A

Property-Directed Inference of Universal Invariants or Proving Their Absence

ALEKSANDR KARBYSHEV, Tel Aviv University
NIKOLAJ BJØRNER, Microsoft Research
SHACHAR ITZHAKY, Massachusetts Institute of Technology
NOAM RINETZKY, Tel Aviv University
SHARON SHOHAM, The Academic College of Tel Aviv Yaffo

We present *Universal Property Directed Reachability* ($\mathsf{PDR}^{\forall}$), a property-directed semi-algorithm 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*. $\mathsf{PDR}^{\forall}$ is not guaranteed to terminate. However, we prove that under certain conditions, e.g., when reasoning about programs manipulating singly-linked lists, it does.

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.

CCS Concepts: •**Theory of computation** → **Invariants; Program verification;** *Verification by model checking;*

Additional Key Words and Phrases: Universal invariants, Property-directed reachability, IC3, PDR, EPR

ACM Reference Format:
Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham, 2016. Property-Directed Inference of Universal Invariants or Proving Their Absence. *J. ACM* V, N, Article A (January YYYY), 30 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

We present *Universal Property Directed Reachability* ($\mathsf{PDR}^{\forall}$), a semi-algorithm 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 [Cousot and Cousot 1977] 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 [Biere et al. 1999; Clarke et al. 2003]. We continue the line of recent work [Itzhaky et al. 2014; Albarghouthi et al. 2015] which simultaneously search for invariants and counterexamples. We follow Bradley's PDR/IC3 algorithm [Bradley 2011] by repeatedly strengthening a candidate invariant until it either becomes inductive, or a counterexample is found.

