. Thus, the iterative strengthening of F_N terminates and a new frontier frame is initialized to *true* (line 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.[3]

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 Fig. 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 $block(j, \sigma)$, shown in Algorithm 2, gets an index of a frame $j = 0 \cdots N$ 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 Lemma 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 is an initial state or has a predecessor in F_{j-1}. (Recall that F_{j-1} over-approximates $\mathcal{R}_{j-1}$.) The former is

[3] For efficiency, in our implementation we represent each frame as a set of clauses (with the meaning of conjunction) and check implication (line 7) by checking inclusion of these sets. To facilitate this fixpoint computation, 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].

equivalent to checking if $\varphi \wedge \mathit{Init}$ is satisfiable. Note that if we reached the initial frame, i.e., if $j = 0$, then $\sigma \models \mathit{Init}$, hence the above formula is guaranteed to be satisfiable. Explicitly checking that $\varphi \wedge \mathit{Init}$ is satisfiable is required only at the second frame, i.e., if $j = 1$:

Lemma 2. *For every $1 < j \leq N$, when block(j, σ) is called, $F_i \Rightarrow \neg \mathit{Diag}(\sigma)$ for every $i \leq j - 1$. In particular, $\mathit{Init} \Rightarrow \neg \mathit{Diag}(\sigma)$.*

If the algorithm finds an *adverse initial state*, i.e., an initial state satisfying φ, (line 22),[4] it invokes procedure *analyzeCEX*() for further analysis (see below). Otherwise, the algorithm checks if the formula $\delta = F_{j-1} \wedge \rho \wedge (\varphi)'$ is satisfiable (line 24),[5] i.e., whether some state of φ has a predecessor in F_{j-1}. There can be two cases:

Case I. If δ is unsatisfiable then 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 by conjoining the jth frame with the universal formula $\neg\varphi$ (line 28), and results in a strengthening of F_j. In fact, $\neg\varphi$ is conjoined to any back frame (line 27). 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 $F_1^1 = \mathit{true} \wedge \neg \mathit{Diag}(\sigma_b)$. This excludes σ_b, but also all states where $i = \mathit{null}$, C is empty, and j is n-reachable from h in *any* (nonzero) number of steps. In later iterations *block* 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 25). If an adverse model σ_a exists then the algorithm *recursively* tries to block it from F_{j-1} (line 26). The recursive procedure continues until the adverse state is either blocked or the algorithm finds an adverse initial state (line 22). 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 $\mathcal{R}_i$.

Finding Concrete Counterexamples and Proving the Absence of Universal Invariants. Procedure *analyzeCEX*(), shown in Algorithm 3, is called when an adverse initial state is found. Such a state indicates that an abstract counterexample exists:

[4] If *Init* is a universal formula, then Lemma 2 holds for $j = 1$ as well, hence $j = 1 \wedge \mathit{SAT}(\varphi \wedge \mathit{Init})$ never holds, and its check can be omitted (line 22).

[5] As an optimization, one can consider $\delta' = F_{j-1} \wedge \neg\varphi \wedge \rho \wedge (\varphi)'$ instead of δ. The two formulae are equivalent since $F_{j-1} \Rightarrow \neg\varphi$ (by Lemma 2 for $j > 1$, and since it was checked for $j = 1$), but the strenthening of δ can make the satisfiability check cheaper.

Definition 5 (Abstract and Spurious Counterexamples). *A sequence of formulae* $\langle \phi_j, \phi_{j+1} \cdots \phi_N \rangle$ *is an* abstract counterexample *if the formulae* $\phi_j \wedge \mathit{Init}$, $\phi_N \wedge \mathit{Bad}$, *and* $\varphi_i \wedge \rho \wedge (\phi_{i+1})'$, *for every* $i = j \cdots N-1$, *are all satisfiable. The abstract counterexample is* spurious *if there exists no sequence of states* $\langle \sigma_j, \sigma_{j+1} \cdots \sigma_N \rangle$ *such that* $\sigma_j \models \mathit{Init}$, $\sigma_N \models \mathit{Bad}$, *and for every* $j \leq i < N$, $(\sigma_i, \sigma_{i+1}) \models \rho$.

An abstract counterexample does not necessarily describe a real counterexample. In fact, if $j \neq 0$, the counterexample is necessarily spurious (as, if a real counterexample shorter than N had existed, the algorithm would have already terminated during the development of the $N-1$th frame). However, when $j = 0$, the algorithm determines if the abstract counterexample is real or spurious by checking whether a bad state can be reached by N applications of the transition relation (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 Proposition 1).

