Counterexample-Guided Focus

Andreas Podelski
University of Freiburg
podelski@informatik.uni-freiburg.de

Thomas Wies
EPFL and University of Freiburg
thomas.wies@epfl.ch

Abstract

The automated inference of quantified invariants is considered one of the next challenges in software verification. The question of the right precision-efficiency tradeoff for the corresponding program analyses here boils down to the question of the right treatment of disjunction below and above the universal quantifier. In the closely related setting of shape analysis one uses the focus operator in order to adapt the treatment of disjunction (and thus the efficiency-precision tradeoff) to the individual program statement. One promising research direction is to design parameterized versions of the focus operator which allow the user to fine-tune the focus operator not only to the individual program statements but also to the specific verification task. We carry this research direction one step further. We fine-tune the focus operator to each individual step of the analysis (for a specific verification task). This fine-tuning must be done automatically. Our idea is to use counterexamples for this purpose. We realize this idea in a tool that automatically infers quantified invariants for the verification of a variety of heap-manipulating programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages–Program Analysis

General Terms Algorithms, Languages, Reliability, Verification

Keywords Data Structures, Quantified Invariants, Predicate Abstraction, Abstraction Refinement, Shape Analysis

1. Introduction

There is a considerable interest in the automated verification of correctness properties of programs that implement or use heap-allocated data structures [6, 10, 19–21, 23, 24, 27, 29, 31, 34, 42, 43, 45, 48–50, 56, 57]. The correctness properties of such programs typically include quantified assertions, e.g., assertions that describe the expected shape of data structures such as *"the reference count for each internal object of the data structure is 1"* and assertions that describe the effect of data structure operations such as *"all objects stored in the data structure are properly initialized"*. The automated inference of quantified invariants for the verification of quantified assertions is considered one of the next challenges in software verification.

POPL'10, January 17–23, 2010, Madrid, Spain.

The eternal quest for the right precision/efficiency tradeoff in the corresponding program analyses here boils down to the question of the right treatment of disjunction below and above the universal quantifier (specifically in the construction of the abstract transformer). To achieve a reasonable average tradeoff between precision and efficiency, existing program analyses follow a common recipe (see, e.g., [19, 42, 48]). The designer of the program analysis starts off with a coarse but efficient generic abstract transformer and then manually adapts this abstract transformer to the individual kinds of program statements such that disjunctions are introduced prudently. In the closely related setting of shape analysis this adaptation is referred to as the *focus operation* [48]. If one has to adapt the abstract transformer to individual program statements, uniformly for all possible uses of the analysis, the designer of the program analysis is obliged to be very conservative with regard to the precision of the focus operator. As a consequence, the analysis is often too inefficient.

It is therefore desirable to fine-tune the focus operator to a specific use of the analysis. A promising research direction is to design parameterized focus operators that allow the user of the analysis to (manually) do this fine-tuning herself, for each new verification task; the resulting gain of scalability is encouraging; new classes of heap-manipulating programs, even concurrent ones, have been successfully analyzed [39–41].

In this paper, we carry this research direction one step further (and perhaps to its logical extreme). We propose to fine-tune the focus operator not only to each individual problem instance, but to each individual step of the analysis, i.e., each application of the abstract transformer. This fine-tuning must be done automatically. Our idea is to use counterexamples for this purpose. The contribution of this paper is to conceptually and practically realize the idea and to demonstrate its interesting potential. We present a new method and tool that automatically infers quantified invariants for the verification of a variety of heap-manipulating programs.

The general idea of using counterexamples for refining an abstraction stems from the classical scheme of counterexample-guided abstraction refinement (CEGAR) [3, 14, 25]. A number of software verification tools based on the scheme, e.g., [3, 12, 26, 44], are able to synthesize expressive invariants (though, not quantified ones, so far). A spurious counterexample is an error trace that is possible in the abstract but not in the concrete. Adding predicates extracted from a spurious counterexample refines the abstract domain; in a subsequent step, the scheme re-defines the abstract transformer (as a function on the new abstract domain). In contrast, the focus operator refines the effect of the abstract transformer without changing the abstract domain and without re-defining the function. We propose to use spurious counterexamples for both: for refining the abstract domain, and for fine-tuning a focus operator that adapts the effect of the abstract transformer.