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.
Author's addresses: A. Karbyshev, (Current address) Department of Computer Science, Aarhus University, Aabogade 34, DK-8200, Aarhus N, Denmark; N. Bjørner, Microsoft Research, USA; S. Itzhaky, Massachusetts Institute of Technology, USA; N. Rinetzky, Tel Aviv University, Tel Aviv, Israel; S. Shoham, The Academic College of Tel Aviv Yaffo, Israel.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© YYYY ACM. 0004-5411/YYYY/01-ARTA $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

```
void split(List h, List g) {
  i := h;
  j := null; k := null;
  while (i ≠ null) {
    if ¬ ok(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:
$g = null \wedge \underline{h} = h \wedge (\forall x, y.\, n^*(x, y) \leftrightarrow \underline{n}^*(x, y))$

ensures:
$(\forall z.\, h \neq null \wedge n^*(h, z) \rightarrow ok(z)) \wedge$
$(\forall z.\, g \neq null \wedge n^*(g, z) \rightarrow \neg ok(z)) \wedge$
$(\forall z.\, z \neq null \rightarrow (\underline{n}^*(\underline{h}, z) \leftrightarrow n^*(h, z) \vee n^*(g, z))) \wedge$
$(\forall x, y.$
$\underline{n}^*(\underline{h}, x) \wedge \underline{n}^*(x, y) \wedge ok(x) \wedge ok(y) \rightarrow n^*(x, y)) \wedge$
$(\forall x, y.$
$\underline{n}^*(\underline{h}, x) \wedge \underline{n}^*(x, y) \wedge \neg ok(x) \wedge \neg ok(y) \rightarrow n^*(x, y))$

(a) A procedure that moves all the elements not satisfying $ok(\cdot)$ from list `h` to list `g` and its specification using pre- nd post-conditions. Variables `h`, `g`, `i`, `j`, and `k` are pointers to list nodes, and `l.n` denotes the "next" field of node `l`. $n^*(x, y)$ means a (possibly empty) path of `n`-fields from x to y. The ghost variables $\underline{h}$ and $\underline{n}^*(\cdot, \cdot)$ record the head of the original list and the reachability order between its elements.

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

requires:
$true$

ensures:
$\forall z.\, h \neq null \wedge n^*(h, z) \rightarrow ok(z)$

(b) A procedure that deletes all the nodes not satisfying $ok(\cdot)$ from list `h`.

Fig. 1: Motivating examples.

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 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.

Diagram Based Abstraction. Unlike previous work [Itzhaky et al. 2014; Albarghouthi et al. 2015], 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* [Chang and Keisler 1990] — a formula describing the set of models that extend σ — and use the diagram to construct a *universal* clause to strengthen a candidate invariant.

$$
\begin{aligned}
I &= L_1 \wedge L_2 \wedge L_3 \wedge L_4 \wedge L_5 \wedge L_6 \wedge L_7 \qquad \text{where} \\
L_1 &= i \neq h \wedge i \neq \mathit{null} \rightarrow n^*(j,i) \\
L_2 &= i \neq h \rightarrow ok(h) \\
L_3 &= n^*(h,j) \vee i \neq j \\
L_4 &= \forall x.\, i \neq h \wedge n^*(j,x) \wedge x \neq j \rightarrow n^*(i,x) \\
L_5 &= i \neq h \rightarrow ok(j) \\
L_6 &= \forall x.\, x = h \vee j = \mathit{null} \vee \neg n^*(h,x) \vee n^*(h,j) \vee \neg ok(j) \\
L_7 &= \forall x.\, j \neq \mathit{null} \wedge n^*(h,x) \wedge x \neq h \wedge \neg ok(x) \rightarrow n^*(j,x)
\end{aligned}
$$

Fig. 2: Invariant for `filter()`. For better readability, some of the inferred clauses are displayed in the form of implications.

Property-Directed Invariant Inference. Similarly to IC3, $\mathsf{PDR}^\forall$ 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, $\mathsf{PDR}^\forall$ uses SMT to check whether one of the candidate invariants became inductive. If so, then the program respects the desired property. If not, $\mathsf{PDR}^\forall$ 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, *$\mathsf{PDR}^\forall$ determines that no universal invariant strong enough to prove the desired property exists* (see Lemma 4.5). We note that $\mathsf{PDR}^\forall$ is not guaranteed to terminate. In Section 6, we show that under certain conditions, e.g., when reasoning about programs manipulating singly-linked lists, it does. Furthermore, in our experiments it terminates even when these conditions do not hold.

Example 1.1. Procedure `split()`, shown in Figure 1(a), moves the elements not satisfying the condition ok from the list pointed to by `h` to the list pointed by `g`. $\mathsf{PDR}^\forall$ 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 ok are kept in h; (iii) all the elements which do not satisfy ok are moved to g; (iv) no new elements are introduced; and (v) stability, i.e., the reachability order between the elements satisfying ok is not changed. Our implementation verified that `split()` satisfies all the above properties fully automatically by inferring an inductive loop invariant consisting of 33 clauses (among them 17 are universal formulae). (The invariant is given in Appendix A.)

Example 1.2. Procedure `filter()`, shown in Figure 1(b), removes and deallocates the elements not satisfying the condition ok from the list pointed to by `h`. Figure 2 shows the loop invariant inferred by $\mathsf{PDR}^\forall$ when it was asked to verify a simplified version of property (iii): all the elements which do not satisfy ok 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 reachable from h and not satisfying ok must occur after j.

Experimental Evaluation. We implemented $\mathsf{PDR}^\forall$ on top of the decision procedure of [Itzhaky et al. 2014], 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 $\mathsf{PDR}^{\forall}$, a pleasantly simple, yet surprisingly powerful, combination of PDR [Bradley 2011] with a strengthening technique based on diagrams [Chang and Keisler 1990]. $\mathsf{PDR}^{\forall}$ enjoys a high degree of automation because it does *not* require predefined 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 provide sufficient conditions that ensure that $\mathsf{PDR}^{\forall}$ terminates.
- We implemented $\mathsf{PDR}^{\forall}$ on top of a decision procedure for the logic EA^R [Itzhaky et al. 2013][1], and applied it successfully to verify a collection of list-manipulating programs, detect bugs, and prove the absence of universal invariants. We show that our technique outperforms an existing state-of-the-art PDR-based verification technique [Itzhaky et al. 2014] which uses the same decision procedure but requires user-supplied abstraction predicates. The implementation is available for download at `https://bitbucket.org/tausigplan/updr-distrib/`
- The modeling of acyclic lists is based on the encoding developed in [Itzhaky et al. 2013]. We also present a novel encoding that allows to model programs that manipulate (restricted) *cyclic* lists in EA^R and to apply our analysis to them.

2. PRELIMINARIES

This section formalizes the verification problem and sets terminology and notation. We start by explaining the way in which we use first-order logic to represent a transition system, which consists of a set of states and transitions between states. We then explain how we translate a program into a transition system and obtain a verification problem which captures the correctness of the program.

2.1. Verification Problems and Their Representation in First-Order Logic

States. A state is represented by a *finite*[2] first-order model $\sigma = (D, \mathcal{I})$ over a vocabulary $\mathcal{V}$ which consists of constants and relation symbols, where D is the finite *domain* of the model, and $\mathcal{I}$ is the interpretation function of the symbols in $\mathcal{V}$. We assume that the domain D of every state is a subset of a fixed set $\mathcal{U}$, called a *universe*.

Transition Relation. The set of transitions of a transition system 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 $\sigma' = (D, \mathcal{I}')$ over $\mathcal{V}'$ describes a program state $\sigma = (D, \mathcal{I})$, where $\mathcal{I}(v) = \mathcal{I}'(v')$ for every symbol $v \in \mathcal{V}$.

Definition 2.1 (*Reduct*). Let $\hat{\sigma} = (D, \mathcal{I})$ be a model of $\hat{\mathcal{V}}$, and let $\Sigma \subseteq \hat{\mathcal{V}}$. The *reduct of* $\hat{\sigma}$ *to* Σ, denoted $\mathit{reduct}_{\Sigma}(\hat{\sigma})$, is the model $(D, \mathcal{I}_{\Sigma})$ of Σ where for every symbol $v \in \Sigma$, $\mathcal{I}_{\Sigma}(v) = \mathcal{I}(v)$.

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 $\mathcal{V}'$. We say that σ_2 is a successor of σ_1, and σ_1 is a predecessor of σ_2.

[1] In [Itzhaky et al. 2013], the logic AE^R was presented, whose *validity* is decidable. In this paper, we are interested in *satisfiability* and consider the logic EA^R. The negation of an EA^R-formula is an AE^R-formula, hence (un)satisfiability of EA^R can be reduced to validity of AE^R, and is hence decidable.

[2] All first-order models considered in this work are finite, i.e., have a finite domain.

Verification Problem. A *transition system* is represented by a pair $TS = (Init, \rho)$, where $Init$ is a closed first-order formula over $\mathcal{V}$ used to denote the *initial states* of the program, and ρ is a closed 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*.

Properties and Assertions. Properties are sets of states. We express properties, such as pre- and post-conditions, and assertions within the loop body, using closed logical formulae over $\mathcal{V}$.

Invariants. An *invariant* of a transition system is a property that should hold for all reachable states. It is *inductive* if it is closed under application of ρ. In the following, we use $(\varphi)'$ to denote the formula obtained by replacing every constant and relation symbol in formula φ with its primed version.

Definition 2.2 (*Invariants*). Let $TS = (Init, \rho)$ be a transition system and $\mathcal{P}$ a safety property over $\mathcal{V}$. A closed 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 (i.e., a formula with a $\forall^*$ quantifier prefix in prenex normal form). We note that the invariants inferred by PDR$^\forall$ are conjunctions of *universal clauses*, where a universal clause is a universally quantified disjunction of literals (positive or negative atomic formulae).

2.2. From Programs to Verification Problems

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 precondition.

Program Semantics and Verification Problem. The semantics of a program is described by a transition system. We consider the states of the program at the beginning of each iteration of the loop. 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.

Technically, following [Itzhaky et al. 2013], we derive the semantics of the loop body as a transition relation formula ρ from a *weakest liberal precondition* predicate transformer, *wlp*, defined for each command type. As an example, the top of Table II presents the definition of *wlp* for the simple language IMP [Winskel 1993]. To construct the transition relation using *wlp*, we define an *identity* formula *Id* that specifies that the input and the output states are identical. That is, *Id* is a two-vocabulary closed formula such that $(\sigma, \sigma') \models Id \Leftrightarrow \sigma = \sigma'$. Formally, *Id* is defined by

$$Id \stackrel{\text{def}}{=} \bigwedge_{c \in \mathcal{C}} c = c' \ \wedge \bigwedge_{R \in \mathcal{R}} \forall \overline{\alpha}.\ R(\overline{\alpha}) \leftrightarrow R'(\overline{\alpha}) \tag{1}$$

where $\mathcal{C}$ and $\mathcal{R}$ denote the sets of constants and relation symbols in $\mathcal{V}$, respectively, and $\overline{\alpha}$ is a list of variables according to the arity of the relation symbol R. The vocabulary $\mathcal{V}$ corresponds to the structure σ, and $\mathcal{V}'$ corresponds to σ'.

We then define

$$\rho \stackrel{\text{def}}{=} Cond \wedge wlp[\![Cmd]\!]\, Id \tag{2}$$

Γ_{n^*}	$\stackrel{\text{def}}{=}$	$\forall\alpha,\beta.\, n^*(\alpha,\beta) \wedge n^*(\beta,\alpha) \leftrightarrow \alpha = \beta$	reflexivity + acyclicity
		$\wedge\ \forall\alpha,\beta,\gamma.\, n^*(\alpha,\beta) \wedge n^*(\beta,\gamma) \to n^*(\alpha,\gamma)$	transitivity
		$\wedge\ \forall\alpha,\beta,\gamma.\, n^*(\alpha,\beta) \wedge n^*(\alpha,\gamma) \to n^*(\beta,\gamma) \vee n^*(\gamma,\beta)$	semi-linearity

Table I: Effectively propositional axiomatization of deterministic reflexive-transitive closure.

where $wlp[\![Cmd]\!]$ denotes the weakest liberal precondition of the loop body.

We define $Init$ and Bad using the programs pre- and post-conditions, as well as its assertions:

$$Init \stackrel{\text{def}}{=} Pre \qquad \text{and} \qquad Bad \stackrel{\text{def}}{=} (\neg Cond \wedge \neg Post) \vee (Cond \wedge \neg wlp[\![Cmd]\!]\,\mathbf{true}). \tag{3}$$

That is, a state is initial if it satisfies the precondition, and it is bad in one of two cases: (i) if the state 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. (ii) if the state leads to a violation of an assertion within the loop body when it is encountered in the loop head. This is captured by the subformula $\neg wlp[\![Cmd]\!]\,\mathbf{true}$ of Bad.

The construction of ρ, $Init$ and Bad ensures that $TS = (Init, \rho)$ satisfies $\mathcal{P} = \neg Bad$ if and only if any execution of the program starting at a state which satisfies the given precondition never violates an assertion, and if it terminates then it ends in a state which satisfies the postcondition.

3. REASONING ABOUT HEAP-MANIPULATING PROGRAMS USING EFFECTIVELY PROPOSITIONAL LOGIC

In this section we exemplify how we represent heap-manipulating programs, such as the ones used in our running examples and experiments, as well as the corresponding verification problems in first-order logic.

We start by defining the fragment of logic used, and continue to describe the construction of the formulae ρ, $Init$ and Bad for a program. First, we present the construction of the formulae for programs that manipulate *acyclic* data structures, as developed in [Itzhaky et al. 2013]. Next, we develop a novel construction that also handles restricted *cyclic* data structures.

EPR and EA^R. Effectively-Propositional logic (*EPR*), also known as the Bernays-Schönfinkel-Ramsey class, is a fragment of first-order logic which allows for relational first-order formulae with a quantifier prefix of the form $\exists^*\forall^*$, but forbids functional symbols. Satisfiability of *EPR* is decidable. *EPR* enjoys the *small model property*: every satisfiable formula in *EPR* is guaranteed to have a finite model [Lewis 1980].

In our running examples and experiments, we represent programs and the corresponding verification problems using *EA^R* [Itzhaky et al. 2013], an auxiliary logic built on top of *EPR*, which enables natural reasoning about programs manipulating linked-data structures. *EA^R* extends *EPR* by allowing a deterministic transitive-closure operator * over acyclic relations. Satisfiability of *EA^R* is reducible to that of *EPR*, and enjoys the same properties. Technically, the reduction introduces first-order axioms (*EPR* formulae) that provide a complete characterization of *. These axioms are given in Table I.

Programs manipulating linked-data structures as transition systems. To represent memory states of list manipulating programs, we fix an infinite countable universe $\mathcal{U}$ whose individuals represent dynamically allocated objects. Recall that a state is represented by a finite first-order model $\sigma = (D, \mathcal{I})$ with $D \subseteq \mathcal{U}$.

We use a vocabulary $\mathcal{V}$ which associates every program variable `x` with a constant x, contains a designated constant $null$ to denote the null value, and contains the special binary predicate symbol $n^*(\cdot\,,\cdot)$ which defines reachability over every pointer field `n`, e.g., in Examples 1.1 and 1.2. Notice that n itself is not part of the vocabulary, but it is definable using the (open) formula φ_{n} in Table II. We use $n^+(\alpha, \beta)$ as a shorthand for $n^*(\alpha, \beta) \wedge \alpha \neq \beta$. Clearly, these definitions for φ_{n} and n^+ rely on acyclicity of n: if there was a cycle, then for every node u on the cycle we would have $n^+(u, u)$, and also for any two nodes u, v we would have $n^*(u, v)$, so there would not be enough information in n^* to define φ_{n} based on it. In particular, the order of the node in the cycle is not encoded in n^*.

In addition, we represent a Boolean function ok with a unary predicate $ok(\cdot)$, and an order relation (e.g. for sorting) with a binary predicate $R(\cdot\,, \cdot)$.