Generalization of Blocked Diagrams. Rather than blocking a diagram ϕ from frames $0 \cdots j$ by conjoining them with the clause $\neg\phi$ (line 28), our implementation uses a minimal unsat core of $\psi = ((\mathit{Init})' \vee (F_{j-1} \wedge \rho)) \wedge (\varphi)'$ to define a clause L which implies $\neg\phi$ and is also disjoint from *Init* and unreachable from F_{j-1}. Blocking is done by conjoining L with F_i for every $i \leq j$.[6]

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 3. *Let* $TS = (\mathit{Init}, \rho)$ *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, $\mathit{Init} \Rightarrow F_0 \Rightarrow F_i$ 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. □

[6] We can also use inductive generalization, i.e., look for a minimal subclause L of $\neg\phi$ that is still inductive relative to F_{j-1}, meaning $((\mathit{Init})' \vee (F_{j-1} \wedge L \wedge \rho)) \wedge (\neg L)'$ is unsatisfiable.

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 $\langle \phi_j \cdots \phi_N \rangle$ 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 $N - i = 0 \cdots N$, of F_i from F_N that every state σ_i generated by $\mathsf{PDR}^\forall$ at frame F_i is such that $\sigma_i \models \neg\mathcal{I}$. This implies, by Lemma 1, that every diagram ϕ_i generated by $\mathsf{PDR}^\forall$ at frame F_i is such that $\phi_i \Rightarrow \neg\mathcal{I}$, and hence $\phi_i \Rightarrow \neg Init$. (Recall that by definition $Init \Rightarrow \mathcal{I}$, i.e., $\neg\mathcal{I} \Rightarrow \neg Init$). This contradicts the existence of a spurious counterexample, where $\phi_j \wedge Init$ is satisfiable.

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 σ_i is the reduct of a model of the formula $F_i \wedge \rho \wedge (Diag(\sigma_{i+1}))'$ to $\mathcal{V}$. Moreover, by the induction hypothesis, $\sigma_{i+1} \models \neg\mathcal{I}$. Since $\neg\mathcal{I}$ is an existential formula, this means by Lemma 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 $\mathsf{PDR}^\forall$, 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

$\mathsf{PDR}^\forall$ is parametric in the vocabulary, and can be implemented on top of any decision procedure for finite satisfiability of first-order logic formulae. The language of these formulae should be expressive enough to capture the assertions,

