We present Universal Property Directed Reachability (PDR\(^\forall\)), a semi-algorithm for automatic inference of invariants in a universal fragment of first-order logic. PDR\(^\forall\) is an extension of Bradley’s PDR/IC3 algorithm for inference of propositional invariants. 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. 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 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:
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

We present Universal Property Directed Reachability (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, 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.
Diagram Based Abstraction. Unlike previous work \cite{Itzhaky:2014:PRP:2803858.2803908,Albarghouthi:2015:FPD:2817501.2817510}, 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 $\varphi$ is valid, or construct a finite model $\sigma$ of $\varphi$. We then translate $\sigma$ into a diagram \cite{Chang:1990:AWZ:93536.93537} — a formula describing the set of models that extend $\sigma$ — and use the diagram to construct a universal clause to strengthen a candidate invariant.

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\textsuperscript{V} 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.

(a) A procedure that moves all the elements not satisfying $ok(\cdot)$ from list $h$ to list $g$ and its specification using pre- and 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 $h$ and $n^*(\cdot,\cdot)$ record the head of the original list and the reachability order between its elements.

\begin{verbatim}
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 }
  }
}
\end{verbatim}

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

Fig. 1: Motivating examples.

\begin{verbatim}
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
    }
  }
}
\end{verbatim}

requires:
\begin{align*}
g = \text{null} \land h = \text{null} \land (\forall x. y. n^*(x,y) \leftrightarrow n^*(x,y))
\end{align*}

ensures:
\begin{align*}
(\forall z. h ≠ \text{null} \land n^*(h,z) \rightarrow ok(z)) \land \\
(\forall z. g ≠ \text{null} \land n^*(g,z) \rightarrow \neg ok(z)) \land \\
(\forall z. z ≠ \text{null} \rightarrow (n^*(h,z) \leftrightarrow n^*(h,z) \lor n^*(g,z))) \land \\
(\forall x, y.
  n^*(h,x) \land n^*(x,y) \land ok(x) \land ok(y) \rightarrow n^*(x,y)) \land \\
(\forall x, y.
  n^*(h,x) \land n^*(x,y) \land \neg ok(x) \land \neg ok(y) \rightarrow n^*(x,y))
\end{align*}
Property-Directed Inference of Universal Invariants or Proving Their Absence

\[ I = L_1 \land L_2 \land L_3 \land L_4 \land L_5 \land L_6 \land L_7 \] where
\[ L_1 = i \neq h \land i \neq \text{null} \rightarrow n^*(j, i) \]
\[ L_2 = i \neq h \rightarrow \text{ok}(h) \]
\[ L_3 = n^*(h, j) \lor i \neq j \]
\[ L_4 = \forall x. i \neq h \land n^*(j, x) \land x \neq j \rightarrow n^*(i, x) \]
\[ L_5 = i \neq h \rightarrow \text{ok}(j) \]
\[ L_6 = \forall x. x = h \lor j = \text{null} \lor \neg n^*(h, x) \lor n^*(h, j) \lor \neg \text{ok}(j) \]
\[ L_7 = \forall x. j \neq \text{null} \land n^*(h, x) \land x \neq h \land \neg \text{ok}(x) \rightarrow n^*(j, x) \]

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, \( \text{PDR}^\forall \) iteratively constructs an increasing sequence of candidate inductive invariants \( F_0, \ldots, F_N \). Every \( F_i \) over-approximates the set \( R_i \) of states that can be reached by up to \( i \) execution steps from a given set of initial states. In every iteration, \( \text{PDR}^\forall \) uses SMT to check whether one of the candidate invariants became inductive. If so, then the program respects the desired property. If not, \( \text{PDR}^\forall \) iteratively strengthens the candidate invariants and adds new ones, guided by the considered property. Specifically, it checks if there exists a bad state \( \sigma \) which satisfies \( F_N \) but not the property. If so, we use SMT again to check whether there is a state \( \sigma_a \) in \( F_{N-1} \) that can lead to a state in the diagram \( \varphi \) of \( \sigma \) in one execution step. If no such state exists, the candidate invariant \( F_N \) can be strengthened by conjointing it with the negation of \( \varphi \). Otherwise, we recursively strengthen \( F_{i-1} \) to exclude \( \sigma_a \) from its over-approximation of \( R_{i-1} \). 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, \( \text{PDR}^\forall \) determines that no universal invariant strong enough to prove the desired property exists (see Lemma 4.5). We note that \( \text{PDR}^\forall \) 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 \( \text{ok} \) from the list pointed to by \( h \) to the list pointed by \( g \). \( \text{PDR}^\forall \) 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 \( \text{ok} \) are kept in \( h \); (iii) all the elements which do not satisfy \( \text{ok} \) are moved to \( g \); (iv) no new elements are introduced; and (v) stability, i.e., the reachability order between the elements satisfying \( \text{ok} \) 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 \( \text{ok} \) from the list pointed to by \( h \). Figure 2 shows the loop invariant inferred by \( \text{PDR}^\forall \) when it was asked to verify a simplified version of property (iii): all the elements which do not satisfy \( \text{ok} \) 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 \( \text{ok} \) must occur after \( j \).

Experimental Evaluation. We implemented \( \text{PDR}^\forall \) 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 PDR\textsuperscript{\textit{v}}, a pleasantly simple, yet surprisingly powerful, combination of PDR [Bradley 2011] with a strengthening technique based on diagrams [Chang and Keisler 1990]. PDR\textsuperscript{\textit{v}} 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 PDR\textsuperscript{\textit{v}} 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 PDR\textsuperscript{\textit{v}} terminates.
- We implemented PDR\textsuperscript{\textit{v}} on top of a decision procedure for the logic EA\textsuperscript{\textit{R}} [Itzhaky et al. 2013] 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/](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\textsuperscript{\textit{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\textsuperscript{1}first-order model \( (D, I) \) over a vocabulary \( \mathcal{V} \) which consists of constants and relation symbols, where \( D \) is the finite domain of the model, and \( I \) 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 \( \hat{\mathcal{V}} = \mathcal{V} \cup \mathcal{V}' \), 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 \( \sigma' = (D, I') \) over \( \mathcal{V}' \) describes a program state \( \sigma = (D, I) \), where \( I(v) = I'(v') \) for every symbol \( v \in \mathcal{V} \).

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

We often write a transition \( \hat{\sigma} \) as a pair of states \((\sigma_1, \sigma_2)\), such that \( \sigma_1 \) is the redact of \( \hat{\sigma} \) to vocabulary \( \mathcal{V} \), and \( \sigma_2 \) is the state described by the redact to \( \mathcal{V}' \). We say that \( \sigma_2 \) is a successor of \( \sigma_1 \), and \( \sigma_1 \) is a predecessor of \( \sigma_2 \).