We depict memory states $\sigma = (D, \mathcal{I})$ as directed graphs (e.g., Figure 3). Individuals in D, representing heap locations, are depicted as circles labeled by their name. We draw an edge from the name of constant x or a unary predicate ok to an individual v if $\sigma \models x = v$ or $\sigma \models ok(v)$, respectively. For clarity, we do not directly depict the interpretation of the n^* relation. Instead, we use a more compact drawing scheme where we draw an n-annotated edge between v and u if $\sigma \models \varphi_{\mathsf{n}}(v, u)$. The interpretation of n^* can be inferred from the n-annotated edges by (i) omitting the incoming edges of the element that corresponds to the $null$ constant, and (ii) considering the reflexive transitive closure of the remaining edges.

Transition relation. We express the semantics of loop-free code as a transition relation ρ over the above vocabulary by defining a *weakest liberal precondition* predicate transformer, $wlp[\![-]\!]$, for each command type. We do this in a simple language IMP^R, which is an extension of IMP [Winskel 1993] with heap-related commands. The rules for *wlp* are shown in Table II. The notation $Q[t/x]$ is used to denote substitution of all the occurrences of the constant x in Q with the term t. The notation $Q[\varphi/n^*(\alpha, \beta)]$ denotes substitution of any atom of the form $n^*(\cdot\,, \cdot)$ in Q with the formula φ, where α and β may occur as term placeholders in φ and are filled in with the arguments of n^*. *vars* is the set of (constant symbols pertaining to) variables used in the program. As shown in [Itzhaky et al. 2014], the rules for *wlp* are sound and complete.

The encoding of lists using n^* and the corresponding update rules *wlp* may seem confounding at first, but follow a fairly simple intuition: when removing a pointer link, all paths that go through the changed node are disconnected; when adding a link, all paths into the source get connected to the (single) path from the target. This is expressed, respectively, by the formulae

$$n^*(\alpha, \beta) \wedge (\neg n^*(\alpha, x) \vee n^*(\beta, x)) \quad (\text{for } x.\mathsf{n} := \mathsf{null})$$

and

$$n^*(\alpha, \beta) \vee (y \neq null \wedge n^*(\alpha, x) \wedge n^*(y, \beta)) \quad (\text{for } x.\mathsf{n} := y)\,.$$

The former describes all n-paths except those that go through x, and the latter describes all n-paths with the addition of those that were connected by the new edge from x to y. Here, α and β denote aribitrary heap locations. When traversing a pointer ($x := y.\mathsf{n}$), the successor can be expressed by its transitive closure n^* using the formula φ_{n}: for any two locations s and t (which are not $null$), the successor of s is t iff t is on the path starting at s (but not s itself), and no other node lies between s and t. The successor of s is $null$ iff the path starting at s is empty. Our ability to recover the successor from n^* is key to having a complete encoding of heap structures using transitive closure.

Given the *wlp* definition, the transition relation ρ is defined as in Equation (2) (see Section 2.2). It is important to notice that, since φ_{n} is a universal formula occurring in a negative context in $wlp[\![x := y.\mathsf{n}]\!]Q$, which is itself defined by a universal formula, the resulting formula ρ will have a quantifier prefix $\forall^*\exists^*$. We would like to get an EA^R formula for our purposes; we achieve this by changing the quantified rules slightly, resulting in the variants in Table III. All other rules remain unchanged. From here on we switch to the definition of the transition relation as

$$\rho \stackrel{\text{def}}{=} Cond \wedge wlp^{\exists}[\![Cmd]\!]Id \tag{4}$$

Notice that the equivalence of semantics relies on the fact that for every location $s \in \mathcal{U}$ which is different from $null$ there is exactly one location t such that $\varphi_{\mathsf{n}}(s, t)$, as follows from the definition.

$$
\begin{array}{rcl}
\mathit{wlp}[\![\mathsf{skip}]\!]\,Q & \stackrel{\text{def}}{=} & Q \\
\mathit{wlp}[\![x := y]\!]\,Q & \stackrel{\text{def}}{=} & Q[y/x] \\
\mathit{wlp}[\![\mathit{Cmd}_1\ ;\ \mathit{Cmd}_2]\!]\,Q & \stackrel{\text{def}}{=} & \mathit{wlp}[\![\mathit{Cmd}_1]\!]\big(\mathit{wlp}[\![\mathit{Cmd}_2]\!]\,Q\big) \\
\mathit{wlp}[\![\mathsf{if}\ B\ \mathsf{then}\ \mathit{Cmd}_1\ \mathsf{else}\ \mathit{Cmd}_2]\!]\,Q & \stackrel{\text{def}}{=} & [\![B]\!] \wedge \mathit{wlp}[\![\mathit{Cmd}_1]\!]\,Q \vee \neg[\![B]\!] \wedge \mathit{wlp}[\![\mathit{Cmd}_2]\!]\,Q \\
\mathit{wlp}[\![\mathsf{assert}\ B]\!]\,Q & \stackrel{\text{def}}{=} & [\![B]\!] \wedge Q
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{wlp}[\![x.\mathsf{n} := \mathsf{null}]\!]\,Q & \stackrel{\text{def}}{=} & Q[n^*(\alpha,\beta) \wedge (\neg n^*(\alpha,x) \vee n^*(\beta,x))/n^*(\alpha,\beta)] \\
\mathit{wlp}[\![x.\mathsf{n} := y]\!]\,Q & \stackrel{\text{def}}{=} & \neg n^*(y,x) \wedge Q[n^*(\alpha,\beta) \vee (y \neq \mathit{null} \wedge n^*(\alpha,x) \wedge n^*(y,\beta))/n^*(\alpha,\beta)] \\
\mathit{wlp}[\![x := y.\mathsf{n}]\!]\,Q & \stackrel{\text{def}}{=} & \forall\alpha.\,\varphi_{\mathsf{n}}(y,\alpha) \rightarrow Q[\alpha/x] \\
\mathit{where}\ \ \varphi_{\mathsf{n}}(s,t) & \stackrel{\text{def}}{=} & (n^+(s,t) \wedge \forall\gamma.\,n^+(s,\gamma) \rightarrow n^*(t,\gamma)) \vee (t = \mathit{null} \wedge \forall\gamma.\,\neg n^+(s,\gamma)) \\
\mathit{wlp}[\![x := \mathsf{new}]\!]\,Q & \stackrel{\text{def}}{=} & \forall\alpha.\,\big(\bigwedge_{p\in \mathit{vars}\cup\{\mathit{null}\}} \neg n^*(p,\alpha)\big) \rightarrow Q[\alpha/x]
\end{array}
$$

Table II: Rules for computing weakest liberal preconditions for procedures in IMP^R. Q is a post-condition expressed as a first-order formula. The top frame shows the standard *wlp* rules for IMP, the bottom frame contains our additions for heap updates, dereference, and memory allocation. We assume that the program nullifies a field before modifying it, i.e., every command of the form $x.\mathsf{n} := y$ is preceded by a command $x.\mathsf{n} := \mathsf{null}$.

$$
\begin{array}{rcl}
\mathit{wlp}^{\exists}[\![x := y.\mathsf{n}]\!]\,Q & \stackrel{\text{def}}{=} & \exists\alpha.\,\varphi_{\mathsf{n}}(y,\alpha) \wedge Q[\alpha/x] \\
\mathit{wlp}^{\exists}[\![x := \mathsf{new}]\!]\,Q & \stackrel{\text{def}}{=} & \exists\alpha.\,\big(\bigwedge_{p\in \mathit{vars}\cup\{\mathit{null}\}} \neg n^*(p,\alpha)\big) \wedge Q[\alpha/x]
\end{array}
$$

Table III: Leading-existential variant of *wlp* rules.

Example 3.1. Since the transition relation obtained for `filter()` is large, we demonstrate the construction of ρ on a simpler program, where the loop body consists of the following command:

$$\mathit{Cmd} \ = \ k := i.\mathsf{n}\,;\ i.\mathsf{n} := \mathsf{null}\,;\ i := k$$

We then have

$$
\begin{aligned}
\mathit{wlp}^{\exists}[\![\mathit{Cmd}]\!]\,Q &= \mathit{wlp}^{\exists}[\![k := i.\mathsf{n}\,;\ i.\mathsf{n} := \mathsf{null}\,;\ i := k]\!]\,Q \\
&= \mathit{wlp}^{\exists}[\![k := i.\mathsf{n}]\!]\big(\mathit{wlp}^{\exists}[\![i.\mathsf{n} := \mathsf{null}]\!]\big(\mathit{wlp}^{\exists}[\![i := k]\!]\,Q\big)\big) \\
&= \exists\alpha.\,\varphi_{\mathsf{n}}(i,\alpha) \wedge \big(\mathit{wlp}^{\exists}[\![i.\mathsf{n} := \mathsf{null}]\!](Q[k/i])\big)[\alpha/k] \\
&= \exists\alpha.\,\varphi_{\mathsf{n}}(i,\alpha) \wedge \big((Q[k/i])[n^*(\alpha,\beta) \wedge (\neg n^*(\alpha,i) \vee n^*(\beta,i))/n^*(\alpha,\beta)]\big)[\alpha/k]
\end{aligned}
$$

The transition relation is then constructed as follows:

$$
\begin{aligned}
\rho &= i \neq \mathit{null} \wedge \mathit{wlp}^{\exists}[\![\mathit{Cmd}]\!]\mathit{Id} \\
&= i \neq \mathit{null} \wedge \mathit{wlp}^{\exists}[\![\mathit{Cmd}]\!](i = i' \wedge k = k' \wedge \forall\alpha,\beta.\,n^*(\alpha,\beta) \leftrightarrow n^{*\prime}(\alpha,\beta)) \\
&= i \neq \mathit{null} \wedge \\
&\quad \exists\alpha.\,\varphi_{\mathsf{n}}(i,\alpha) \wedge \big(\alpha = i' \wedge \alpha = k' \wedge \forall\alpha,\beta.\,\big(n^*(\alpha,\beta) \wedge (\neg n^*(\alpha,i) \vee n^*(\beta,i))\big) \leftrightarrow n^{*\prime}(\alpha,\beta)\big)
\end{aligned}
$$

PROPOSITION 3.2. *EA^R is closed under $\mathit{wlp}^{\exists}[\![\mathit{Cmd}]\!]$; that is, if $Q \in \mathit{EA}^R$ then $\mathit{wlp}^{\exists}[\![\mathit{Cmd}]\!]\,Q \in \mathit{EA}^R$. In particular, $\rho \in \mathit{EA}^R$ (as defined by Equation* (4)*).*

Initial and bad states. We express properties of list-manipulating programs, e.g., their pre- and post-conditions, *Pre* and *Post*, respectively, using assertions written in EA^R over the above vocabulary. $Init$ and Bad are defined based on these assertions, as shown in Equation (3) (see Section 2.2).

Example 3.3. In Example 1.2, we have $Pre = i = h \wedge j = null$ and $Post = h \neq null \rightarrow \forall z.\, n^*(h, z) \rightarrow ok(z)$. Note that these refer to the pre- and post-conditions that should hold right before the loop begins and right after it terminates, respectively. Therefore, $Init \stackrel{\text{def}}{=} i = h \wedge j = null$ and $Bad \stackrel{\text{def}}{=} i = null \wedge \neg(h \neq null \rightarrow \forall z.\, n^*(h, z) \rightarrow ok(z))$. 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 ok. In this example there are no assert statements in the body, hence the second disjunct in the definition of Bad in Equation (3), which captures the semantics of assertion violations, simplifies to $\mathbf{false}$ and is subsumed by the first disjunct.

In our analysis, the EA^R formulae $Init$, Bad and ρ are translated into equisatisfiable *EPR* formulae [Itzhaky et al. 2013].

3.1. Modeling of Programs Manipulating Cyclic Linked Lists

As an extension of previous work [Itzhaky et al. 2014; Karbyshev et al. 2015], which targeted acyclic data structures, we augment the formalism shown above to handle a restricted form of cycles. The new formalism allows at most one cycle to be present in the heap at any given time. This is achieved by decomposing the pointer edges, labeled n, into a set of acyclic edges labeled k plus *at most one* additional edge labeled m. This is always possible — if the heap contains (at most) one cycle, then it is enough to remove (at most) one edge to make it acyclic.

We denote n^* and k^* the reflexive transitive closures of n and k, and $\langle m_s, m_t \rangle$ the source and destination of the edge labeled m, if it is present (if m is not present, $m_s = m_t = null$). The following relationship holds between n^* and k^*, m_s, m_t:

$$\forall \alpha, \beta.\, n^*(\alpha, \beta) \;\Leftrightarrow\; k^*(\alpha, \beta) \vee (m_s \neq null \wedge k^*(\alpha, m_s) \wedge k^*(m_t, \beta)) \qquad (5)$$

Therefore k^*, m_s, m_t fully characterize the heap reachability. The axioms in Table I now hold for k^* instead of n^*. In addition, we require that if m is present, then m_s has no k-successor, and the edge m closes the cycle; that is, $m_s \neq null \rightarrow k^*(m_t, m_s) \wedge \neg\exists \alpha.\, k^+(m_s, \alpha)$. We now modify the *wlp* formulae from Table II to reflect the new situation. The new semantics for $x.\mathsf{n} := \mathsf{null}$, $x.\mathsf{n} := y$, and $x := y.\mathsf{n}$ are shown in Table IV. The multiple substitutions in the brackets are done in parallel. The operator $\mathrm{ite}(p, a, b)$ denotes a term that is equal to a if p is $\mathbf{true}$, and b otherwise.[3] $\psi_{\mathsf{km}}(s, t)$ is a formula expressing a path between s and t utilizing the special edge m. $\psi_{\mathsf{km}}(u, u)$ means that u lies on the cycle.

The semantics maintains the edge m by creating it when a cycle is closed as a result of an assignment of the form $x.\mathsf{n} := y$, and removing it or replacing it with a k edge when the cycle is broken by $x.\mathsf{n} := \mathsf{null}$. Notice that $wlp[\![x := y.\mathsf{n}]\!]$ remains as in Table II, except that the definition of φ_{n} is changed. The adjustment in Table III for $wlp^{\exists}$ is suitable in this case as well.

4. 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 [Bradley 2011] that uses universal formulae instead of propositional predicates [Bradley 2011; Eén et al. 2011; Hoder and Bjørner 2012] or predicate abstraction [Itzhaky et al. 2014]. We use Example 1.2 as a running example throughout this section.

We note that several flavors of PDR have been developed in the past years, including both particular implementations [Bradley 2011; Eén et al. 2011] and abstract presentations [Hoder and Bjørner

[3] Any formula containing ite can be translated to an equivalent first-order formula using only standard connectives; however, SMT-LIB-compliant solvers natively support ite [Barrett et al. 2010] so this translation is not needed.

$$
\begin{array}{rcl}
\mathit{wlp}[\![x.\mathsf{n} := \mathsf{null}]\!]Q & \stackrel{\text{def}}{=} & Q\begin{bmatrix}\mathrm{ite}(\psi_{\mathsf{km}}(x,x), \mathit{null}, m_s)/m_s,\\ \mathrm{ite}(\psi_{\mathsf{km}}(x,x), \mathit{null}, m_t)/m_t,\\ k^*(\alpha,\beta) \land (\neg k^*(\alpha,x) \lor k^*(\beta,x)) \lor\\ \quad(\psi_{\mathsf{km}}(\alpha,\beta) \land \neg k^*(\alpha,x) \land k^*(\beta,x) \land x \neq m_s)/k^*(\alpha,\beta)\end{bmatrix}\\
\mathit{where}\ \ \psi_{\mathsf{km}}(s,t) & \stackrel{\text{def}}{=} & m_s \neq \mathit{null} \land k^*(s,m_s) \land k^*(m_t,t)\\
\mathit{wlp}[\![x.\mathsf{n} := y]\!]Q & \stackrel{\text{def}}{=} & (k^*(y,x) \to m_s = \mathit{null}) \land\\
& & Q\begin{bmatrix}\mathrm{ite}(k^*(y,x), x, m_s)/m_s,\\ \mathrm{ite}(k^*(y,x), y, m_t)/m_t,\\ k^*(\alpha,\beta) \lor\\ \quad(y \neq \mathit{null} \land \neg k^*(y,x) \land k^*(\alpha,x) \land k^*(y,\beta))/k^*(\alpha,\beta)\end{bmatrix}\\
\mathit{wlp}[\![x := y.\mathsf{n}]\!]Q & \stackrel{\text{def}}{=} & \forall\alpha.\, \varphi_{\mathsf{n}}(y,\alpha) \to Q[\alpha/x]\\
\mathit{where}\ \ \varphi_{\mathsf{n}}(s,t) & \stackrel{\text{def}}{=} & \begin{cases} t = m_t & s = m_s\\ (k^+(s,t) \land \forall\gamma.\, k^+(s,\gamma) \to k^*(t,\gamma)) \lor & \\ \quad(t = \mathit{null} \land \forall\gamma.\, \neg k^+(s,\gamma)) & \mathit{otherwise}\end{cases}
\end{array}
$$

Table IV: Modified *wlp* rules for handling (restricted) cyclic lists.

2012]. In this paper, we use a deterministic implementation of PDR that employs recursion instead of the more commonly used *obligation queue*. This allows us to simplify the presentation and focus on the unique aspects of $\mathsf{PDR}^{\forall}$ at the expense of precluding discussion of several optimizations.

Requirements. We require that the transition relation ρ, as well as the *Init* and *Bad* conditions, are expressible in a logic $\mathcal{L}$, which is a fragment of first-order logic. (We can partly handle transitive closure using the approach of [Itzhaky et al. 2013]. See Section 3.) We assume that $\mathcal{L}$ is closed under conjunction and contains all universal and existential formulae. 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 $\mathit{model}(\psi)$, which returns a *finite* model σ of ψ if such a model exists and None otherwise. *EPR* (and EA^R) is an example of a logic $\mathcal{L}$ that can be used to express transition systems and properties. In particular, as $\mathcal{P}$ is the negation of *Bad*, it has to be described by a negation of an *EPR* (or EA^R) formula, that is $\mathcal{P}$ is a $\forall^*\exists^*$ formula.

4.1. 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* [Chang and Keisler 1990] 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 $\mathit{Diag}(\sigma)$, is an existential cube which describes explicitly the relations between all the elements of the model.[4]

Definition 4.1 (*Diagrams*). Given a *finite* model $\sigma = (D, \mathcal{I})$ over alphabet $\mathcal{V}$, the *diagram* of σ, denoted by $\mathit{Diag}(\sigma)$, is a closed formula over alphabet $\mathcal{V}$ which denotes the set of models in which σ can be isomorphically embedded. $\mathit{Diag}(\sigma)$ is constructed as follows.
— For every element $e_i \in D$, a fresh variable x_{e_i} is introduced.

[4]Def. 4.1, as well as the property formulated by Lemma 4.5, are an adaptation of the standard model-theoretic notion of a diagram [Chang and Keisler 1990].

— $\varphi_{distinct}$ 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.

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

Intuitively, one can think of *Diag*(σ) 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. Note that *Diag*(σ) is well defined, because we consider only models with a finite domain.

Example 4.2. The diagram of σ_b, depicted in Figure 3(σ_b), is

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

The first line records the fact that the domain 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 ok and n^* in σ_b.

We say that $\sigma_1 = (D_1, \mathcal{I}_1)$ is a *substructure* of $\sigma_2 = (D_2, \mathcal{I}_2)$ if $D_1 \subseteq D_2$ and for every $v \in \mathcal{V}$, $\mathcal{I}_1(v)$ is the restriction of $\mathcal{I}_2(v)$ to D_1. The following lemma is well known:

LEMMA 4.3. *$\sigma' \models$ Diag(σ) iff σ is isomorphic to a substructure of σ'.*