More precisely, we transcend the idea of lazy abstraction [25], a CEGAR scheme which adapts the abstract domain exactly to the point of the execution of the analyzed program. We devise a *nested*

lazy abstraction refinement loop that adapts both, the abstract domain *and* the abstract transformer exactly to the point. The refinement loop consists of two nested refinement steps. The first step refines the abstract domain by extracting new predicates from the spurious error trace. If a spurious error trace is not eliminated by merely refining the abstract domain then the second step uses the spurious error trace to construct a focus operator that adapts the effect of the abstract transformer on the current abstract domain. This adaptation can thus be performed locally and lazily, i.e., anew for each single application of the abstract transformer.

In passing, let us note that our development of the counterexample-guided focus was originally motivated by the goal to establish the so-called *progress property* [25]. The property means that every spurious counterexample encountered during the analysis is eventually eliminated by a refinement step. The property is of foremost theoretical interest (a priori, it need not improve the chances of convergence in practice since there may be always a new spurious counterexample). In the setting of quantified invariants, however, the theoretical interest of counterexample-guided focus is in line with its practical relevance. The progress property holds (and only holds) in the presence of counterexample-guided focus. The practical verification does succeed *with* counterexample-guided focus and does not *without*; the reason is apparently that in many benchmarks, a uniformly precise abstract transformer with feasible cost comes with a too low precision.

Summary. To summarize, our work leverages the research on the CEGAR scheme in software verification [3, 12, 14, 25] and the research on the focus operator in shape analysis [39–41, 48]. Our contribution is to show that the techniques developed in these two research directions can be fruitfully integrated to enhance one another for the inference of quantified invariants:

- the focus operator can be made effective in a CEGAR setting because it can be fine-tuned lazily and its locality can be driven to the extreme,
- the CEGAR scheme can be made effective for quantified invariants because adding an inner refinement loop for the focus operator provides the progress property and the precision required on practical examples (without the otherwise prohibitive cost for a precise abstract transformer on an abstract domain for quantified assertions).

2. Motivating Example

In this section, we will use an example to motivate the counterexample-guided focus and explain it in more detail. The program INIT shown in Fig. 1 initializes all entries in a singly-linked list. The assert statement at location ℓ_2 checks that all entries in the list are indeed initialized after termination of the while loop. We would like to automatically compute an inductive invariant for the loop cutpoint at location ℓ_1 that implies the safety of this assertion.

In the remainder of this section, we will first present the abstract domain, then the most precise abstract transformer (which is too costly to implement), then a less costly abstract transformer (which is too coarse), and finally the transformer obtained by the counterexample-guided focus.

Abstract Domain: Boolean heaps. We use a variation of predicate abstraction [22] which we call *Boolean heap abstraction* [45]. Here we do not use *state* predicates (which can be defined by closed first-order formulas as, e.g., in the assert statement at location ℓ_2). Instead, we use predicates that range over objects in the heap (and which can be defined by formulas with free first-order variables). These predicates are reminiscent of the predicates used in three-valued shape analysis [48] and indexed predicate abstraction [31]. Figure 2 shows three such predicates for Program INIT. The predicates range over objects in the heap. For notational convenience we write predicates as sets. For instance, the predicate $Cont$ denotes the content of the list pointed to by x, i.e., the set of all non-null objects that are reachable from x by following $next$ pointers in the heap. Data structure fields such as $next$ and $data$ are modeled as function symbols. The binary relation $next^*$ denotes the reflexive transitive closure of the function $next$. Given a program state (i.e., a valuation of $next$ and $data$ as functions), the finite set of predicates induces a partition of the heap into finitely many equivalence classes of heap objects.