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

\textsuperscript{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 $V$ used to denote the initial states of the program, and $\rho$ is a closed formula over $V$ used to denote its transition relation. A state $\sigma$ is initial if $\sigma \models Init$, and a pair of states $(\sigma_1, \sigma_2)$ is a transition if $(\sigma_1, \sigma_2) \models \rho$. We say that a state is reachable by at most $i$ steps of $\rho$ (or $i$-reachable for short, when $\rho$ is clear from the context) if it can be reached by at most $i$ applications of $\rho$ starting from some initial state. We denote the set of $i$-reachable states by $R_i$. We say that a state is reachable if it is $i$-reachable for some $i$. We say that $TS$ satisfies a safety property $P$ if all reachable states satisfy $P$. We often define $Bad \overset{def}{=} \neg P$, 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 $V$.

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 $\rho$. In the following, we use $(\varphi)'$ to denote the formula obtained by replacing every constant and relation symbol in formula $\varphi$ with its primed version.

Definition 2.2 (Invariants). Let $TS = (Init, \rho)$ be a transition system and $P$ a safety property over $V$. A closed formula $I$ is a safety inductive invariant (invariant, in short) for $TS$ and $P$ if (i) $Init \Rightarrow I$, and (ii) $I \land \rho \Rightarrow (I)'$, and (iii) $I \Rightarrow P$.

If there exists an invariant for $TS$ and $P$, then $TS$ satisfies $P$. 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$^2$ 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 $\text{while } Cond \text{ do } Cmd$, 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 $(\sigma_1, \sigma_2)$ describes one possible execution of the loop body, $Cmd$, i.e., it relates the state $\sigma_1$ at the beginning of an iteration of the loop to the state $\sigma_2$ 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 $\rho$ from a weakest liberal precondition predicate transformer, wlp, defined for each command type. As an example, the top of Table 2 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 $(\sigma, \sigma') \models Id \iff \sigma = \sigma'$. Formally, $Id$ is defined by

$$Id \overset{def}{=} \bigwedge_{c \in C} c = c' \land \bigwedge_{R \in R} \forall \tau. R(\tau) \leftrightarrow R'(\tau)$$ (1)

where $C$ and $R$ denote the sets of constants and relation symbols in $V$, respectively, and $\tau$ is a list of variables according to the arity of the relation symbol $R$. The vocabulary $V$ corresponds to the structure $\sigma$, and $V'$ corresponds to $\sigma'$.

We then define

$$\rho \overset{def}{=} \text{Cond} \land \text{wlp}[Cmd]Id$$ (2)
where \( \text{wlp}[\text{Cmd}] \) denotes the weakest liberal precondition of the loop body.

We define \( \text{Init} \) and \( \text{Bad} \) using the programs pre- and post-conditions, as well as its assertions:

\[
\text{Init} \overset{\text{def}}{=} \text{Pre} \quad \text{and} \quad \text{Bad} \overset{\text{def}}{=} (\neg\text{Cond} \land \neg\text{Post}) \lor (\text{Cond} \land \neg\text{wlp}[\text{Cmd}] \text{true}).
\]  

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 \( \neg\text{wlp}[\text{Cmd}] \text{true} \) of \( \text{Bad} \).

The construction of \( \rho \), \( \text{Init} \) and \( \text{Bad} \) ensures that \( \text{TS} = (\text{Init}, \rho) \) satisfies \( P = \neg\text{Bad} \) 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 \( \rho \), \( \text{Init} \) and \( \text{Bad} \) 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.

\( \text{EPR} \) and \( \text{EA}^R \). 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 \( \text{EA}^R \) [Itzhaky et al. 2013], an auxiliary logic built on top of EPR, which enables natural reasoning about programs manipulating linked-data structures. \( \text{EA}^R \) extends EPR by allowing a deterministic transitive-closure operator \( ^* \) over acyclic relations. Satisfiability of \( \text{EA}^R \) 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 \( \sigma = (D, \tau) \) 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 \( \text{null} \) 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 [I.1] and [I.2]. Notice
that \( n \) itself is not part of the vocabulary, but it is definable using the (open) formula \( \varphi_n \) in Table III. We use \( n^*(\alpha, \beta) \) as a shorthand for \( n^* (\alpha, \beta) \land \alpha \neq \beta \). Clearly, these definitions for \( \varphi_n \) 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 \( \varphi_n \) 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 \( \sigma = (D, I) \) 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 \( \sigma \models \varphi_n(v, u) \). 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 \( \rho \) over the above vocabulary by defining a weakest liberal precondition predicate transformer, \( wlp[\cdot] \), 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 III. 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 \( Q[\varphi/\gamma^*(\alpha, \beta)] \) denotes substitution of any atom of the form \( \gamma^*(\cdot, \cdot) \) in \( Q \) with the formula \( \varphi \), where \( \alpha \) and \( \beta \) may occur as term placeholders in \( \varphi \) and are filled in with the arguments of \( \gamma^* \). vars is the set of (constant symbols pertaining to) variables used in the program. As shown in [Izhaky 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

\[
n^*(\alpha, \beta) \land (\neg n^*(\alpha, x) \lor n^*(\beta, x)) \quad \text{(for } x.\text{n} := \text{null})
\]

and

\[
n^*(\alpha, \beta) \lor (y \neq \text{null} \land n^*(\alpha, x) \land n^*(y, \beta)) \quad \text{(for } x.\text{n} := y) .
\]

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, \( \alpha \) and \( \beta \) denote arbitrary heap locations. When traversing a pointer \( (x := y.\text{n}) \), the successor can be expressed by its transitive closure \( n^* \) using the formula \( \varphi_n \): for any two locations \( s \) and \( t \) (which are not \( \text{null} \)), the successor of \( s \) is \( t \) if \( t \) 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 \( \text{null} \) 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 \( \rho \) is defined as in Equation 2 (see Section 2.2). It is important to notice that, since \( \varphi_n \) is a universal formula occurring in a negative context in \( wlp[x := y.\text{n}] Q \), which is itself defined by a universal formula, the resulting formula \( \rho \) 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

\[
\rho \overset{\text{def}}{=} Cond \land wlp[\gamma] [\text{Cond}] Id
\]

(4)

Notice that the equivalence of semantics relies on the fact that for every location \( s \in \mathcal{U} \) which is different from \( \text{null} \) there is exactly one location \( t \) such that \( \varphi_n(s, t) \), as follows from the definition.
\[
\begin{align*}
\text{wlp}[\text{skip}] Q & \equiv Q \\
\text{wlp}[x := y] Q & \equiv Q[y/x] \\
\text{wlp}[\text{Cmd}_1 ; \text{Cmd}_2] Q & \equiv \text{wlp}[\text{Cmd}_1](\text{wlp}[\text{Cmd}_2] Q) \\
\text{wlp}[\text{if } B \text{ then } \text{Cmd}_1 \text{ else } \text{Cmd}_2] Q & \equiv [B] \wedge \text{wlp}[\text{Cmd}_1] Q \lor \neg[B] \wedge \text{wlp}[\text{Cmd}_2] Q \\
\text{wlp}[\text{assert } B] Q & \equiv [B] \wedge Q
\end{align*}
\]

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 \(x.n := y\) is preceded by a command \(x.n := \text{null}\).

\[
\begin{align*}
\text{wlp}^3[x := y.n] Q & \equiv \exists \alpha. \varphi_n(y, \alpha) \land Q[\alpha/x] \\
\text{wlp}^3[x := \text{new}] Q & \equiv \exists \alpha. (\bigwedge_{p \in \text{vars} \cup \{\text{null}\}} \neg n^+(p, \alpha)) \land Q[\alpha/x]
\end{align*}
\]

Table III: Leading-existential variant of wlp rules.