That is, the diagram of σ abstracts away the exact number of elements in the domain of σ and as such, provides a natural abstraction of states.

Example 4.4. In Figure 3, several models of *Diag*(σ_b) are depicted. For clarity, the edges drawn correspond to n-links extracted from n^* of each structure using φ_{n} (recall from Table II). Note that all of them contain σ_b as a substructure. For example, in σ_b^1, there is an additional element (v_3) representing a node pointing to the head of the list, as well as an additional element (v_4) representing an additional list with a single element. In σ_b^2, there is an additional element (v_3) representing an additional node in the list between h and j (represented by elements v_0 and v_1 respectively). To see why σ_b^2 contains σ_b as a substructure, recall that the vocabulary contains n^*, and not n itself; while no n-annotated edge appears in σ_b^2 from v_0 to v_1, $n^*(v_0, v_1)$ does hold in σ_b^2 as well due to the transitive nature of n^*. Similarly, σ_b^3, which represents a list that contains two additional nodes between h and j, also contains σ_b as a substructure. As these examples demonstrate, for linked list programs modelled with n^*, the diagram-based abstraction allows us to "forget" the exact length of list segments.

The following property of diagrams will be useful in the sequel.

LEMMA 4.5. *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 Diag$(\sigma) \Rightarrow \varphi$.*

Stated differently, *Diag*(σ) is the strongest existentially quantified formula that has σ as a model. Semantically, Lemma 4.5 means that for any models σ and σ' such that $\sigma' \models$ *Diag*(σ) if $\sigma \models \varphi$ then $\sigma' \models \varphi$. 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 the (closed existential)

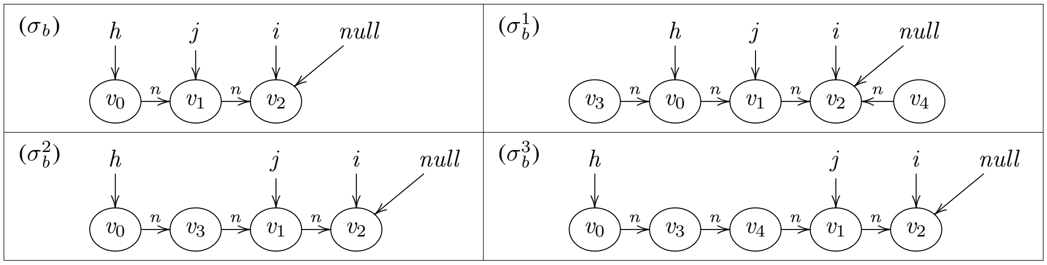

Fig. 3: Graphical depiction of the model σ_b and models σ_b^1, σ_b^2, σ_b^3 of $\mathit{Diag}(\sigma_b)$.

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. This property of the diagrams will allow PDR$^\forall$ to also prove absence of universal invariants (see Prop. 5.6).

4.2. 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 closed 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 4.6. Let $TS = (\mathit{Init}, \rho)$ be a transition system and $\mathcal{P}$ a safety property. A sequence $\langle F_0, F_1, \ldots, F_N\rangle$ of closed formulae 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 $0 \leq i \leq N$.

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.

4.3. Iterative Construction of an Approximate Reachability Sequence

PDR$^\forall$ is an iterative algorithm. At every iteration, the algorithm either strengthens the Nth frame, if it contains a bad state, or otherwise starts to develop the $N{+}1$st frame. 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

Algorithm 1: PDR$^\forall$ $(Init, \rho, Bad)$

1 **if** $SAT(Init \wedge Bad)$ **then**
2   **exit** invalid: $model(Init \wedge Bad)$
3 $F_0 := Init$
4 $F_1 := true$
5 $N := 1$
6 **while** true **do**
7   **if there exists** $0 \leq j < N$
        such that $F_{j+1} \Rightarrow F_j$ **then**
8   **return** valid
9   **if** $\neg SAT(F_N \wedge Bad)$ **then**
10         $F_{N+1} := true$
11         $N := N + 1$
12   **else**
13         $\sigma_b := model(F_N \wedge Bad)$
14   *block*(N, σ_b)

Algorithm 2: *block*(j, σ)

21 $\varphi = Diag(\sigma)$
22 **if** $(j = 0) \vee (j = 1 \wedge SAT(\varphi \wedge Init))$ **then**
23   *analyzeCEX*(j, N)
24 **while** $SAT(F_{j-1} \wedge \rho \wedge (\varphi)')$ **do**
25     $\sigma_a = reduct_{\mathcal{V}}(model(F_{j-1} \wedge \rho \wedge (\varphi)'))$
26   *block*$(j - 1, \sigma_a)$
27 **for** $i = 0 \ldots j$ **do**
28     $F_i := F_i \wedge \neg\varphi$

Algorithm 3: *analyzeCEX*(j, N)

31 **if** $j = 0 \wedge$ **there exists** $\sigma_0, \ldots, \sigma_N$ **such that**
32     $\sigma_0 \models Init$
33     $(\sigma_i, \sigma_{i+1}) \models \rho$ for every $0 \leq i < N$, and
34     $\sigma_N \models Bad$
35 **then exit** invalid: $\sigma_0, \ldots, \sigma_N$
36 **else exit** no universal invariant exists

(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 $true$ (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.[5]

Example 4.7. When analyzing the running example, our algorithm discovers that state σ_b, shown in Figure 4 (as well as in Figure 3), 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

[5]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) [Bradley 2011].

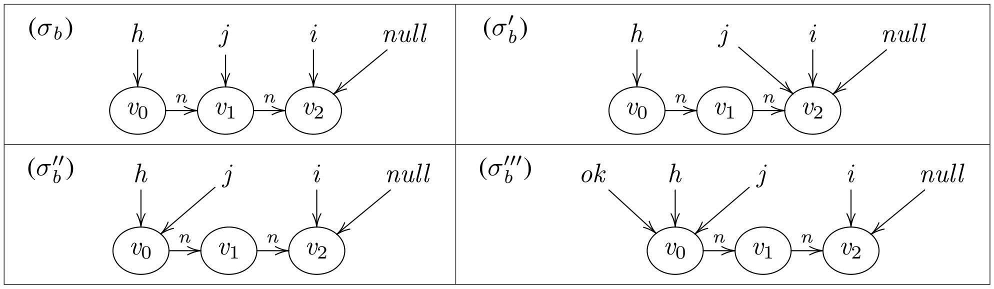

Fig. 4: Graphical depiction of models found during the analysis of the running example.

iterations find σ'_b, depicted in Figure 4, 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 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, 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 4.5, 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 equivalent to checking if $\varphi \wedge Init$ is satisfiable. Note that if we reached the initial frame, i.e., if $j = 0$, then $\sigma \models Init$, hence the above formula is guaranteed to be satisfiable. Explicitly checking that $\varphi \wedge Init$ is satisfiable is required only at the second frame, i.e, if $j = 1$ (see proof in Section 5.1):

LEMMA 4.8. *For every $1 < j \leq N$, when block(j, σ) is called, $F_i \Rightarrow \neg Diag(\sigma)$ for every $i \leq j - 1$. In particular, $Init \Rightarrow \neg Diag(\sigma)$.*

If the algorithm finds an *adverse initial state*, i.e., an initial state satisfying φ, (line 22),[6] 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),[7] 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). Similarly to traditional PDR, the last step is done in order to maintain the inclusion property of frames (Def. 4.6 (ii)). Moreover, this update maintains all the properties of Def. 4.6, in particular, $F_i \wedge \rho \Rightarrow (F_{i+1})'$ (iii) is preserved despite conjoining

[6]If $Init$ is a universal formula, then Lemma 4.8 holds for $j = 1$ as well, hence $j = 1 \wedge SAT(\varphi \wedge Init)$ never holds, and its check can be omitted (line 22).

[7]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 4.8 for $j > 1$, and since it was checked for $j = 1$), but the strenthening of δ can make the satisfiability check cheaper.

$(F_{i+1})'$ with $(\neg\varphi)'$ because frame inclusion ensures that for every $i < j$, $F_i \wedge \rho \Rightarrow F_{j-1} \wedge \rho$ and the latter implies $(\neg\varphi)'$. Note that preserving the properties of the approximate reachability sequence means that after the update, F_i remains an over-approximation of $\mathcal{R}_i$. In the following, we refer to the exclusion of the states of φ as the *blocking* of (the diagram of) σ from frame F_j.

Example 4.9. In our running example, in the first iteration *block*$(1, \sigma_b)$ updates F_1^0 to $F_1^1 = true \wedge \neg Diag(\sigma_b)$. This excludes σ_b, but also all states where $i = null$, ok is empty, and j is n-reachable from h in *any* (nonzero) number of steps (e.g. the states σ_b^2 and σ_b^3 depicted in Figure 3). In later iterations *block* updates $F_1^2 = F_1^1 \wedge \neg Diag(\sigma_b')$, $F_1^3 = F_1^2 \wedge \neg Diag(\sigma_b'')$, and $F_1^4 = F_1^3 \wedge \neg Diag(\sigma_b''')$.

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 to $\mathcal{V}$ (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:

Definition 4.10 (*Abstract and Spurious Counterexamples*). A sequence of closed formulae $\langle \varphi_j, \varphi_{j+1}, \ldots, \varphi_N \rangle$ is an *abstract counterexample* if the formulae $\varphi_j \wedge Init$, $\varphi_N \wedge Bad$, and $\varphi_i \wedge \rho \wedge (\varphi_{i+1})'$, for every $i = j, \ldots, N-1$, are all satisfiable. The abstract counterexample is *spurious* if there exists no sequence of states $\langle \sigma_j, \sigma_{j+1}, \ldots, \sigma_N \rangle$ such that $\sigma_j \models Init$, $\sigma_N \models Bad$, 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 [Biere et al. 2003]. 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. 5.6).

Generalization of blocked diagrams. Rather than blocking a diagram φ from frames $0, \ldots, j$ by conjoining them with the clause $\neg\varphi$ (line 28), our implementation uses a minimal unsat core of $\psi = ((Init)' \vee (F_{j-1} \wedge \rho)) \wedge (\varphi)'$ to define a clause L which implies $\neg\varphi$ and is also disjoint from $Init$ and unreachable from F_{j-1}. Blocking is done by conjoining L with F_i for every $i \leq j$.[8]

5. CORRECTNESS

In this section we formalize the correctness guarantees of $\mathsf{PDR}^\forall$. We start by formalizing the invariants of the sequence of frames maintained by $\mathsf{PDR}^\forall$. We then prove that the output of $\mathsf{PDR}^\forall$ is correct.

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

5.1. Properties of the Frames Computed by PDR$^\forall$

The following lemma summarizes the invariants of the sequence of frames computed by PDR$^\forall$. It follows from a simple induction on the steps of PDR$^\forall$. These invariants are adopted from traditional PDR.

LEMMA 5.1. *Let $TS = (Init, \rho)$ be a transition system and $\mathcal{P}$ a safety property. For every $N \geq 1$, the sequence $\langle F_0, F_1, \ldots, F_N\rangle$ obtained by PDR$^\forall$ right before N is increased in line 11, is an approximate reachability sequence for TS and $\mathcal{P}$ (Def. 4.6). Further, in every other step of the while loop (Line 6), the sequence $\langle F_0, F_1, \ldots, F_N\rangle$ satisfies all the requirements of Def. 4.6, except for, possibly, requirement (iv) for $i = N$.*

In order to provide a slightly tighter characterization of the frames computed by PDR$^\forall$, which, unlike Lemma 5.1, is specific to PDR$^\forall$, we need the following definition.

Definition 5.2 (*Relaxed Traces*). Given a transition system $TS = (Init, \rho)$, we say that a sequence of states $\sigma_0, \ldots, \sigma_n$ is a *relaxed trace* of TS if for every $0 \leq i \leq n-1$, there exists σ'_{i+1} such that $(\sigma_i, \sigma'_{i+1}) \models \rho$ and $\sigma'_{i+1} \models (Diag(\sigma_{i+1}))'$.

For a transition system TS and a property $\mathcal{P}$, with $Bad = \neg\mathcal{P}$, we denote by B_i the set of states that can reach a bad state in at most i steps via a relaxed trace. That is, $\sigma \in B_i$ if and only if there exists a relaxed trace $\sigma_0, \ldots, \sigma_n$ such that $\sigma_0 = \sigma$, $\sigma_n \models Bad$, and $n \leq i$. In particular, $B_0 = \{\sigma \mid \sigma \models Bad\}$. We denote by B the set of states that can reach a bad state in some number of steps via a relaxed trace, i.e. $B = \bigcup_{i \geq 0} B_i$.

The following lemma states that the frames computed by PDR$^\forall$ do not contain states that lead to bad states via relaxed traces, where the length of the considered traces depends on the frame index.

LEMMA 5.3. *Let $TS = (Init, \rho)$ be a transition system and $\mathcal{P}$ a safety property. For every $N \geq 1$ and $0 \leq j \leq N-1$, if $\sigma \models F_j$ then $\sigma \notin B_{N-1-j}$.*

That is, F_j does not include any state that reaches a bad state via a relaxed trace in $N-1-j$ steps or less. Note that this lemma holds in particular when N is increased to $N+1$ (line 11), in which case it implies that for every $0 \leq j \leq N$, if $\sigma \models F_j$ then $\sigma \notin B_{N-j}$. The proof follows a simple induction on j that shows that if $\sigma \in B_{N-1-j}$, then it is excluded from F_j.

On the other hand, it is easy to prove by induction that every state that PDR$^\forall$ attempts to block in frame F_j is a state in B:

LEMMA 5.4. *Let $TS = (Init, \rho)$ be a transition system and $\mathcal{P}$ a safety property. For every $N \geq 1$ and $0 \leq j \leq N$, if block(j, σ) is called, then $\sigma \in B_{N-j}$.*

We can now return to the proof of Lemma 4.8.

PROOF OF LEMMA 4.8. Suppose $\sigma \models F_j$ is discovered in the backward traversal from F_N. By Lemma 5.4, $\sigma \in B_{N-j}$. Therefore, by Lemma 5.3, $\sigma \not\models F_{j-1}$. To complete the proof, recall that we consider $j > 1$. Therefore $j-1 > 0$, hence F_{j-1} is a universal formula. Therefore, by Lemma 4.5, $Diag(\sigma) \Rightarrow \neg F_{j-1}$, or equivalently, $F_{j-1} \Rightarrow \neg Diag(\sigma)$. Since $F_i \Rightarrow F_{j-1}$ for every $i \leq j-1$, we conclude that $F_i \Rightarrow \neg Diag(\sigma)$ for every $i \leq j-1$. □

Note that if *Init* is a universal formula, then the claim formulated by Lemma 4.8 holds for $j = 1$ as well. That is, when *block*$(1, \sigma)$ is called, $F_0 \Rightarrow \neg Diag(\sigma)$. In this case, the additional check of $(j = 1 \wedge SAT(\varphi \wedge Init))$ performed in line 22 of *block*(j, σ) is always false, and hence not needed.

5.2. Correctness of the Outcome of PDR$^\forall$

We recall that if 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 5.5. *Let $TS = (Init, \rho)$ be a transition system and let $\mathcal{P}$ be a safety property. If PDR$^\forall$ returns* valid *then TS satisfies $\mathcal{P}$. Further, if PDR$^\forall$ returns a counterexample, then TS does not satisfy $\mathcal{P}$.*

PROOF. 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. □

Note that the proof of Lemma 5.5 also implies that when PDR$^\forall$ determines that TS satisfies $\mathcal{P}$, it also obtains a *universal* inductive invariant. Namely, the frame F_i in which a fixpoint was identified comprises such an invariant. (Recall that each frame is a universal formula as it is obtained as a conjunction of negated diagrams.)

PROPOSITION 5.6. *Let $TS = (Init, \rho)$ be a transition system and let $\mathcal{P}$ be a safety property. If PDR$^\forall$ obtains a spurious counterexample $\langle \varphi_j, \ldots, \varphi_N \rangle$ then there exists no universal safety inductive invariant $\mathcal{I}$ for TS and $\mathcal{P}$ over the given vocabulary.*

In order to prove Prop. 5.6, we first show that if a universal inductive invariant exists for TS and $\mathcal{P}$, then no state that reaches a bad state via a relaxed trace satisfies the invariant:

PROPOSITION 5.7. *Let $TS = (Init, \rho)$ be a transition system and $\mathcal{P}$ a safety property, and let B be the set of states that reach a bad state via a relaxed trace (see Section 5.1). Let $\mathcal{I}$ be a universal inductive safety invariant for $\mathcal{P}$. For any $\sigma \in B$, we have $\sigma \models \neg\mathcal{I}$.*

PROOF. We show by induction on i that $\sigma \in B_i$ implies $\sigma \models \neg\mathcal{I}$. For the base case, $\sigma \in B_0$, we have $\sigma \models$ *Bad*. Since *Bad* $= \neg\mathcal{P}$ and $\mathcal{I} \Rightarrow \mathcal{P}$, we conclude that $\sigma \models \neg\mathcal{I}$.

For the inductive step, let $\sigma \in B_{i+1}$. By definition of B_{i+1}, there exist models σ_i and σ'_i such that $(\sigma, \sigma'_i) \models \rho \wedge (Diag(\sigma_i))'$. and $\sigma_i \in B_i$. Moreover, by the induction hypothesis, $\sigma_i \models \neg\mathcal{I}$. Since $\neg\mathcal{I}$ is an existential formula, this means by Lemma 4.5 that $Diag(\sigma_i) \Rightarrow \neg\mathcal{I}$. We conclude that $\rho \wedge (Diag(\sigma_{i+1}))' \Rightarrow \rho \wedge (\neg\mathcal{I})'$. Therefore, (σ, σ'_i) is also a model of the formula $\rho \wedge (\neg\mathcal{I})'$.

If we assume that $\sigma \models \mathcal{I}$, we would get that $\mathcal{I} \wedge \rho \wedge (\neg\mathcal{I})'$ is satisfiable, in contradiction $\mathcal{I}$ being inductive. Hence, $\sigma \models \neg\mathcal{I}$. □

PROOF OF PROP. 5.6. Assume that there exists a universal safety inductive invariant $\mathcal{I}$ over $\mathcal{V}$. By Lemma 5.4, for every state σ_i generated by PDR$^\forall$ at frame F_i, $\sigma_i \in B$. Hence by Prop. 5.7, $\sigma_i \models \neg\mathcal{I}$. This implies, by Lemma 4.5, that every diagram φ_i generated by PDR$^\forall$ at frame F_i is such that $\varphi_i \Rightarrow \neg\mathcal{I}$, and hence $\varphi_i \Rightarrow \neg$*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 $\varphi_j \wedge$ *Init* is satisfiable. □

Example 5.8. Procedure `traverseTwo()`, presented in Figure 5 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 6. 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. The model σ_2 is a successor of σ_1 and hence $\sigma_2 \models \mathcal{I}$. However, $\sigma_2 \not\models \mathcal{P}$, in contradiction to the property of a safety invariant. Indeed, when using PDR$^\forall$, the spurious counterexample $\langle \sigma_0, \sigma_1, \sigma_2 \rangle$ presented in Figure 6 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.

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