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

```
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.

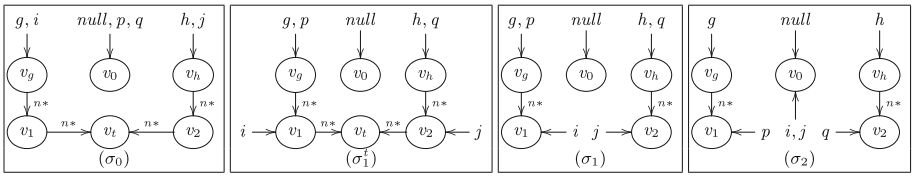

Fig. 4. A spurious counterexample found for procedure `traverseTwo()`, shown in Fig. 3.

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 [31].

EA^R allows for relational first-order formulae with a quantifier prefix of the form $\exists^*\forall^*$ and a deterministic transitive-closure operator *, but forbids functional symbols. We use n^* to construct reachability constraints over the pointer field `n`, e.g., in Examples1 and 2, and to define the "next" relation `n` [31] using a universal formula. We note that The latter can be done only when the prefix of the resulting formula is of the form $\exists^*\forall^*$.

EA^R satisfiability is reducible to effectively-propositional (EPR) satisfiability, also known as the Bernays-Schönfinkel-Ramsey class, and hence is decidable and enjoys the *small model property*, i.e., every satisfiable formula in EA^R is guaranteed to have a finite model. Technically, the reduction introduces axioms (EPR formulae) that capture the reflexivity, transitivity, acyclicity and linearity properties of the * operator [31].

Benchmarks. We implemented $\mathsf{PDR}^\forall$ and applied it to a collection of procedures that manipulate singly-linked lists, doubly-linked lists, multi-linked lists, and implementations of an insertion-sort algorithm [16], and a union-find algorithm [16]. Our experiments were conducted using a 3.6GHz Intel Core i7 machine with 32GB of RAM, running Ubuntu 14.04. We used the 64bit version of Z3 4.4 [19] with the default settings to check satisfiability of EPR formulae. 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 that the procedure never creates cycles in the list, and functional correctness of several singly- and doubly-linked list manipulating

Table 1. Experimental results. Running time is measured in seconds. N denotes the highest index for a developed frame F_i. "# Z3" denotes the number of calls to Z3. **AF** denotes "Abstraction Failure" of [32]. **TO** means timeout (> 1 hr). (a) Correct programs; "# Cl. (∀)" = number of (∀-)clauses in the inferred invariant. (b) Correct programs for which there is no universal inductive invariant. (c) Incorrect program; "C.e. size"is the maximal number of elements in a model that arises in the counterexample.

	Full				**Memory safety**				**Memory safety** [32]			
(a) Verification	Time	N	# Z3	# Cl. (∀)	Time	N	# Z3	# Cl. (∀)	Time	N	# Z3	# Cl. (∀)
— Singly-linked lists —												
concat	2.1	3	59	7 (4)	1.5	4	59	5 (2)	**AF**			
delete	15	5	279	23 (12)	1.5	3	59	7	9.7	4	108	11
delete-all	16	6	300	16 (9)	0.6	3	37	3 (1)	2.7	3	60	6
filter	26	5	336	19 (12)	2.6	4	98	9 (1)	6.6	5	144	9
insert-at	1.9	3	70	9 (2)	1.6	4	60	9 (1)	7.8	5	157	10
insert	3.2	3	71	9 (2)	1.4	3	59	7 (1)	2.1	3	48	7
merge	201	6	1251	34 (22)	12	5	255	13 (3)	**AF**			
reverse	13	5	218	12 (7)	6.0	7	183	5 (1)	8.4	6	266	5
split	206	8	1143	36 (19)	9.6	6	216	13	24	6	186	10
uf-find	37	7	531	21 (13)	4.9	9	201	7 (2)	8.3	11	309	10
uf-union	77	6	618	26 (12)	79	8	819	22 (4)	**TO**			
— Sorted singly-linked lists —												
sorted-insert	6.2	3	95	14 (6)	1.8	3	56	8 (1)	26	3	63	10
sorted-merge	655	8	1822	36 (22)	18	5	263	11 (3)	**AF**			
bubble-sort	112	11	931	24 (8)	2.0	5	53	4 (1)	3.5	6	54	2
insertion-sort	1934	14	4783	41 (18)	265	13	1878	37 (6)	**TO**			
— Doubly-linked lists —												
create	15	6	195	9 (5)	5.5	6	135	7 (2)	47	3	43	6
delete	4.2	3	68	11 (4)	1.5	3	36	5 (2)	403	6	98	8
insert-at	8.0	5	130	15 (6)	2.7	3	60	10 (3)	439	5	208	16
— Composite linked-list structures —												
nested-flatten	734	17	3018	34 (20)	262	14	1714	25 (10)	**AF**			
nested-split	278	9	930	25 (19)	7.3	4	152	9 (1)	**AF**			
overlaid-delete	163	6	918	26 (5)	60	5	518	23 (3)	**TO**			
ladder	117	7	723	30 (16)	9.2	6	152	13 (3)	12	4	70	7

(b) Absence of a universal invariant	Description	Time	N	Z3
shared-tail	See Example 6	3.6	2	42
comb	See Section 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

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. To verify the

absence of memory leaks, we used a unary predicate $alloc(\cdot)$ to record whether a node is allocated. To verify the other properties, we used auxiliary predicates to mark the elements of the input list and record the reachability order between them.

We also verified the correctness of several procedures that manipulate sorted lists: `sorted-insert()` inserts an element into its appropriate place in a sorted list, `sorted-merge()` creates a sorted list by merging two sorted ones, and `bubble-sort()` and `insertion-sort()` sort their input lists. We represented the order on data elements by a binary predicate together with the appropriate axioms.

In addition, we verified several procedures that manipulate multi-linked lists: `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.) `nested-split()` moves all the elements not satisfying C into a sublist. `flatten()` takes a nested list and flattens it by concatenating its sublists. `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 Example 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.

6 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 preconditions 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. Interestingly, 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.

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].