**Example 3.1.** Since the transition relation obtained for \(\text{filter}()\) is large, we demonstrate the construction of \(\rho\) on a simpler program, where the loop body consists of the following command:

\[
\text{Cmd} = k := i.n; \ i.n := \text{null}; \ i := k
\]

We then have

\[
\begin{align*}
\text{wlp}^3[\text{Cmd}] Q & = \text{wlp}^3[k := i.n; \ i.n := \text{null}; \ i := k] Q \\
& = \text{wlp}^3[k := i.n](\text{wlp}^3[i.n := \text{null}](\text{wlp}^3[i := k] Q)) \\
& = \exists \alpha. \varphi_n(i, \alpha) \land (\text{wlp}^3[i.n := \text{null}](Q[k/i])[\alpha/k]) \\
& = \exists \alpha. \varphi_n(i, \alpha) \land ((Q[k/i])[n^+(\alpha, i) \land \neg n^+(\alpha, i) \lor n^+(\beta, i))/n^+(\alpha, \beta)))[\alpha/k]
\end{align*}
\]

The transition relation is then constructed as follows:

\[
\begin{align*}
\rho = i \neq \text{null} \land \text{wlp}^3[\text{Cmd}]Id \\
& = i \neq \text{null} \land \text{wlp}^3[\text{Cmd}]\exists (i = i' \land k = k' \land \forall \alpha, \beta. n^+(\alpha, \beta) \leftrightarrow n^+(\alpha, \beta)) \\
& = i \neq \text{null} \land \\
& \quad \exists \alpha. \varphi_n(i, \alpha) \land (\alpha = i' \land \forall \alpha, \beta, (n^+(\alpha, \beta) \land \neg n^+(\alpha, i) \lor n^+(\beta, i))) \leftrightarrow n^+(\alpha, \beta))
\end{align*}
\]

**Proposition 3.2.** \(\text{EA}^R\) is closed under \(\text{wlp}^3[\text{Cmd}]\); that is, if \(Q \in \text{EA}^R\) then \(\text{wlp}^3[\text{Cmd}] Q \in \text{EA}^R\). In particular, \(\rho \in \text{EA}^R\) (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 $Pre = i = h \land j = \text{null}$ and $Post = h \neq \text{null} \rightarrow \forall z. n^*(h, z) \rightarrow ok(z)$. 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, $Init \equiv i = h \land j = \text{null}$ and $Bad \equiv i = \text{null} \land \neg (h \neq \text{null} \rightarrow \forall z. n^*(h, z) \rightarrow ok(z))$. Here, a state is bad if $i = \text{null}$ (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 false and is subsumed by the first disjunct.

In our analysis, the $EA^R$ formulae $Init$, $Bad$ and $\rho$ 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, $m_s = m_t = \text{null}$). The following relationship holds between $n^*$ and $k^*, m_s, m_t$:

$$\forall\alpha,\beta. n^*(\alpha,\beta) \iff k^*(\alpha,\beta) \lor (m_s \neq \text{null} \land k^*(\alpha, m_s) \land k^*(m_t, \beta))$$

(5)

Therefore $k^*, m_s, m_t$ fully characterize the heap reachability. The axioms in Table II 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, $m_s \neq \text{null} \rightarrow k^*(m_t, m_s) \land \exists k^+ (m_s, \alpha)$. We now modify the $wlp$ formulae from Table II to reflect the new situation. The new semantics for $x.n := y$, $x.n := y.n$ are shown in Table IV. The multiple substitutions in the brackets are done in parallel. The operator $\text{ite}(p, a, b)$ denotes a term that is equal to $a$ if $p$ is $\text{true}$, and $b$ otherwise. $\psi_{km}(s, t)$ is a formula expressing a path between $s$ and $t$ utilizing the special edge $m$. $\psi_{km}(u, u)$ 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 $x.n := y$, and removing it or replacing it with a $k$ edge when the cycle is broken by $x.n := \text{null}$. Notice that $wlp[x := y.n]$ remains as in Table II except that the definition of $\varphi_n$ is changed. The adjustment in Table II for $wlp^3$ is suitable in this case as well.

4. UNIVERSAL-PROPERTY-DIRECTED REACHABILITY

In this section, we present Universal Property Directed Reachability (PDR$^U$), an algorithm for checking if a transition system $TS$ satisfies a safety property $P$. PDR$^U$ 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. 2013]. 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 2012].
formula in conjunction and contains all universal and existential formulae. We require that every satisfiable
on the unique aspects of PDR diagram [Chang and Keisler 1990].

A:10 Aleksandr Karbyshev et al.

of the more commonly used to express transition systems and properties. In particular, as
None a model exists and a formula expressible in a logic closure using the approach of [Itzhaky et al. 2013]. See Section 3.) We assume that
L
expresses in a logic

— For every element $e_i \in D$, a fresh variable $x_{e_i}$ is introduced.

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].

4.1. Diagrams as Structural Abstractions

PDR\textsuperscript{\textcopyright} iteratively strengthens a candidate invariant by retrieving program states that lead to bad states and checking whether the retrieved states are reachable. In that sense, PDR\textsuperscript{\textcopyright} 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 $\sigma$, denoted by $\text{Diag}(\sigma)$, is an existential cube which describes explicitly the relations between all the elements of the model.\textsuperscript{4}

Definition 4.1 (Diagrams). Given a finite model $\sigma = (D, I)$ over alphabet $V$, the diagram of $\sigma$, denoted by $\text{Diag}(\sigma)$, is a closed formula over alphabet $V$ which denotes the set of models in which $\sigma$ can be isomorphically embedded. $\text{Diag}(\sigma)$ is constructed as follows.

- For every element $e_i \in D$, a fresh variable $x_{e_i}$ is introduced.

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

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
wlp[x.n := null] $Q$ & $= Q$
\begin{align*}
\text{ite}(\psi_{km}(x, x), \text{null}, m_s)/m_s, \\
\text{ite}(\psi_{km}(x, x), \text{null}, m_t)/m_t, \\
k^*(\alpha, \beta) \land (\neg k^*(\alpha, x) \lor k^*(\beta, x)) \lor \\
(\psi_{km}(\alpha, \beta) \land \neg k^*(\alpha, x) \land k^*(\beta, x) \land x \neq m_s)/k^*(\alpha, \beta)
\end{align*}
\end{tabular}
\end{table}
— \( \varphi_{\text{distinct}} \) 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.
— \( \varphi_{\text{constants}} \) is a conjunction of inequalities of the form \( c = x_e \) for every constant symbol \( c \) such that \( \sigma \models c = e \).
— \( \varphi_{\text{atomic}} \) is a conjunction of atomic formulae which include for every predicate \( p \in V \) the atomic formula \( p(x_e) \) if \( \sigma \models p(\bar{e}) \), and \( \neg p(\bar{e}) \) otherwise.