```
void traverseTwo(List g, List h) {
   i := g; j := 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. 5: A procedure that finds the last elements of two non-empty acyclic lists.

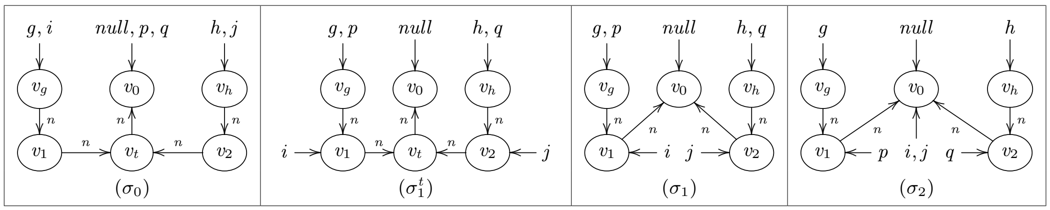

Fig. 6: A spurious counterexample found for procedure `traverseTwo()`, shown in Figure 5.

6. SUFFICIENT CONDITIONS FOR TERMINATION

The $\mathsf{PDR}^\forall$ algorithm is in general not guaranteed to terminate: it does not restrict the number of distinct existentially quantified variables in diagrams. This comes as a blessing for finding invariants with an arbitrary number of quantified variables, but is a curse when it comes to bounding the search space. There are however non-trivial classes of programs where it terminates. In particular, we show that $\mathsf{PDR}^\forall$ terminates for programs that manipulate singly linked lists. Inferring universal invariants for this class of programs was shown to be decidable [Padon et al. 2016, Theorem 4.2] using the naïve backwards reachability algorithm presented in Algorithm 4. The main result in this section bridges backwards reachability with $\mathsf{PDR}^\forall$, here formulated for a transition system $TS = (Init, \rho)$ and a property $\mathcal{P}$, with $Bad = \neg\mathcal{P}$:

PROPOSITION 6.1. *If backwards reachability, given by Algorithm 4, terminates on input transition system TS and property $\mathcal{P}$, then $\mathsf{PDR}^\forall$ terminates as well.*

In order to establish this result, we introduce a notion of *effective* encodings of the (backwards) reachable states, B (defined in Section 5.1). In a nutshell, a set of (backwards) reachable states B is effective for a class of transition systems and properties if the set of reachable states can be defined using a finitary existential formula, i.e., a formula that uses a finite number of quantified variables and contains a finite number of sub-formulae. This allows us to connect backwards reachability with $\mathsf{PDR}^\forall$ using the two lemmas:

LEMMA 6.2. *If Algorithm 4 terminates on TS and $\mathcal{P}$ then B is effective.*

and

LEMMA 6.3. *If B is effective for TS and $\mathcal{P}$ then $\mathsf{PDR}^\forall$ is guaranteed to terminate.*

We start with the description of Algorithm 4. It computes the complement of the least fixpoint of diagrams that can reach (via a relaxed trace) the bad states Bad. If the states represented by the resulting formula do not properly contain the initial states $Init$, then there are no universal inductive invariants, otherwise there is one.

Algorithm 4: Naïve Backward Reachability

$I := true$
while $SAT(I \wedge Bad)$ ***or*** $SAT(I \wedge \rho \wedge (\neg I)')$ **do**
  $(\sigma, \sigma') := model((I \wedge Bad) \vee (I \wedge \rho \wedge (\neg I)'))$
  $I := I \wedge \neg Diag(\sigma)$
if $SAT(Init \wedge \neg I)$ **then**
  return *no universal inductive invariant*
else
  return I *is a universal inductive invariant*

Next, let us define what we mean by a set of backward reachable states being *effective*. Recall that we use B_i to denote the set of states that can reach a bad state via a relaxed trace of TS in at most i steps, and B to denote the set of states that can reach a bad state via a relaxed trace in some number of steps. Thus, $B = \bigcup_{i \geq 0} B_i$ (see Section 5.1).

Definition 6.4. We say that B is *effective* if it can be described by an existential formula. That is, there exists an existential formula ψ_B such that $B = \{\sigma \mid \sigma \models \psi_B\}$.

Lemma 6.2 follows from the description of the naïve algorithm and the definition of effective states:

PROOF OF LEMMA 6.2. Suppose Algorithm 4 terminates, and let $\neg I$ be the negation of the obtained formula, I. Then $B = \{\sigma \mid \sigma \models \neg I\}$. Further, since $\neg I$ is equivalent to a disjunction of diagrams, it is an existential formula. This ensures effectiveness. □

Establishing Lemma 6.3 requires a bit more context. Recall that 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. For the termination argument, we place a stronger requirement: that finite model sizes are functions of the number of constants and existentially quantified variables. In other words, we require the existence of a function *bound* : $\mathbb{N} \to \mathbb{N}$ that given a formula ψ in $\mathcal{L}$, with at most n_ψ existentially quantified variables and constants, ψ has a finite model if and only if it has a model which contains no more than *bound*(n_ψ) elements. Note that such as bound only depends on the existential quantifiers and constants, and not, e.g., on the number of universal quantifiers. We note that *EPR* has such a bound; see Section 6.1.

Further, we also assume that $model(\psi)$ always returns a model of size at most *bound*(n_ψ). Assuming that $\mathcal{L}$ satisfies the additional requirement, and that all satisfiability checks performed by PDR$^\forall$ are in $\mathcal{L}$, we can prove Lemma 6.3.

PROOF OF LEMMA 6.3. The proof consists of two main arguments. First, we show that there exists a window of frames of a fixed length in which PDR$^\forall$ generates new clauses. Then, we show that the size of generated clauses is bounded by a function that depends on their distance from the last frame. As the latter is bounded by the length of the window, we obtain a bound on the size of clauses, which ensures termination.

We first note that since B is effective, there exists k such that $B = B_k$. This holds because every existential formula can be written as a finite disjunction of diagrams. Indeed, this can be achieved by converting the formula to DNF and performing for every existential cube a case splitting on whether the existentially quantified variables are distinct or not and on whether the missing instances of constants and relations appear negatively or positively. Therefore, effectiveness of B implies that there exists a finite set of states $\sigma_1, \ldots, \sigma_m$ such that $B = \{\sigma \mid \sigma \models \bigvee_{i=1}^m Diag(\sigma_i)\}$. For each $i = 1, \ldots, m$, let k_i be the length of a shortest relaxed trace leading from σ_i to a bad state, and let $k = \max_{i=1}^m k_i$. Since every state in B is a model of $Diag(\sigma_i)$ for at least one of the states σ_i, every such state starts a relaxed trace of length at most k_i to a bad state. Therefore, $B = B_k$. Hence for every $i > k$, $B_i = B_k$ as well.

We now turn to show that in each step, $\mathsf{PDR}^\forall$ only generates new clauses in the last k frames. Consider a fixed N. By Lemma 5.3, it holds that for every $j < N$, if $\sigma \models F_j$ then $\sigma \notin B_{N-1-j}$. By the choice of k, if $N-1-j \geq k$ then $B_{N-1-j} = B$. This means that for every $j \leq N-1-k$, if $\sigma \models F_j$ then $\sigma \notin B$. On the other hand, by Lemma 5.4, every state that $\mathsf{PDR}^\forall$ attempts to block in frame F_j is a state in B. As no such state exists for $j \leq N-1-k$, we conclude that no state is blocked at F_j for $j \leq N-1-k$. Therefore, new clauses, which result from blocked states, are only generated in the last $k+1$ frames F_{N-k} to F_N. (They are of course pushed backwards once generated.)

We now show that we can bound the size of models obtained by $\mathsf{PDR}^\forall$ in its backward traversal, and hence can bound the size of generated clauses. Let n_c denote the number of constants in $\mathcal{V}$, n_{Bad} denote the number of existential quantifiers in Bad and n_ρ denote the number of existential quantifiers in ρ. We show by induction on j that we can bound the size of models obtained by $\mathsf{PDR}^\forall$ in frame $N-j$ by a function of n_c, n_{Bad} and n_ρ (i.e., the bound does not depend on additional parameters such as N). For the base case ($j=0$) recall that the backward traversal starts from a state $\sigma \models F_N \wedge Bad$. Since F_N is a universal formula, the properties of $\mathcal{L}$ ensure that the size of the obtained model is bounded by $\mathit{bound}(n_c + n_{Bad})$. For the induction step, when $\mathsf{PDR}^\forall$ makes a step backward from a diagram of a state σ using ρ, it uses the formula $F_{j-1} \wedge \rho \wedge (\varphi)'$, where $\varphi = \mathit{Diag}(\sigma)$. The only existential quantifiers in this formula result from ρ and φ, where the number of the latter is equal to the size of the domain of σ (by the construction of the diagram). Our assumptions on $\mathcal{L}$ and on $model(\psi)$ therefore ensure that the size of the domain of the obtained model is bounded by

$$\mathit{bound}(n_c + n_\rho + n) \tag{*}$$

where n is the size of σ. By the induction hypothesis, n is bounded by a function of n_c, n_{Bad} and n_ρ only, hence the claim follows.