ℓ_0: $y := x$
ℓ_1: **while** $y \neq \mathsf{null}$ **do**
  $y.data := 0$
  $y := y.next$
ℓ_2: **assert** $(\forall v.\, (x,v) \in next^* \wedge v \neq \mathsf{null} \rightarrow v.data = 0)$

Figure 1. Program INIT

$$Cont = \{\, v \mid (x,v) \in next^* \wedge v \neq \mathsf{null} \,\}$$
$$Iter = \{\, v \mid (y,v) \in next^* \wedge v \neq \mathsf{null} \,\}$$
$$Init = \{\, v \mid v.data = 0 \,\}$$

Figure 2. Predicates for Program INIT denoting sets of objects

The abstract domain of Boolean heaps consists of formulas that describe such partitions. Each formula in the abstract domain is a disjunction of universally quantified Boolean combinations of the given predicates. The abstract domain is finite. We call the outer disjuncts in these formulas *abstract states*. For instance, the formula F given by

$$F \;\equiv\; \forall v.\;\; (v \in Iter \rightarrow v \in Cont) \;\wedge\; (v \in Cont \rightarrow v \in Iter \vee v \in Init)$$

is an abstract state for the predicates in Fig. 2. An abstract state is a special case of an element of the abstract domain (a disjunction with only one disjunct).

We use elements of the abstract domain to express inductive invariants. For example, the formula F is an inductive invariant for location ℓ_1 in program INIT and implies that the assert statement at location ℓ_2 does not fail.

Most Precise Abstract Transformer. For the purpose of this exposition, we represent the most precise abstract transformer using an *abstract program* over abstract states; see Fig. 3. The abstraction of the concrete transformer for each basic block in the concrete program is translated to a statement in the abstract program. The abstract program has a program variable for each of the given predicates (carrying the same name, e.g., $Iter$, $Cont$). The program variables in the abstract program range over sets of objects.

Figure 3 shows the Boolean heap abstraction of program INIT for the predicates given in Fig. 2. Here "*" stands for the nondeterministic choice of a Boolean value. The statement **havoc** x stands for the nondeterministic assignment of program variable x. In the example, for the sake of simplicity, we implicitly assume that all lists are acyclic. We express the updates in the abstract program through logical formulas over unprimed and primed variables (which, as usual, model the pre and the post value for the update). For instance, the abstract transformer for the loop body of program INIT is given by

$$\begin{aligned}\forall v.\, &(v \in Cont \leftrightarrow v \in Cont') \wedge\\ &(v \in Init \rightarrow v \in Init') \wedge\\ &(v \in Iter' \rightarrow v \in Iter) \wedge\\ &(v \in Init' \wedge v \notin Init \leftrightarrow v \in Iter \wedge v \notin Iter').\end{aligned}$$

```
ℓ0 :  havoc Iter′
      assume Iter′ = Cont
      Iter := Iter′
ℓ1 :  while ∗ do
        havoc Iter′, Init′
        assume Init ⊆ Init′
        assume Iter′ ⊆ Iter
        assume Init′ − Init = Iter − Iter′
        (Iter, Init) := (Iter′, Init′)
      assume Iter = ∅
ℓ2 :  assert Cont ⊆ Init
```

Figure 3. Program ABSINIT: Boolean heap abstraction of Program INIT

```
ℓ0 :  havoc Iter′
      assume Iter′ = Cont
      Iter := Iter′
ℓ1 :  while ∗ do
        havoc Iter′, Init′
        assume Init ⊆ Init′
        assume Iter′ ⊆ Iter
        (Iter, Init) := (Iter′, Init′)
      assume Iter = ∅
ℓ2 :  assert Cont ⊆ Init
```