Thus: \( \text{Diag}(\sigma) \equiv \exists x_{e_1} \cdots x_{e_{|\sigma|}}, \varphi_{\text{distinct}} \land \varphi_{\text{constants}} \land \varphi_{\text{atomic}}. \)

Intuitively, one can think of \( \text{Diag}(\sigma) \) as the formula produced by treating individuals in \( \sigma \) 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 \( \text{Diag}(\sigma) \) is well defined, because we consider only models with a finite domain.

**Example 4.2.** The diagram of \( \sigma_b \), depicted in Figure 3, is

\[ \text{Diag}(\sigma_b) \equiv \exists x_0, x_1, x_2, x_0 \neq x_1 \land x_0 \neq x_2 \land x_1 \neq x_2 \land h = x_0 \land j = x_1 \land i = x_2 \land \text{null} = x_2 \land \neg \text{ok}(x_0) \land \neg \text{ok}(x_1) \land \neg \text{ok}(x_2) \land n^*(x_0, x_0) \land n^*(x_1, x_1) \land n^*(x_2, x_2) \land n^*(x_0, x_1) \land \neg n^*(x_0, x_2) \land \neg n^*(x_1, x_2) \land \neg n^*(x_2, x_0) \land \neg n^*(x_2, x_1) \]

The first line records the fact that the domain of \( \sigma_b \) consists of three elements. The second line characterizes the interpretations of all the constant symbols in \( \sigma_b \). The other lines capture precisely the interpretation of predicates \( \text{ok} \) and \( n^* \) in \( \sigma_b \).

We say that \( \sigma_1 = (D_1, I_1) \) is a substructure of \( \sigma_2 = (D_2, I_2) \) if \( D_1 \subseteq D_2 \) and for every \( v \in V \), \( I_1(v) \) is the restriction of \( I_2(v) \) to \( D_1 \). The following lemma is well known:

**Lemma 4.3.** \( \sigma' \models \text{Diag}(\sigma) \) iff \( \sigma \) is isomorphic to a substructure of \( \sigma' \).

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

**Example 4.4.** In Figure 3, several models of \( \text{Diag}(\sigma_b) \) are depicted. For clarity, the edges drawn correspond to \( n \)-links extracted from \( n^* \) of each structure using \( \varphi_n \) (recall from Table II). Note that all of them contain \( \sigma_b \) as a substructure. For example, in \( \sigma_b^1 \), 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 \( \sigma_b^2 \), 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 \( \sigma_b^2 \) contains \( \sigma_b \) as a substructure, recall that the vocabulary contains \( n^* \), and not \( n \) itself; while no \( n \)-annotated edge appears in \( \sigma_b^2 \) from \( v_0 \) to \( v_1 \), \( n^*(v_0, v_1) \) does hold in \( \sigma_b^2 \) as well due to the transitive nature of \( n^* \). Similarly, \( \sigma_b^3 \), which represents a list that contains two additional nodes between \( h \) and \( j \), also contains \( \sigma_b \) 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 \( \sigma \) be a model over \( V \), and let \( \varphi \) be a closed existential first-order formula over \( V \). If \( \sigma \models \varphi \) then \( \text{Diag}(\sigma) \models \varphi \).

Stated differently, \( \text{Diag}(\sigma) \) is the strongest existentially quantified formula that has \( \sigma \) as a model. Semantically, Lemma 4.5 means that for any models \( \sigma \) and \( \sigma' \) such that \( \sigma' \models \text{Diag}(\sigma) \) if \( \sigma \models \varphi \) then \( \sigma' \models \varphi \). This implies that if a bad state is reachable from \( \sigma \) and the program can be proven correct using an inductive universal invariant \( I \) then all the states in \( \sigma \)'s diagram are unreachable too; \( I \) is an inductive invariant, thus any state \( \sigma \) leading to a bad state must satisfy the (closed existential)
For clarity, we refer to Algorithm 2 and Algorithm 3, respectively, as subroutines. The algorithm uses an array \( F \) to any

4.2. Data Structures and Frames

\( PDR^v \) is shown in Algorithm 1. It uses procedures \textit{block()} and \textit{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 \( i \)-th 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 \textit{frame counter} \( N \) which records the number of frames it developed. We refer to \( F_0 \) as the \textit{initial} frame, to \( F_N \) as the \textit{frontier} frame, and to any \( F_i \), where \( 0 \leq i < N \), as a \textit{back} frame.

\( PDR^v \) 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 \textit{approximate reachability sequence}.

\begin{definition}
Let \( TS = (\text{Init}, \rho) \) be a transition system and \( P \) a safety property. A sequence \( \langle F_0, F_1, \ldots, F_N \rangle \) of closed formulae is an \textit{approximate reachability sequence} for \( TS \) and \( P \) if:

\begin{enumerate}
\item \( \text{Init} \Rightarrow F_0 \).
\item \( F_i \Rightarrow F_{i+1} \), for all \( 0 \leq i < N \), i.e., for every state \( \sigma \), if \( \sigma \models F_i \) then \( \sigma \models F_{i+1} \).
\item \( F_i \land \rho \Rightarrow (F_{i+1})' \), for all \( 0 \leq i < N \), i.e., for every transition \( (\sigma_1, \sigma_2) \models \rho \), if \( \sigma_1 \models F_i \) then \( \sigma_2 \models F_{i+1} \).
\item \( F_i \Rightarrow P \), for all \( 0 \leq i \leq N \).
\end{enumerate}

\end{definition}

Items (i) and (iii) ensure that every frame includes the states of the previous frame and their successors, respectively. Together with item (ii), 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

\( PDR^v \) is an iterative algorithm. At every iteration, the algorithm either strengthens the \( N \)-th 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 \( \varphi_i \) which holds for any \( i \)-reachable state, and then conjoining \( F_i \) with \( \varphi_i \).