We denote by f_j the function of n_c, n_{Bad} and n_ρ that provides a bound on the size of models obtained by $\mathsf{PDR}^\forall$ in frame $N-j$. Therefore, when $\mathsf{PDR}^\forall$ makes at most k backward steps from Bad, the number of elements in the obtained models is bounded by

$$max = \max_{j=0}^{k} f_j(n_c, n_{Bad}, n_\rho)\,.$$

This provides a bound on the number of elements in the models that $\mathsf{PDR}^\forall$ tries to block.

Altogether, we conclude that the clauses generated by $\mathsf{PDR}^\forall$ (blocked diagrams) have at most max quantifiers, which makes the potential number of clauses finite and hence ensures termination of $\mathsf{PDR}^\forall$. $\square$

6.1. Termination when Reasoning with Effectively Propositional Logic

If Init and ρ are *EPR* formulae and $\mathcal{P}$ is a universal formula, then all the satisfiability queries made by $\mathsf{PDR}^\forall$ are of *EPR* formulae. *EPR* has the finite model property, and its satisfiability is decidable. Further, *EPR* meets the additional requirement needed for termination, as an *EPR* formula ψ is satisfiable if and only if it has a satisfying model whose size is bounded by n_ψ, where n_ψ is the number of constants and existentially quantified variables in ψ. That is, for every *EPR* formula ψ, $\mathit{bound}(n_\psi) = n_\psi$. The existence of this bound is a well-known property of the Bernays-Schönfinkel-Ramsey class of first-order formulae [Börger et al. 2008].

Note that in this case, the bound on the number of quantifiers in clauses generated by $\mathsf{PDR}^\forall$ in the last $k+1$ frames can be explicitly expressed as $(k+1)\cdot n_c + k \cdot n_\rho + n_{Bad}$, where n_c denotes the number of constants in $\mathcal{V}$, n_{Bad} denotes the number of existential quantifiers in Bad and n_ρ denotes the number of existential quantifiers in ρ. This can be proven by induction on k using the bound (*).

In [Padon et al. 2016], it is shown that Algorithm 4 is guaranteed to terminate in the case where the substructure relation is a well-quasi-order on the set of states[9]. This is the case for programs manipulating singly linked lists which are modeled in *EPR* using a single transitive reflexive binary relation for n^* (axiomatized as in [Itzhaky et al. 2013]), any number of constants and any number of unary relations (but no additional binary or higher-arity relations) [Padon et al. 2016]. We therefore conclude that $\mathsf{PDR}^\forall$ also terminates on such programs.

7. 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, 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 as explained in Section 3.

We note that the use of EA^R is key for the ability of $\mathsf{PDR}^\forall$ to infer inductive invariants of list-manipulating programs through generalization of particular counterexamples: At first blush, one might expect the tool to keep enumerating formulae about lists of every possible length. Luckily, EA^R specifies transition relations and properties using n^* instead of n. This formulation gives a natural abstraction for the lengths of list segments. In combination with generalization, via diagrams and unsat cores, this allows inferring clauses that apply to many lengths. In particular, we can reason about reachability without having to enumerate all the possible lengths. For example, consider Example 4.7: when blocking the diagram of the state σ_b presented in Figure 4, we also block all lists in which j is reachable from h in any number of steps.

Benchmarks. We implemented[10] $\mathsf{PDR}^\forall$ 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 [Cormen et al. 2009], and a union-find algorithm [Cormen et al. 2009]. 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 4.4.0 (build hashcode 0482e7fe727c) [de Moura and Bjørner 2008] with the default settings to check satisfiability of EPR formulae. Table V summarizes our experimental results.

(a) Verification. Our analyzer successfully verified(i) *memory safety*, i.e., the absence of null-dereferences and of memory leaks, (ii) *preservation of data-structure integrity*, meaning that the procedure never creates cycles in the list (for acyclic list manipulating procedures) or that the cyclic structure is preserved (for procedures operating on cyclic lists), and (iii) *functional correctness*, for several singly- and doubly-linked list manipulating procedures, and procedures operating with (restricted) cyclic lists. The precondition says that the expected input is a (possibly empty) (a-)cyclic 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. We encode the instructions `new` or `free` by means of $alloc(\cdot)$ by updating it accordingly in the transition relation. The absence of memory leaks then is formulated, informally, by saying that all non-reachable elements do not satisfy $alloc$: $\forall \alpha.\ alloc(\alpha) \rightarrow \bigvee_{x \in PVar} n^*(x, \alpha)$, where $PVar$ denotes the set of variables in the program. Additionally, we used auxiliary (ghost) constants and predicates and to mark the elements of the input list and record the reachability order between them. For example, we can specify a reversed order condition by introducing a ghost

[9]A *well-quasi-ordering* $\sqsubseteq$ on a set X is a preorder (i.e., a reflexive, transitive binary relation) such that any infinite sequence of elements $x_0, x_1, \ldots$ from X contains an increasing pair $x_i \sqsubseteq x_j$ with $i < j$.
[10]The implementation is available for download at `https://bitbucket.org/tausigplan/updr-distrib/`

Table V: Experimental results. Running time is measured in seconds. “[2014]” stands for results obtained using the analysis of [Itzhaky et al. 2014]. N denotes the highest index for a developed frame F_i. “# Z3” denotes the number of calls to Z3. **AF** denotes “Abstraction Failure” of [Itzhaky et al. 2014]. **TO** means timeout ($>$ 1 hr). (a) Correct programs; “# Cl. ($\forall$)” = number of ($\forall$-)clauses in the inferred invariant, after the elimination of redundant clauses. (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** [2014]			
(a) Verification	Time	N	# Z3	# Cl. ($\forall$)	Time	N	# Z3	# Cl. ($\forall$)	Time	N	# Z3	# Cl.
Singly-linked lists												
concat	1.2	3	56	7 (4)	0.5	3	42	4 (1)	**AF**			
create	1.4	3	62	7 (1)	0.8	3	40	5 (1)	1.8	3	28	3
delete	13.2	5	278	24 (11)	1.5	3	67	9 (1)	5.1	4	91	10
delete-all	10.3	5	255	15 (8)	0.6	3	40	3 (1)	1.6	3	56	5
filter	37.1	6	430	26 (16)	1.9	3	80	10 (3)	9.9	5	123	8
insert-at	1.8	3	69	8 (1)	1.5	4	60	9 (1)	6.6	4	130	14
insert	1.5	3	68	9 (2)	0.8	3	54	7 (1)	0.8	3	67	9
merge	244.6	7	1429	36 (21)	10.7	6	260	13 (1)	**AF**			
reverse	19.7	5	289	13 (9)	2.7	5	114	5 (1)	5.0	6	246	4
split	178.0	8	1079	33 (17)	5.9	5	146	11	7.0	5	97	9
uf-find	43.3	8	590	25 (18)	3.1	8	144	4	6.1	10	266	8
uf-union	178.8	7	1189	28 (13)	92.1	9	974	20 (5)	**TO**			
Sorted singly-linked lists												
sorted-insert	3.9	3	85	13 (5)	1.2	3	56	8 (1)	6.8	3	63	8
sorted-merge	400.3	8	1535	34 (21)	29.1	6	414	11 (2)	**AF**			
bubble-sort	90.5	11	904	23 (9)	2.1	5	61	6 (2)	2.3	4	34	3
insertion-sort	1681.1	13	4601	48 (25)	100.2	11	1220	35 (5)	**TO**			
Doubly-linked lists												
create	5.7	5	139	7 (3)	3.0	5	104	7 (2)	60.9	4	99	8
delete	3.1	4	90	10 (5)	0.8	3	36	5 (2)	38.5	4	78	8
insert-at	4.8	4	132	17 (8)	1.7	3	64	10 (3)	141.4	5	151	12
Composite linked-list structures												
nested-flatten	564.6	17	3019	39 (24)	169.5	17	1810	22 (11)	**AF**			
nested-split	660.5	9	1144	27 (21)	6.8	4	143	9 (1)	**AF**			
overlaid-delete	188.5	6	1054	27 (4)	101.0	7	843	24 (2)	**TO**			
Restricted cyclic singly-linked lists												
is-cycle	1.5	4	81	6	0.1	2	3	0	0.1	2	3	0
last	4.6	6	143	8 (1)	1.6	4	56	5 (1)	10.2	8	155	5
unchain	823.8	9	1986	50 (30)	0.1	2	3	0	0.1	2	3	0
insert	91.2	5	208	18 (6)	9.6	5	101	11 (2)	405.1	5	127	10
delete	8.2	4	138	16 (4)	5.1	4	95	10 (1)	55.4	3	81	13
reverse	253.9	8	1202	24 (10)	20.2	8	319	13 (4)	84.1	7	421	12

(b) Absence of a universal invariant	Description	Time	N	Z3
shared-tail	See Example 5.8	3.6	2	42
comb	See Section 7(b)	2	3	52

(c) Bug finding	Bug description	Time	N	Z3	C.e. size
insert-at	Precondition is too weak (omitted $e \neq \texttt{null}$)	0.4	1	11	4
filter	Forgot a corner case where $\neg ok(h)$	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

predicate $\underline{n}^*$ for the initial reachability, and using the following formula as the post-condition: $\forall\alpha\beta.\ \underline{n}^*(h,\alpha) \wedge \underline{n}^*(\alpha,\beta) \leftrightarrow n^*(\beta,\alpha) \wedge n^*(\alpha,h)$.

Our current implementation uses a fault avoidance version of *wlp* which ensures that the programs do not perform null-pointer dereferences and preserve the data structure integrity. Checking for the absence of memory leaks is done by adding the aforementioned formula to the post-condition. Thus, the user is required only to come up with the post-conditions pertaining to functional correctness.

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()`

Name	Formula
$x\langle f\rangle y$	$\varphi_f(x, y)$ (see Table II)
$f.ls\,[x, y]$	$\forall\alpha, \beta.\, f^*(x, \alpha) \wedge f^*(\alpha, y) \wedge f^*(\beta, \alpha) \rightarrow (f^*(\beta, x) \vee f^*(x, \beta))$
$f.stable(h)$	$\forall\alpha.\, f^*(h, \alpha) \rightarrow alloc(\alpha)$
$f/b.rev\,[x, y]$	$\forall\alpha, \beta. \begin{pmatrix} \alpha \neq null \wedge \beta \neq null \wedge \\ f^*(x, \alpha) \wedge f^*(\alpha, y) \wedge f^*(x, \beta) \wedge f^*(\beta, y) \end{pmatrix} \rightarrow (f^*(\alpha, \beta) \leftrightarrow b^*(\beta, \alpha))$
$f.sorted\,[x, y]$	$\forall\alpha, \beta. \begin{pmatrix} \alpha \neq null \wedge \beta \neq null \wedge \\ f^*(x, \alpha) \wedge f^*(\alpha, \beta) \wedge f^*(\beta, y) \end{pmatrix} \rightarrow \mathrm{dle}(\alpha, \beta)$

Table VI: Instrumentation predicates that were manually provided in the previous work [Itzhaky et al. 2014]. f and b denote pointer fields. dle is an uninterpreted predicate that denotes a total order on the data values; its semantics is enforced by an appropriate total-order background theory.

sort their input lists. We represented the order on data elements by a binary predicate together with the appropriate axioms. Using this predicate we express sortedness, without directly considering the data.

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 *ok* into a sublist. `flatten()` takes a nested list and flattens it by concatenating its sublists. We also verify the union-find algorithm. E.g., for compressing `find()` operation, we prove that it maintains the reachability between every node and its root and preserves the elements.

Finally, we verified several procedures manipulating cyclic linked lists that contain not more than one cycle. `is-cycle()` checks if a given list is cyclic. `unchain()` breaks every `n`-edge of a given cyclic list. `delete()` deletes a given element from an input cyclic list. `insert()` insert a given element into a given sorted cyclic list while preserving the sortedness.

We compared our results to [Itzhaky et al. 2014], where EA^R was used to verify properties of list-manipulating programs with PDR, using the human-supplied (universally-quantified) abstraction predicate templates given in Table VI. The templates are parameterized for arbitrary pointer fields f and b and program variables x, y, and h. They have the following meanings

— $x\langle f\rangle y$ — x is the immediate f-successor of y;
— $f.ls\,[x, y]$ — the list segment from x to y is not shared by any heap location;
— $f.stable(h)$ — h is the head of a list containing only allocated objects;
— $f/b.rev\,[x, y]$ — field b is a back-pointer for field f in list segment from x to y;
— $f.sorted\,[x, y]$ — the list segment from x to y is sorted.

Note that some of the predicates are far from trivial, and some are specific for one or two examples. Our technique was able to discover similar properties without the need for manually provided instrumentation predicates. We note that [Itzhaky et al. 2014] 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 5.8, does not have a universal invariant. We applied our tool to procedure `comb()`, which for a given list `h` and every element `a` of `h` allocates a new element `b` and places a pointer `p` from `a` to `b`, 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.