Acknowledgments. We thank Mooly Sagiv and the reviewers for helpful comments. This work was supported by EU FP7 project ADVENT (308830), ERC grant agreement no. [321174-VSSC], by Broadcom Foundation and Tel Aviv University Authentication Initiative, and by BSF grant no. 2012259.

References

1. The Open Networking Foundation. http://opennetworking.org
2. Albarghouthi, A., Berdine, J., Cook, B., Kincaid, Z.: Spatial interpolants. CoRR, abs/1501.04100 (2015)
3. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: SAFARI: SMT-based abstraction for arrays with interpolants. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 679–685. Springer, Heidelberg (2012)
4. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
5. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 118–149 (2003)
6. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
7. Birgmeier, J., Bradley, A.R., Weissenbacher, G.: Counterexample to induction-guided abstraction-refinement (CTIGAR). In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 831–848. Springer, Heidelberg (2014)
8. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013)
9. 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)
10. Chang, C., Keisler, H.: Model Theory. Studies in Logic and the Foundations of Mathematics. Elsevier Science, New York (1990)
11. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 277–293. Springer, Heidelberg (2012)

12. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 46–61. Springer, Heidelberg (2014)
13. Clarke, E., Kroening, D., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: Proceedings of the 40th Annual Design Automation Conference, DAC 2003, pp. 368–371. ACM, New York, NY, USA (2003)
14. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: a parallel SMT-based model checker for parameterized systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 718–724. Springer, Heidelberg (2012)
15. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaïdi, F.: Invariants for finite instances and beyond. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 61–68. IEEE (2013)
16. Cormen, T., Leiserson, C., Rivest, R.: Introduction To Algorithms. MIT Press, Cambridge (1990)
17. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
18. Das, S., Dill, D.L.: Counter-example based predicate discovery in predicate abstraction. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 19–32. Springer, Heidelberg (2002)
19. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
20. Dillig, I., Dillig, T., Aiken, A.: Symbolic heap abstraction with demand-driven axiomatization of memory invariants. In: Cook, W.R., Clarke, S.. Rinard, M.C. (eds.) ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 397–410. ACM (2010)
21. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
22. Eén, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: FMCAD (2011)
23. Flanagan, C., M. Leino, K.R.: 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)
24. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. SIGPLAN Not. **37**(1), 191–202 (2002)
25. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 813–829. Springer, Heidelberg (2013)
26. Garg, P., Madhusudan, P., Parlato, G.: Quantified data automata on skinny trees: an abstract domain for lists. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. LNCS, vol. 7935, pp. 172–193. Springer, Heidelberg (2013)
27. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: termination and invariant synthesis. Log. Methods Comput. Sci. **6**(4), 1–48 (2010)
28. Ghilardi, S., Ranise, S.: MCMT: a model checker modulo theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 22–29. Springer, Heidelberg (2010)

29. 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)
30. Hoder, K., Kovács, L., Voronkov, A.: Invariant generation in vampire. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 60–64. Springer, Heidelberg (2011)
31. Itzhaky, S., Banerjee, A., Immerman, N., Nanevski, A., Sagiv, M.: Effectively-propositional reasoning about reachability in linked data structures. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 756–772. Springer, Heidelberg (2013)
32. Itzhaky, S., Bjørner, N., Reps, T., Sagiv, M., Thakur, A.: Property-directed shape analysis. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 35–51. Springer, Heidelberg (2014)
33. Lahiri, S.K., Bryant, R.E.: Predicate abstraction with indexed predicates. ACM Trans. Comput. Logic **9**(1), 4 (2007). doi:10.1145/1297658.1297662
34. Lev-Ami, T., Immerman, N., Sagiv, M.: Abstraction for shape analysis with fast and precise transformers. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 547–561. Springer, Heidelberg (2006)
35. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, M.: Predicate abstraction and canonical abstraction for singly-linked lists. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 181–198. Springer, Heidelberg (2005)
36. Podelski, A., Wies, T.: Counterexample-guided focus. In: POPL (2010)
37. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. TOPLAS **24**(3), 217–298 (2002)
38. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: PLDI, pp. 223–234 (2009)
39. Thakur, A., Lal, A., Lim, J., Reps, T.: PostHat and all that: attaining most-precise inductive invariants. TR-1790, Computer Science Department, University of Wisconsin, Madison, WI, April 2013
40. Thakur, A., Lal, A., Lim, J., Reps, T.: PostHat and all that: automating abstract interpretation. Electronic Notes in Theoretical Computer Science (2013)
41. Yorsh, G., Reps, T., Sagiv, M.: Symbolically computing most-precise abstract operations for shape analysis. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 530–545. Springer, Heidelberg (2004)