\textit{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 \textit{Init} and \textit{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

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\( (\sigma_b) \) & \( h \) & \( j \) & \( i \) & \text{null} \\
\hline
\( \sigma_0 \) & \( n \) & \( v_2 \) & \( n \) & \text{true} \\
\hline
\( (\sigma_b^1) \) & \( h \) & \( j \) & \( i \) & \text{null} \\
\hline
\( \sigma_0 \) & \( n \) & \( v_0 \) & \( n \) & \text{true} \\
\hline
\( (\sigma_b^2) \) & \( h \) & \( j \) & \( i \) & \text{null} \\
\hline
\( \sigma_0 \) & \( n \) & \( v_0 \) & \( n \) & \text{true} \\
\hline
\end{tabular}
\caption{Graphical depiction of the model \( \sigma_b \) and models \( \sigma_b^1, \sigma_b^2, \sigma_b^3 \) of \( \text{Diag}(\sigma_b) \).}
\end{figure}

formulas \( \neg \mathcal{I} \). Hence, \( \text{Diag}(\sigma) \Rightarrow \neg \mathcal{I} \), which means that all states satisfying \( \text{Diag}(\sigma) \) are unreachable. In this sense, the abstraction based on diagrams is precise for programs with universal invariants. This property of the diagrams will allow \( PDR^v \) to also prove absence of universal invariants (see Prop. 5.6).
Algorithm 1: PDR\textsuperscript{5} (Init, ρ, Bad)

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

Algorithm 2: block(j, σ)

19. ϕ = Diag(σ)
20. if (j = 0) ∨ (j = 1 ∧ SAT(ϕ ∧ Init)) then
21. analyzeCEX(j, N)
22. while SAT(F_{j-1} ∧ ρ ∧ (ϕ')) do
23. σ_{a} := reduce_{ϕ}(model(F_{j-1} ∧ ρ ∧ (ϕ')))
24. block(j - 1, σ_{a})
25. for i = 0...j do
26. F_{i} := F_{i} ∧ ¬ϕ

Algorithm 3: analyzeCEX(j, N)

31. if j = 0 ∧ there exists σ_{0},...,σ_{N} such that
32. σ_{0} \models Init
33. (σ_{i}, σ_{i+1}) \models ρ for every 0 ⩽ i < N, and
34. σ_{N} \models Bad
35. then exit invalid: σ_{0},...,σ_{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},...,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 9). 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 10). If no such state exists, it means that no bad state is N-reachable. Moreover, at this point \langle F_{0},...,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} ∧ 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_{i} 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.\textsuperscript{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, σ_{b}). In this example, block() succeeds to block σ_{b}. Unfortunately, the strengthened frame F_{1}^{\prime} (see below) still has bad models. Therefore, the iterative strengthening continues and the next

\textsuperscript{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} ∨ ρ ⇒ (ϕ') 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].
iterations find $\sigma'$, depicted in Figure 4 as a bad witness model for $F_1^1$, $\sigma''_b$ as a bad witness model of $F_1^2$ and $\sigma''_b$ as a bad witness model of $F_3^2$. At that point, however, the algorithm determines that the strengthened frame $F_4^1$ does not have a bad witness. $(F_0, F_4^1)$ is now an approximate reachability sequence and PDR goes on and initializes a new frame, $F_2$, to true, and the search for an inductive invariant continues.

**Diagram-Based Abstract Blocking.** Procedure $\text{block}(j, \sigma)$, shown in Algorithm 2, gets an index of a frame $j = 0, \ldots, N$ and a state $\sigma$ which is included in the $j$th frame, i.e., $\sigma \models F_j$, and tries to determine whether $\sigma$ is $j$-reachable. The unique aspect of our approach is the way in which it abstracts $\sigma$ to a set of states in order to accelerate the strengthening routine. Namely, the use of diagrams. More specifically, PDR computes the diagram $\varphi$ of $\sigma$ (line 21) and then checks whether there is a $j$-reachable state satisfying $\varphi$. Importantly, due to Lemma 4.5, if a universal invariant exists then the generalization of $\sigma$ 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 $\sigma$, but its entire diagram.

The check if the diagram $\varphi$ of $\sigma$ includes a $j$-reachable state is done conservatively by determining whether some state of $\varphi$ is an initial state or has a predecessor in $F_{j-1}$. (Recall that $F_{j-1}$ over-approximates $R_{j-1}$.) The former is equivalent to checking if $\varphi \land \text{Init}$ is satisfiable. Note that if we reached the initial frame, i.e., if $j = 0$, then $\sigma \models \text{Init}$, hence the above formula is guaranteed to be satisfiable. Explicitly checking that $\varphi \land \text{Init}$ is satisfiable is required only at the second frame, i.e., when $j = 1$ (see proof in Section 5.1).

**Lemma 4.8.** For every $1 < j \leq N$, when $\text{block}(j, \sigma)$ is called, $F_i \models \neg \text{Diag}(\sigma)$ for every $i \leq j - 1$. In particular, $\text{Init} \models \neg \text{Diag}(\sigma)$.

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

**Case I.** If $\delta$ is unsatisfiable then no state represented by $\varphi$ is $j$-reachable. Hence, $F_j$ remains an over-approximation of $R_j$, even if any state of $\varphi$ is excluded. The exclusion is done by conjoining the $j$th 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, $F_i \land \rho \Rightarrow (F_{i+1})'$ (iii) is preserved despite conjoining

---

**Footnotes:**

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

7As an optimization, one can consider $\delta' = F_{j-1} \land \neg \varphi \land \rho \land (\varphi)'$ instead of $\delta$. 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 strengthening of $\delta$ can make the satisfiability check cheaper.
An unreachable from $\text{Init}$

$\phi$ is variants of the sequence of frames maintained by $\text{PDR}$

In this section we formalize the correctness guarantees of $\psi = (\neg \varphi)$. 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 $\varphi$ as the blocking of (the diagram of) $\sigma$ from frame $F_j$.

Example 4.9. In our running example, in the first iteration block$(1, \sigma_b)$ updates $F_0^1$ to $F_1^1 = \text{true} \land \neg \Diag(\sigma_b)$. This excludes $\sigma_b$, but also all states where $i = \text{null}$, ok is empty, and $j$ is $n$-reachable from $h$ in any (nonzero) number of steps (e.g. the states $\sigma_2^b$ and $\sigma_3^b$ depicted in Figure 3). In later iterations block updates $F_1^2 = F_1^1 \land \neg \Diag(\sigma_b)$, $F_1^3 = F_1^2 \land \neg \Diag(\sigma_b)$, and $F_1^4 = F_1^3 \land \neg \Diag(\sigma_b)$.

Case II. If $\delta$ is satisfiable, then there exists an adverse state $\sigma_a$ in frame $F_{j-1}$, a state which is the predecessor of some state of the diagram of $\sigma$ that we try to block at frame $F_j$. Note that $\sigma_a$ is not necessarily a predecessor of $\sigma$ itself. The adverse state $\sigma_a$ is found by taking the reduct of a (finite) model of $\delta$ to $\forall$ (line 25). If an adverse model $\sigma_a$ 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 $N$th 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 $\text{analyzeCEX}()$, 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 $⟨\varphi_j, \varphi_{j+1}, \ldots, \varphi_N⟩$ is an abstract counterexample if the formulae $\varphi_j \land \text{Init}, \varphi_N \land \text{Bad}$, and $\varphi_{i+1} \land \rho \land (\varphi_{i+1})'$ for every $i = j, \ldots, N-1$, are all satisfiable. The abstract counterexample is spurious if there exists no sequence of states $⟨\sigma_j, \sigma_{j+1}, \ldots, \sigma_N⟩$ such that $\sigma_j \models \text{Init}, \sigma_N \models \text{Bad}$, 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, $\text{analyzeCEX}()$ 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 $\varphi$ from frames $0, \ldots, j$ by conjoining them with the clause $\neg \varphi$ (line 28), our implementation uses a minimal unsat core of $\psi = ((\text{Init})' \lor (F_{j-1} \land \rho)) \land (\varphi)'$ to define a clause $L$ which implies $\neg \varphi$ and is also disjoint from $\text{Init}$ and unreachable from $F_{j-1}$. Blocking is done by conjoining $L$ with $F_i$ for every $i \leq j$.