In our experience, the tool works well when the body of the loop is simple, even if the resulting invariants are not small. However, as the loop's body becomes more complicated, in particular containing conditional statements, the first-order formula describing the transition system can grow quite large, which has an adverse effect on scalability. This is most evident in the verification of `insertion-sort()`, where a nested loop was encoded using a single loop and some additional control statements.

8. RELATED WORK

Synthesizing quantified invariants has received significant attention. Several works have considered discovery of quantified predicates, e.g., based on counterexamples [Das and Dill 2002] or by extension of predicate abstraction to support free variables [Flanagan and Qadeer 2002; Lahiri and Bryant 2007]. 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* [Garg et al. 2013a; Garg et al. 2013b] or ones tailored to Presburger arithmetic with arrays [Dillig et al. 2010], instantiating quantifier templates [Bjørner et al. 2013; Srivastava and Gulwani 2009], or applying symbolic proof techniques [Hoder et al. 2011].

Other works aim to identify loop invariants *given* a set of predicates as candidate ingredients. Houdini [Flanagan and Leino 2001] is the first such algorithm of which we are aware. Santini [Thakur et al. 2015; Thakur et al. 2013] is a recent algorithm which is based on full predicate abstraction. In the context of IC3, predicate abstraction was used in [Birgmeier et al. 2014; Cimatti et al. 2014; Itzhaky et al. 2014], the last of which specifically targeting shape analysis. The above require a a predefined set of predicates, and are therefore less automatic than our approach, since the diagrams provide an "on-the-fly" abstraction mechanism. A CEGAR-based approach for inferring predicates, is shown in [Podelski and Wies 2010]. They go beyond the usual state predicates by introducing unary predicates that range over heap locations. This allows to infer quantified properties and permits lazy refinement to improve performance, but does not provide completeness guarantees. In addition, the inferred invariants have quantifier-nesting depth of one, whereas $\mathsf{PDR}^{\forall}$ allows for arbitrary nesting depth.

PDR has been shown to work extremely well in other domains, such as hardware verification [Bradley 2011; Eén et al. 2011]. Subsequently, it was generalized to software model checking for program models that use linear real arithmetic. The approach in [Hoder and Bjørner 2012] uses conflict-based projection for linear real arithmetic, while [Cimatti and Griggio 2012] employs a quantifier-elimination procedure for linear real arithmetic to provide an approximate pre-image operation. Finally, [Komuravelli et al. 2014] uses model-based projection to extract under-approximations of pre-images. 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 [Alberti et al. 2012] system is relevant. It is related to MCMT and Cubicle [Ghilardi and Ranise 2010b; Ghilardi and Ranise 2010a; Conchon et al. 2013; Conchon et al. 2012]. 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. More generally, our use of diagrams lazily produces finite state abstractions of array-based systems. Lazy abstraction was applied in [Henzinger et al. 2002], and shown to terminate when the abstraction structures satisfy the ascending-chain condition, e.g., that every chain of abstractions is well-founded. In [Jhala and McMillan 2006], various abstraction domains are considered based on interpolation where termination of the interpolation process can be enforced by limiting the creation of atomic formulae to use only existing sub-terms. In contrast, our termination argument is based on well-quasi-orders.

The logic used by our implementation has limited capabilities to express properties of list segments that are not pointed to by variables [Itzhaky et al. 2014]. This is similar to the self-imposed limitations on expressibility used in a number of past approaches, including (a) canonical abstraction [Sagiv et al. 2002]; (b) a prior method for applying predicate abstraction to linked lists [Manevich et al. 2005]; (c) an abstraction method based on "must-paths" between nodes that are either pointed to by variables or are list-merge points [Lev-Ami et al. 2006]; and (d) domains based on separation logic's list-segment primitive [Distefano et al. 2006; Berdine et al. 2007] (i.e., "$\mathrm{ls}[x, y]$" asserts the existence of a possibly empty list segment running from the node pointed to by x to the node pointed to by y). Decision procedures have been used in previous work to compute the best transformer for individual statements that manipulate linked lists [Yorsh et al. 2004; Podelski and Wies 2010].

Several logics used for heap verification have decision procedures obtained by reduction to traditional logics. STRAND [Madhusudan and Qiu 2011; Madhusudan et al. 2011] has the ability to reason about heap data structures and the data they contain, using MSO-defined relations over trees to describe heaps. It has a decidable fragment inspired by EPR where universally quantified variables can be used only in so-called *elastic relation*. Essentially, elasticity is a generalization of transitive closure. STRAND is similar to the earlier PALE [Møller and Schwartzbach 2001], which also translates reachability properties to MSO, and is based on *graph types* [Cook and Oppen 1975]. STRAND does not have an invariant inference mechanism, thus it can be interesting to use STRAND as the logic $\mathcal{L}$ in $\mathsf{PDR}^{\forall}$. The logic CSL [Bouajjani et al. 2009] has a similar flavor to STRAND, with similar sort restrictions on the syntax, but generalizes to handle doubly linked lists, and allows size constraints on structures. Its decidability is obtained by proving a small model property and via a reduction to first order logic.

A logic for reasoning about (cyclic) singly-linked lists is proposed by [Rakamaric et al. 2007]. The logic contains transitive closure of a single link function symbol, and has a decision procedure based on custom inference rules. A logic for reasoning about list segments and data is given in [Lahiri and Qadeer 2008]. The logic, LISBQ, provides a ternary primitive $\cdot \rightarrow \cdot \rightarrow \cdot$ that corresponds to heap paths through *three* nodes. This allows reasoning about pointer cycles. The decision procedure is based on a custom proof system, and its termination relies on stratification of the sorts — a semantic property of the formulae. A previous work by the same authors [Lahiri and Qadeer 2006] proposes a translation to first-order logic, but requires manual instantiations of quantifiers.

Separation logic [Reynolds 2002] uses inductive predicates conjoined with the $*$-operator to describe unbounded heaps. It has been used as a basis for static shape analyses, e.g., [Distefano et al. 2006], and also has some decidable fragments, e.g., [Berdine et al. 2004]. The latter allows to describe heaps as a collection of separated list segments between program variables or existentially quantified ones. Our logic uses similar segments, however it does not express separation explicitly, but rather as a collection of properties that hold at all the heaps. TREX [Wies et al. 2011] and GRIT [Piskac et al. 2014a; Piskac et al. 2014b] are essentially similar decidable fragments of separation logic. The former has a decision procedure based on reduction to first order logic, and the latter was recently shown to be reducible to EPR. GRIT also allows to reason on numerical data stored in heap cells.

9. CONCLUSIONS

$\mathsf{PDR}^{\forall}$ is a combination of PDR/IC3 [Bradley 2011] with the model-theoretic notion of diagrams [Chang and Keisler 1990]. 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. Recent work [Frumkin et al. 2017] has extended $\mathsf{PDR}^{\forall}$ to interprocedural analysis, where procedure summaries [Reps et al. 1995] are inferred instead of invariants. The interprocedural version was used to verify correct use of iterators in Java programs. In the future, we plan to apply it in other contexts too, e.g., for the verification of network programs [ONF 2016].

Acknowledgments. We thank Mooly Sagiv and the reviewers of this paper and of the preliminary conference version [Karbyshev et al. 2015] for their 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

Aws Albarghouthi, Josh Berdine, Byron Cook, and Zachary Kincaid. 2015. Spatial Interpolants. *CoRR* abs/1501.04100 (2015). http://arxiv.org/abs/1501.04100

Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. 2012. SAFARI: SMT-Based Abstraction for Arrays with Interpolants. In *Proceedings of 24th International Conference on Computer Aided Verification (CAV '12)*, Vol. 7358. Springer. DOI:http://dx.doi.org/10.1007/978-3-642-31424-7_49

Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The Satisfiability Modulo Theories Library (SMT-LIB). (2010). http://www.smt-lib.org

Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. 2007. Shape Analysis for Composite Data Structures. In *Proceedings of 19th International Conference on Computer Aided Verification (CAV '07)*, Vol. 4590. Springer, 178–192. DOI:http://dx.doi.org/10.1007/978-3-540-73368-3_22

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2004. A Decidable Fragment of Separation Logic. In *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '04)*. Springer-Verlag, Berlin, Heidelberg, 97–109. DOI:http://dx.doi.org/10.1007/978-3-540-30538-5_9

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded model checking. *Advances in Computers* 58 (2003), 118–149.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Proceedings of 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '99)*. Springer, 193–207. DOI:http://dx.doi.org/10.1007/3-540-49059-0_14

Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. 2014. Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR). In *Proceedings of 26th International Conference on Computer Aided Verification (CAV '14)*, Vol. 8559. Springer, 831–848. DOI:http://dx.doi.org/10.1007/978-3-319-08867-9_55

Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. 2013. On Solving Universally Quantified Horn Clauses. In *Proceedings of 20th International Static Analysis Symposium (SAS '13)*, Vol. 7935. Springer, 105–125. DOI:http://dx.doi.org/10.1007/978-3-642-38856-9_8

Egon Börger, Erich Grädel, and Yuri Gurevich. 2008. *The Classical Decision Problem*. Springer-Verlag.

Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. 2009. A Logic-Based Framework for Reasoning about Composite Data Structures. In *Proceedings of 20th International Conference on Concurrency Theory (CONCUR 2009)*, Vol. 5710. Springer, 178–195. DOI:http://dx.doi.org/10.1007/978-3-642-04081-8_13

Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*, Vol. 6538. Springer, 70–87. DOI:http://dx.doi.org/10.1007/978-3-642-18275-4_7

Chen C. Chang and Howard J. Keisler. 1990. *Model Theory*. Elsevier Science.

Alessandro Cimatti and Alberto Griggio. 2012. Software Model Checking via IC3. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012)*, Vol. 7358. Springer, 277–293. DOI:http://dx.doi.org/10.1007/978-3-642-31424-7_23

Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. 2014. IC3 Modulo Theories via Implicit Predicate Abstraction. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*, Vol. 8413. Springer, 46–61. DOI:http://dx.doi.org/10.1007/978-3-642-54862-8_4

Edmund Clarke, Daniel Kroening, and Karen Yorav. 2003. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In *Proceedings of the 40th Annual Design Automation Conference (DAC '03)*. ACM, New York, NY, USA, 368–371. DOI:http://dx.doi.org/10.1145/775832.775928

Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. 2012. Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems — Tool Paper. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012)*, Vol. 7358. Springer, 718–724. DOI:http://dx.doi.org/10.1007/978-3-642-31424-7_55

Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. 2013. Invariants for finite instances and beyond. In *Proceedings of Conference on Formal Methods in Computer-Aided Design (FMCAD 2013)*. IEEE, 61–68. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679392

Stephen A. Cook and Derek C. Oppen. 1975. An Assertion Language for Data Structures. In *Proceedings of the 2Nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '75)*. ACM, New York, NY, USA, 160–166. DOI:http://dx.doi.org/10.1145/512976.512993

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL '77)*. ACM, 238–252. DOI:http://dx.doi.org/10.1145/512950.512973

Satyaki Das and David L. Dill. 2002. Counter-Example Based Predicate Discovery in Predicate Abstraction. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2002)*, Vol. 2517. Springer, 19–32. DOI:http://dx.doi.org/10.1007/3-540-36126-X_2

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, Vol. 4963. Springer, 337–340. DOI:http://dx.doi.org/10.1007/978-3-540-78800-3_24

Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Symbolic heap abstraction with demand-driven axiomatization of memory invariants. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*. ACM, 397–410. DOI:http://dx.doi.org/10.1145/1869459.1869493

Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2006. A Local Shape Analysis Based on Separation Logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006)*, Vol. 3920. Springer, 287–302. DOI:http://dx.doi.org/10.1007/11691372_19

Niklas Eén, Alan Mishchenko, and Robert K. Brayton. 2011. Efficient implementation of property directed reachability. In *Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD '11)*. FMCAD Inc., 125–134. http://dl.acm.org/citation.cfm?id=2157675

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of Symposium of Formal Methods Europe (FME 2001)*, Vol. 2021. Springer, 500–517. DOI:http://dx.doi.org/10.1007/3-540-45251-6_29

Cormac Flanagan and Shaz Qadeer. 2002. Predicate abstraction for software verification. In *Conference Record of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*. ACM, 191–202. DOI:http://dx.doi.org/10.1145/503272.503291

Asya Frumkin, Yotam M. Y. Feldman, Ondřej Lhoták, Oded Padon, Mooly Sagiv, and Sharon Shoham. 2017. Property Directed Reachability for Proving Absence of Concurrent Modification Errors. In *Proceedings of the 18th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2017)*. Springer. To appear.

Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2013a. Learning Universally Quantified Invariants of Linear Data Structures. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013)*, Vol. 8044. Springer, 813–829. DOI:http://dx.doi.org/10.1007/978-3-642-39799-8_57

Pranav Garg, P. Madhusudan, and Gennaro Parlato. 2013b. Quantified Data Automata on Skinny Trees: An Abstract Domain for Lists. In *Proceedings of the 20th International Symposium on Static Analysis (SAS 2013)*, Vol. 7935. Springer, 172–193. DOI:http://dx.doi.org/10.1007/978-3-642-38856-9_11

Silvio Ghilardi and Silvio Ranise. 2010a. Backward Reachability of Array-based Systems by SMT solving: Termination and Invariant Synthesis. *Logical Methods in Computer Science* 6, 4 (2010). DOI:http://dx.doi.org/10.2168/LMCS-6(4:10)2010

Silvio Ghilardi and Silvio Ranise. 2010b. MCMT: A Model Checker Modulo Theories. In *Proceedings of the 5th International Joint Conference on Automated Reasoning (IJCAR 2010)*, Vol. 6173. Springer, 22–29. DOI:http://dx.doi.org/10.1007/978-3-642-14203-1_3

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy abstraction. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*. ACM, 58–70. DOI:http://dx.doi.org/10.1145/503272.503279

Krystof Hoder and Nikolaj Bjørner. 2012. Generalized Property Directed Reachability. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT 2012)*, Vol. 7317. Springer, 157–171. DOI:http://dx.doi.org/10.1007/978-3-642-31612-8_13

Krystof Hoder, Laura Kovács, and Andrei Voronkov. 2011. Invariant Generation in Vampire. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*, Vol. 6605. Springer, 60–64. DOI:http://dx.doi.org/10.1007/978-3-642-19835-9_7

Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. 2013. Effectively-Propositional Reasoning about Reachability in Linked Data Structures. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013)*, Vol. 8044. Springer, 756–772. DOI:http://dx.doi.org/10.1007/978-3-642-39799-8_53

Shachar Itzhaky, Nikolaj Bjørner, Thomas W. Reps, Mooly Sagiv, and Aditya V. Thakur. 2014. Property-Directed Shape Analysis. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014)*, Vol. 8559. Springer, 35–51. DOI:http://dx.doi.org/10.1007/978-3-319-08867-9_3

Ranjit Jhala and Kenneth L. McMillan. 2006. A Practical and Complete Approach to Predicate Refinement. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006)*, Vol. 3920. Springer, 459–473. DOI:http://dx.doi.org/10.1007/11691372_33

Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. 2015. Property-Directed Inference of Universal Invariants or Proving Their Absence. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV 2015)*, Vol. 9206. Springer, 583–602. DOI:http://dx.doi.org/10.1007/978-3-319-21690-4_40

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014)*, Vol. 8559. Springer, 17–34. DOI:http://dx.doi.org/10.1007/978-3-319-08867-9_2

Shuvendu K. Lahiri and Randal E. Bryant. 2007. Predicate Abstraction with Indexed Predicates. *ACM Trans. Comput. Logic* 9, 1 (2007).

Shuvendu K. Lahiri and Shaz Qadeer. 2006. Verifying properties of well-founded linked lists. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*. ACM, New York, NY, USA, 115–126. DOI:http://dx.doi.org/10.1145/1111037.1111048

Shuvendu K. Lahiri and Shaz Qadeer. 2008. Back to the future: revisiting precise program verification using SMT solvers. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*. ACM, 171–182. DOI:http://dx.doi.org/10.1145/1328438.1328461

Tal Lev-Ami, Neil Immerman, and Shmuel Sagiv. 2006. Abstraction for Shape Analysis with Fast and Precise Transformers. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*, Vol. 4144. Springer, 547–561. DOI:http://dx.doi.org/10.1007/11817963_49

Harry R. Lewis. 1980. Complexity results for classes of quantificational formulas. *J. Comput. System Sci.* 21, 3 (1980), 317 – 353.

P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. 2011. Decidable logics combining heap structures and data. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*. ACM, New York, NY, USA, 611–622. DOI:http://dx.doi.org/10.1145/1926385.1926455

P. Madhusudan and Xiaokang Qiu. 2011. Efficient Decision Procedures for Heaps Using STRAND. In *Proceedings of the 18th International Symposium on Static Analysis (SAS 2011)*, Vol. 6887. Springer, 43–59. DOI:http://dx.doi.org/10.1007/978-3-642-23702-7_8

Roman Manevich, Eran Yahav, Ganesan Ramalingam, and Shmuel Sagiv. 2005. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2005)*, Vol. 3385. Springer, 181–198. DOI:http://dx.doi.org/10.1007/978-3-540-30579-8_13

Anders Møller and Michael I. Schwartzbach. 2001. The Pointer Assertion Logic Engine. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001)*. ACM, New York, NY, USA, 221–231. DOI:http://dx.doi.org/10.1145/378795.378851

ONF. 2016. (2016). http://opennetworking.org The Open Networking Foundation.

Oded Padon, Neil Immerman, Sharon Shoham, Aleksandr Karbyshev, and Mooly Sagiv. 2016. Decidability of inferring inductive invariants. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM, 217–231. DOI:http://dx.doi.org/10.1145/2837614.2837640

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014a. Automating Separation Logic with Trees and Data. In *Proceedings of 26th International Conference on Computer Aided Verification (CAV '14)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, 711–728. DOI:http://dx.doi.org/10.1007/978-3-319-08867-9_47

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014b. GRASShopper - Complete Heap Verification with Mixed Specifications. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*, Vol. 8413. Springer, 124–139. DOI:http://dx.doi.org/10.1007/978-3-642-54862-8_9

Andreas Podelski and Thomas Wies. 2010. Counterexample-guided focus. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*. ACM, 249–260. DOI:http://dx.doi.org/10.1145/1706299.1706330

Zvonimir Rakamaric, Jesse D. Bingham, and Alan J. Hu. 2007. An Inference-Rule-Based Decision Procedure for Verification of Heap-Manipulating Programs with Mutable Data and Cyclic Data Structures. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2007)*, Vol. 4349. Springer, 106–121. DOI:http://dx.doi.org/10.1007/978-3-540-69738-1_8

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*. ACM Press, 49–61. DOI:http://dx.doi.org/10.1145/199448.199462

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*. IEEE Computer Society, 55–74. DOI:http://dx.doi.org/10.1109/LICS.2002.1029817

Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (2002), 217–298. DOI:http://dx.doi.org/10.1145/514188.514190

Saurabh Srivastava and Sumit Gulwani. 2009. Program verification using templates over predicate abstraction. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009)*. ACM, 223–234. DOI:http://dx.doi.org/10.1145/1542476.1542501

A. Thakur, A. Lal, J. Lim, and T. Reps. 2013. *PostHat and All That: Attaining Most-Precise Inductive Invariants*. TR-1790. Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI.

Aditya V. Thakur, Akash Lal, Junghee Lim, and Thomas W. Reps. 2015. PostHat and All That: Automating Abstract Interpretation. *Electr. Notes Theor. Comput. Sci.* 311 (2015), 15–32. DOI:http://dx.doi.org/10.1016/j.entcs.2015.02.003

Thomas Wies, Marco Muñiz, and Viktor Kuncak. 2011. An Efficient Decision Procedure for Imperative Tree Data Structures. In *Proceedings of the 23rd International Conference on Automated Deduction (CADE-23)*, Vol. 6803. Springer, 476–491. DOI:http://dx.doi.org/10.1007/978-3-642-22438-6_36

Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA.

Greta Yorsh, Thomas W. Reps, and Shmuel Sagiv. 2004. Symbolically Computing Most-Precise Abstract Operations for Shape Analysis. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, Vol. 2988. Springer, 530–545. DOI:http://dx.doi.org/10.1007/978-3-540-24730-2_39

A. THE INVARIANT OBTAINED FOR SPLIT()

$$
\begin{array}{rcl}
I & = & \bigwedge_{i=0}^{32} L_i \qquad \text{where} \\
L_0 & = & (j \neq i) \vee (i = h) \\
L_1 & = & \forall\alpha.\ (ok(\alpha) \vee n^*(\alpha,k) \vee (g = \underline{h}) \vee \underline{n}^*(j,\alpha) \vee \neg\underline{n}^*(g,\alpha)) \\
L_2 & = & ((g = null) \vee \neg ok(g)) \\
L_3 & = & (\underline{n}^*(k,j) \vee \neg n^*(j,k)) \\
L_4 & = & ((h = i) \vee \underline{n}^*(j,i) \vee (null = i)) \\
L_5 & = & (\underline{n}^*(j,i) \vee (j = null) \vee (null = i)) \\
L_6 & = & \forall\alpha\beta.\ (n^*(\underline{h},\alpha) \vee \neg\underline{n}^*(\underline{h},\alpha) \vee \neg n^*(\underline{h},\beta) \vee \neg n^*(\alpha,\beta)) \\
L_7 & = & \forall\alpha.\ (\neg\underline{n}^*(\underline{h},\alpha) \vee ok(\alpha) \vee (g = null) \vee \underline{n}^*(g,\alpha)) \\
L_8 & = & \forall\alpha\beta.\ (\underline{n}^*(\alpha,\underline{h}) \vee \neg\underline{n}^*(\alpha,\beta) \vee n^*(\alpha,\beta) \vee \underline{n}^*(\alpha,k)) \\
L_9 & = & (\neg n^*(\underline{h},h) \vee \underline{n}^*(h,\underline{h})) \\
L_{10} & = & \forall\alpha.\ (\underline{n}^*(i,\alpha) \vee \neg\underline{n}^*(j,\alpha) \vee (\alpha = j) \vee \underline{n}^*(j,k)) \\
L_{11} & = & (ok(j) \vee (j = null)) \\
L_{12} & = & (n^*(h,j) \vee (j = null)) \\
L_{13} & = & \forall\alpha.\ (n^*(g,\alpha) \vee ok(\alpha) \vee \neg\underline{n}^*(\underline{h},\alpha) \vee (g = k) \vee \underline{n}^*(k,\alpha)) \\
L_{14} & = & ((\underline{h} = g) \vee (\underline{h} = h)) \\
L_{15} & = & \forall\alpha\beta.\ (\neg n^*(\alpha,\beta) \vee \underline{n}^*(\alpha,\beta)) \\
L_{16} & = & \underline{n}^*(g,k) \\
L_{17} & = & \forall\alpha.\ (\neg\underline{n}^*(\underline{h},\alpha) \vee n^*(\underline{h},\alpha) \vee \underline{n}^*(g,\alpha)) \\
L_{18} & = & \forall\alpha\beta.\ (\underline{n}^*(k,\beta) \vee \neg\underline{n}^*(\alpha,\beta) \vee n^*(\alpha,\beta) \vee \neg ok(\beta) \vee \neg\underline{n}^*(\underline{h},\alpha) \vee n^*(\alpha,k)) \\
L_{19} & = & \forall\alpha.\ (n^*(\underline{h},\alpha) \vee \neg\underline{n}^*(\underline{h},\alpha) \vee \underline{n}^*(h,\alpha)) \\
L_{20} & = & ((null = i) \vee \underline{n}^*(k,i) \vee (g = null)) \\
L_{21} & = & \forall\alpha.\ ((h = i) \vee \underline{n}^*(j,\alpha) \vee (k \neq null) \vee ok(\alpha) \vee \neg\underline{n}^*(h,\alpha)) \\
L_{22} & = & \forall\alpha.\ (n^*(\alpha,k) \vee \neg ok(\underline{h}) \vee \underline{n}^*(\alpha,g) \vee \underline{n}^*(j,\underline{h}) \vee \underline{n}^*(\alpha,j) \vee \neg\underline{n}^*(\alpha,k)) \\
L_{23} & = & \forall\alpha\beta.\ (\underline{n}^*(j,\alpha) \vee (\alpha = k) \vee \neg\underline{n}^*(k,\alpha) \vee \underline{n}^*(k,\beta) \vee \neg\underline{n}^*(\underline{h},\beta) \vee n^*(\beta,\alpha) \vee n^*(\beta,k)) \\
L_{24} & = & \forall\alpha.\ ((\alpha = k) \vee \underline{n}^*(i,\alpha) \vee \neg\underline{n}^*(k,\alpha) \vee \underline{n}^*(k,j)) \\
L_{25} & = & ((h = null) \vee \underline{n}^*(\underline{h},h)) \\
L_{26} & = & (\underline{n}^*(\underline{h},g) \vee (g = null)) \\
L_{27} & = & \forall\alpha.\ (\neg\underline{n}^*(j,\alpha) \vee \neg\underline{n}^*(\alpha,k) \vee n^*(\alpha,k) \vee \underline{n}^*(\alpha,j)) \\
L_{28} & = & \forall\alpha.\ (\neg n^*(g,\alpha) \vee \underline{n}^*(\alpha,g) \vee \neg ok(\alpha)) \\
L_{29} & = & ((\underline{h} = h) \vee (k \neq h)) \\
L_{30} & = & \forall\alpha\beta.\ (\neg\underline{n}^*(\alpha,\beta) \vee ok(\beta) \vee \underline{n}^*(j,\beta) \vee (\alpha = \beta) \vee \neg\underline{n}^*(k,j) \vee \neg\underline{n}^*(k,\alpha)) \\
L_{31} & = & (n^*(h,i) \vee (i = null)) \\
L_{32} & = & (\underline{n}^*(j,g) \vee (i \neq g) \vee (g = null))
\end{array}
$$

Fig. 7: The inferred invariant for split. It contains 33 clauses, of which 17 are universal.