5. CORRECTNESS

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

---

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 $((\text{Init})' \lor (F_{j-1} \land \rho) \land (\neg L)'$ is unsatisfiable.
5.1. Properties of the Frames Computed by PDR

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

Lemma 5.1. Let TS = (Init, ρ) be a transition system and P a safety property. For every \( N \geq 1 \), the sequence \( \langle F_0, F_1, \ldots, F_N \rangle \) obtained by PDR right before \( N \) is increased in line [11] is an approximate reachability sequence for TS and \( P \) (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, we need the following definition.

Definition 5.2 (Relaxed Traces). Given a transition system TS = (Init, ρ), we say that a sequence of states \( σ_0, \ldots, σ_n \) is a relaxed trace of TS if for every \( 0 \leq i \leq n-1 \), there exists \( σ_{i+1} \) such that \( (σ_i, σ_{i+1}) \models ρ \) and \( σ_{i+1} \models \text{Diag}(σ_{i+1}) \).

For a transition system TS and a property \( P \), with Bad = \( \neg P \), 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, \( σ \models F_j \) if and only if there exists a relaxed trace \( σ_0, \ldots, σ_n \) such that \( σ_0 = σ, σ_n \models \text{Bad} \), and \( n \leq i \). In particular, \( B_0 = \{ σ | σ \models \text{Bad} \} \). 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 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, ρ) be a transition system and P a safety property. For every \( N \geq 1 \) and \( 0 \leq j \leq N-1 \), if \( σ \models F_j \), then \( σ \not\models B_{N-1-j} \).

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 \( σ \models F_j \) then \( σ \not\models B_{N-j} \). The proof follows a simple induction on \( j \) that shows that if \( σ \models B_{N-1-j} \), then it is excluded from \( F_j \).

On the other hand, it is easy to prove by induction that every state that \( \text{PDR} \) attempts to block in frame \( F_j \) is a state in \( B \):

Lemma 5.4. Let TS = (Init, ρ) be a transition system and P a safety property. For every \( N \geq 1 \) and \( 0 \leq j \leq N \), if block\( (j, σ) \) is called, then \( σ \models B_{N-j} \).

We can now return to the proof of Lemma 4.8.

Proof of Lemma 4.8 Suppose \( σ \models F_j \) is discovered in the backward traversal from \( F_N \). By Lemma 5.4 \( σ \models B_{N-j} \). Therefore, by Lemma 5.3 \( σ \not\models F_{j-1} \). 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 \( \text{Diag}(σ) \Rightarrow \neg F_{j-1} \), or equivalently, \( F_{j-1} \Rightarrow \neg \text{Diag}(σ) \). Since \( F_i \Rightarrow F_{j-1} \) for every \( i \leq j - 1 \), we conclude that \( F_i \Rightarrow \neg \text{Diag}(σ) \) 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\( (1, σ) \) is called, \( F_0 \Rightarrow \neg \text{Diag}(σ) \). In this case, the additional check of \( (j = 1 \land SAT(φ \land \text{Init})) \) performed in line [22] of block\( (j, σ) \) is always false, and hence not needed.

5.2. Correctness of the Outcome of PDR

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

Proof. $PDR^\forall$ returns valid if there exists $i$ such that $F_{i+1} \Rightarrow F_i$. Therefore, $F_i \land \rho \Rightarrow (F_{i+1})' \Rightarrow (F_i)'$. Recall that, by the properties of an approximate reachability sequence, $\text{Init} \Rightarrow F_0 \Rightarrow F_1$ and $F_1 \Rightarrow P$. Therefore, $F_1$ is an inductive invariant, which ensures that $TS$ satisfies $P$.

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 $P$, it also obtains a universal inductive invariant. Namely, the frame $F_1$, 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 $TS = (\text{Init}, \rho)$ be a transition system and let $P$ be a safety property. If $PDR^\forall$ obtains a spurious counterexample $\langle \varphi_0, \ldots, \varphi_N \rangle$ then there exists no universal safety inductive invariant $I$ for $TS$ and $P$ over the given vocabulary.

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

Proposition 5.7. Let $TS = (\text{Init}, \rho)$ be a transition system and $P$ 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 $I$ be a universal inductive safety invariant for $P$. Then for any $\sigma \in B$, we have $\sigma \models \neg I$.

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

For the inductive step, let $\sigma \in B_{i+1}$. By definition of $B_{i+1}$, there exist models $\sigma_i$ and $\sigma_i'$ such that $(\sigma, \sigma_i') \models \rho \land (\text{Diag}(\sigma_i))'$ and $\sigma_i \in B_i$. Moreover, by the induction hypothesis, $\sigma_i \models \neg I$.

Since $\neg I$ is an existential formula, this means by Lemma 4.5 that $\text{Diag}(\sigma_i) \models \neg I$. We conclude that $\rho \land (\text{Diag}(\sigma_i))' \Rightarrow \rho \land (\neg I)'$. Therefore, $(\sigma, \sigma_i')$ is also a model of the formula $\rho \land (\neg I)'$.

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

Proof of Prop. 5.6. Assume that there exists a universal safety inductive invariant $I$ over $\forall$. By Lemma 5.4, for every state $\sigma_i$ generated by $PDR^\forall$ at frame $F_i$, $\sigma_i \in B$. Hence by Prop. 5.7, $\sigma_i \models \neg I$.

This implies, by Lemma 4.5, that every diagram $\varphi_i$ generated by $PDR^\forall$ at frame $F_i$ is such that $\varphi_i \Rightarrow \neg I$, and hence $\varphi_i \Rightarrow \neg \text{Init}$.

(Recall that by definition $\text{Init} \Rightarrow I$, i.e., $\neg I \Rightarrow \neg \text{Init}$). This contradicts the existence of a spurious counterexample, where $\varphi_i \land \neg I$ 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, $\text{Init}$ to be the procedure’s precondition and $P$ to be the safety property whose negation is $\text{Bad} = (i = \text{null} \land j = \text{null}) \land \neg \text{Post}$, where $\text{Post}$ is the procedure’s postcondition. Consider the state $\sigma_0$ depicted in Figure 6. Clearly, this model satisfies $\text{Init}$. Therefore, if $I$ exists, $\sigma_0 \models I$. $\sigma_0$ is a predecessor of $\sigma_1$ and hence it should be the case that $\sigma_1 \models I$. Now consider $\sigma_1$, which is a submodel of $\sigma_1$ and interprets all constants as in $\sigma_1$. If $I$ is universal, then $\sigma_1 \models I$ as well. The model $\sigma_2$ is a successor of $\sigma_1$ and hence $\sigma_2 \models I$. However, $\sigma_2 \not\models P$, 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 $P$ exists. Note that state $\sigma_1$ is a predecessor of $\sigma_2$ and recall that $\sigma_0$ is a predecessor of $\sigma_1$. The spurious counterexample was obtained because $\sigma_1$ satisfies the diagram of state $\sigma_1$. 

requires: \( p = \text{null} \land q = \text{null} \land i = g \land q \neq \text{null} \land j = h \land h \neq \text{null} \land \\
\exists v. n'(g, v) \land h'(v, h) \land v \neq \text{null} \)

ensures: \( p = q \land p \neq \text{null} \land i = \text{null} \land j = \text{null} \)

void traverseTwo(List g, List h) {
    i := g; j := h;
    while (i \neq \text{null} \lor j \neq \text{null}) {
        if i \neq \text{null} then { p := i; i := i.n };
        if j \neq \text{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 \( \text{PDR}^\land \) 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 \( \text{PDR}^\land \) 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 \( \text{PDR}^\land \), here formulated for a transition system \( TS = (\text{Init}, \rho) \) and a property \( \mathcal{P} \), with \( \mathcal{B} = \neg \mathcal{P} \):

**Proposition 6.1.** If backwards reachability, given by Algorithm 4, terminates on input transition system \( TS \) and property \( \mathcal{P} \), then \( \text{PDR}^\land \) terminates as well.

In order to establish this result, we introduce a notion of effective encodings of the (backwards) reachable states, \( \mathcal{B} \) (defined in Section 5.1). In a nutshell, a set of (backwards) reachable states \( \mathcal{B} \) 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 \( \text{PDR}^\land \) using the two lemmas:

**Lemma 6.2.** If Algorithm 4 terminates on \( TS \) and \( \mathcal{P} \), then \( \mathcal{B} \) is effective.

and

**Lemma 6.3.** If \( \mathcal{B} \) is effective for \( TS \) and \( \mathcal{P} \), then \( \text{PDR}^\land \) 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 \( \mathcal{B} \). If the states represented by the resulting formula do not properly contain the initial states \( \text{Init} \), then there are no universal inductive invariants, otherwise there is one.
Algorithm 4: Naïve Backward Reachability

1 $I := \text{true}$
2 while SAT$(I \land \text{Bad})$ or SAT$(I \land \rho \land (\neg I)')$ do
3 $(\sigma, \sigma') := \text{model}((I \land \text{Bad}) \lor (I \land \rho \land (\neg I)'))$
4 $I := I \land \neg \text{Diag}(\sigma)$
5 if SAT$(\text{Init} \land \neg I)$ then
6 return no universal inductive invariant
7 else
8 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 $B$ to denote the set of states that can reach a bad state via a relaxed trace of $TS$ in at most $i$ steps, and $B_i$ 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 $\psi_B$ 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 $\psi$ in $\mathcal{L}$ is satisfiable, and a function model$(\psi)$, which returns a finite model $\sigma$ of $\psi$ 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 : $\mathbb{N} \rightarrow \mathbb{N}$ that given a formula $\psi$ in $\mathcal{L}$, with at most $n_\psi$ existentially quantified variables and constants, $\psi$ has a finite model if and only if it has a model which contains no more than $\text{bound}(n_\psi)$ elements. Note that such a 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$(n_\psi)$. Assuming that $\mathcal{L}$ satisfies the additional requirement, and that all satisfiability checks performed by PDR$^\mathcal{L}$ 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$^\mathcal{L}$ 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 $B = \{\sigma \mid \sigma \models \bigvee_{i=1}^m \text{Diag}(\sigma_i)\}$. For each $i = 1, \ldots, m$, let $k_i$ be the length of a shortest relaxed trace leading from $\sigma_i$ 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 $\sigma_i$, 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, \( \text{PDR}^\forall \) 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 \( \text{PDR}^\forall \) 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 \( \text{PDR}^\forall \) in its backward traversal, and hence can bound the size of generated clauses. Let \( n_c \) denote the number of constants in \( \forall \), \( n_{\text{Bad}} \) denote the number of existential quantifiers in \( \text{Bad} \) and \( n_{\rho} \) denote the number of existential quantifiers in \( \rho \). We show by induction on \( j \) that we can bound the size of models obtained by \( \text{PDR}^\forall \) in frame \( N - j \) by a function of \( n_c, n_{\text{Bad}} \) and \( n_{\rho} \) (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 \( \sigma \models F_N \land \text{Bad} \). Since \( F_N \) is a universal formula, the properties of \( \mathcal{L} \) ensure that the size of the obtained model is bounded by \( \text{bound}(n_c + n_{\text{Bad}}) \). For the induction step, when \( \text{PDR}^\forall \) makes a step backward from a diagram of a state \( \sigma \) using \( \rho \), it uses the formula \( F_{j-1} \land \rho \land (\varphi)^\forall \), where \( \varphi = \text{Diag}(\sigma) \). The only existential quantifiers in this formula result from \( \rho \) and \( \varphi \), where the number of the latter is equal to the size of the domain of \( \sigma \) (by the construction of the diagram).

Our assumptions on \( \mathcal{L} \) and on \( \text{model}(\psi) \) therefore ensure that the size of the domain of the obtained model is bounded by

\[
\text{bound}(n_c + n_{\rho} + n) \quad (*)
\]

where \( n \) is the size of \( \sigma \). By the induction hypothesis, \( n \) is bounded by a function of \( n_c, n_{\text{Bad}} \) and \( n_{\rho} \) only, hence the claim follows.

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

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

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

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

6.1. Termination when Reasoning with Effectively Propositional Logic

If \( \text{Init} \) and \( \rho \) are \( \text{EPR} \) formulae and \( \mathcal{P} \) is a universal formula, then all the satisfiability queries made by \( \text{PDR}^\forall \) are of \( \text{EPR} \) formulae. \( \text{EPR} \) has the finite model property, and its satisfiability is decidable. Further, \( \text{EPR} \) meets the additional requirement needed for termination, as an \( \text{EPR} \) formula \( \psi \) is satisfiable if and only if it has a satisfying model whose size is bounded by \( n_\psi \), where \( n_\psi \) is the number of constants and existentially quantified variables in \( \psi \). That is, for every \( \text{EPR} \) formula \( \psi \), \( \text{bound}(n_\psi) = n_\psi \). 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 \( \text{PDR}^\forall \) in the last \( k + 1 \) frames can be explicitly expressed as \((k + 1) \cdot n_c + k \cdot n_{\rho} + n_{\text{Bad}} \), where \( n_c \) denotes the number of constants in \( \forall \), \( n_{\text{Bad}} \) denotes the number of existential quantifiers in \( \text{Bad} \) and \( n_{\rho} \) denotes the number of existential quantifiers in \( \rho \). 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. 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 PDR also terminates on such programs.

7. IMPLEMENTATION AND EMPIRICAL EVALUATION

PDR 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 as explained in Section 5.

We note that the use of EA is key for the ability of PDR 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 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 \( \sigma_b \) presented in Figure 4 we also block all lists in which \( j \) is reachable from \( h \) in any number of steps.

**Benchmarks.** We implemented PDR 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 0482712727c) [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() to record whether a node is allocated. We encode the instructions new or free by means of alloc() 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: \( \forall \alpha. \text{alloc}(\alpha) \rightarrow \forall x. x \in PVar \Rightarrow n^*(x, \alpha) \), 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

\[ A \sqsubseteq B \]

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 \( i < j \).

\[ \text{https://bitbucket.org/tausigplan/updr-distrib/} \]

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.

<table>
<thead>
<tr>
<th>(a) Verification</th>
<th>Full</th>
<th>Memory safety</th>
<th>Memory safety [2014]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>N</td>
<td>#Z3</td>
</tr>
<tr>
<td>conical</td>
<td>1.2</td>
<td>3</td>
<td>56</td>
</tr>
<tr>
<td>create</td>
<td>1.4</td>
<td>3</td>
<td>62</td>
</tr>
<tr>
<td>delete</td>
<td>13.2</td>
<td>5</td>
<td>278</td>
</tr>
<tr>
<td>delete-all</td>
<td>10.3</td>
<td>5</td>
<td>255</td>
</tr>
<tr>
<td>filter</td>
<td>37.1</td>
<td>6</td>
<td>430</td>
</tr>
<tr>
<td>insert-at</td>
<td>1.8</td>
<td>3</td>
<td>69</td>
</tr>
<tr>
<td>insert</td>
<td>1.5</td>
<td>3</td>
<td>68</td>
</tr>
<tr>
<td>merge</td>
<td>244.6</td>
<td>7</td>
<td>1429</td>
</tr>
<tr>
<td>reverse</td>
<td>19.7</td>
<td>5</td>
<td>289</td>
</tr>
<tr>
<td>split</td>
<td>178.0</td>
<td>8</td>
<td>1079</td>
</tr>
<tr>
<td>uf-find</td>
<td>43.3</td>
<td>8</td>
<td>590</td>
</tr>
<tr>
<td>uf-union</td>
<td>178.8</td>
<td>7</td>
<td>1189</td>
</tr>
<tr>
<td>Sorted singly-linked lists</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sorted-insert</td>
<td>3.9</td>
<td>3</td>
<td>85</td>
</tr>
<tr>
<td>sorted-merge</td>
<td>400.3</td>
<td>8</td>
<td>1535</td>
</tr>
<tr>
<td>bubble-sort</td>
<td>90.5</td>
<td>11</td>
<td>904</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>1681.1</td>
<td>13</td>
<td>4601</td>
</tr>
<tr>
<td>Doubly-linked lists</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>create</td>
<td>5.7</td>
<td>5</td>
<td>139</td>
</tr>
<tr>
<td>delete</td>
<td>3.1</td>
<td>4</td>
<td>90</td>
</tr>
<tr>
<td>insert-at</td>
<td>4.8</td>
<td>4</td>
<td>132</td>
</tr>
<tr>
<td>Composite linked-list structures</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nested-flatten</td>
<td>564.6</td>
<td>17</td>
<td>3019</td>
</tr>
<tr>
<td>nested-split</td>
<td>680.5</td>
<td>9</td>
<td>1144</td>
</tr>
<tr>
<td>overlaid-delete</td>
<td>188.5</td>
<td>6</td>
<td>1054</td>
</tr>
<tr>
<td>Restricted cyclic singly-linked lists</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>is-cycle</td>
<td>1.5</td>
<td>4</td>
<td>81</td>
</tr>
<tr>
<td>last</td>
<td>4.6</td>
<td>6</td>
<td>143</td>
</tr>
<tr>
<td>unchain</td>
<td>823.8</td>
<td>9</td>
<td>1986</td>
</tr>
<tr>
<td>insert</td>
<td>91.2</td>
<td>5</td>
<td>208</td>
</tr>
<tr>
<td>delete</td>
<td>8.2</td>
<td>4</td>
<td>138</td>
</tr>
<tr>
<td>reverse</td>
<td>253.9</td>
<td>8</td>
<td>1202</td>
</tr>
</tbody>
</table>

### (b) Absence of a universal invariant

<table>
<thead>
<tr>
<th>Bug finding</th>
<th>Description</th>
<th>Time</th>
<th>N</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted-insert</td>
<td>Precondition is too weak (omitted $e \neq null$)</td>
<td>3.6</td>
<td>2</td>
<td>42</td>
</tr>
<tr>
<td>insert</td>
<td>Precondition is too weak (omitted $e \neq null$)</td>
<td>3.6</td>
<td>2</td>
<td>42</td>
</tr>
<tr>
<td>insert-at</td>
<td>Forgot a corner case where $\sim ok(h)$</td>
<td>5</td>
<td>4</td>
<td>68</td>
</tr>
<tr>
<td>insert-sort</td>
<td>Forgot to link the two segments</td>
<td>7.5</td>
<td>1</td>
<td>49</td>
</tr>
<tr>
<td>sorted-merge</td>
<td>Forgot to link the two segments</td>
<td>7.5</td>
<td>1</td>
<td>49</td>
</tr>
</tbody>
</table>

Itzhaky et al. 2014.

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()
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(f)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

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.
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 exten-
sion of predicate abstraction to support free variables [Planagan 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
[Planagan 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 ab-
straction. 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 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 in-
troducing 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 PDR 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] em-
loys 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 approx-
imation 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 [Ghi-
lardi and Ranise 2010b; Gilardi 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 invari-
ants. 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., “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 Opden 1975]. STRAND does not have an invariant inference mechanism, thus it can be interesting to use STRAND as the logic \( \mathcal{L} \) in PDR\(^\prime\). 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 conjuncted with the \( \ast \) 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

PDR\(^\prime\) 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 PDR 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


Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. 2012. Cubicle: A Parallel SMT-Based Model
Verification (CAV 2012), Vol. 7358. Springer, 718–724. DOI: 10.1007/978-3-642-31424-7_55

ism.acm.org/citation.cfm?id=2157675

SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL ’75). ACM, New York, NY, USA, 160–166. DOI: 10.1145/512976.512993


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: 10.1145/1512950.1512973

Satyaki Das and David L. Dill. 2002. Counter-Example Based Predicate Discovery in Predicate Abstraction. In Proceedings of


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: 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


Directed Reachability for Proving Absence of Concurrent Modification Errors. In Proceedings of the 18th International

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


A. THE INVARIANT OBTAINED FOR \textsc{split()}

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

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