Figure 4. Program CARTABSINIT: Cartesian abstraction of Program ABSINIT

```
ℓ0 :  havoc Iter′
      assume Iter′ = Cont
      Iter := Iter′
ℓ1 :  while ∗ do
        havoc Iter′, Init′, Y
        assume Init′ = Init ∪ Y
        assume Iter′ = Iter − Y
        (Iter, Init) := (Iter′, Init′)
      assume Iter = ∅
ℓ2 :  assert Cont ⊆ Init
```

Figure 5. Counterexample-guided focus applied to Program CARTABSINIT

For readability, in the abstract program in Fig. 3, we decompose the abstract transformers into several assume statements and represent them as set constraints (instead of universally quantified formulas). Also, we omit some redundant information.

The successor abstract state under the execution of a basic block in the abstract program is obtained as one expects. First the abstract state is conjoined with the logical formula (in unprimed and primed variables) that represents the abstract transformer of the basic block. Then the unprimed variables are projected and the primed variables renamed by their unprimed version. The (finite) set of reachable abstract states for the abstract program ABSINIT represents an (inductive) invariant of the program INIT. Projected to location ℓ_1, this invariant implies the formula F. I.e., the corresponding analysis (which computes the set of reachable abstract states of ABSINIT) succeeds to prove the correctness of the program INIT.

Program ABSINIT represents the *most precise* abstract transformer with respect to the abstract domain induced by the given predicates. This abstraction is in general too expensive. First, the construction of the abstract program requires exponentially many theorem prover calls in the number of predicates. Second, the most precise abstraction often keeps track of more information than is necessary for proving a specific property and causes the analysis to explore unnecessarily large parts of the abstract domain.

Abstract Transformer with Cartesian Abstraction. In order to obtain an abstract transformer with feasible cost, one can apply the *Cartesian abstraction* [2, 16] on top of the Boolean heap abstraction. Cartesian abstraction is originally defined on abstract domains that are power sets of vectors (in our case bitvectors). Its name stems from the fact that it abstracts a set of vectors S by the smallest Cartesian product (of component sets) that contains S. It applies to our setting because we can represent an abstract state canonically as a set of bitvectors where each bitvector corresponds to an inner disjunct of the abstract state.

The Cartesian abstraction of the most precise abstract post (with respect to the Boolean heap abstraction) can be constructed effectively from the concrete program (in practice using only a polynomial number of theorem prover calls) [45]. Furthermore, Cartesian abstraction avoids an explosion of the size and number of abstract states that are explored in the fixed point computation by restricting the disjuncts that can appear below and above the universal quantifier in abstract states. The resulting analysis is efficient in practice. In general, however, it is too imprecise. This is demonstrated by our example program.

Figure 4 shows the Cartesian abstraction of Program ABSINIT. Program CARTABSINIT loses the correlation between the predicates $Iter$ and $Init$ in the abstract transformer of the loop body (the correlation is expressed by the statement **assume** $Init' - Init = Iter - Iter'$ in the program ABSINIT in Fig. 3). As a consequence, the invariant F of Program ABSINIT is not an invariant of Program CARTABSINIT. I.e., the corresponding analysis (which computes the set of reachable abstract states of CARTABSINIT) does not succeed to prove the correctness of the program INIT.

Counterexample-Guided Focus. The analysis of the abstract program CARTABSINIT produces a spurious error trace that witnesses a violation of the assert statement at location ℓ_2. It seems tempting to try to use the spurious error trace to refine the abstraction as done in the CEGAR scheme. As mentioned above, in the existing CEGAR scheme, the abstract domain is refined; the desired effect is to eliminate the spurious error trace. The underlying assumption which guarantees this effect is that the refined analysis is based on the most precise abstract transformer. In our case, however, the spurious error trace results from an imprecise abstract transformer, rather than an imprecise abstract domain. In fact, the abstract domain is already able to express an inductive invariant that is sufficiently strong to rule out all spurious error traces. An attempt to extract new predicates from the proof of spuriousness of the counterexample is therefore pointless. It makes more sense to use the spurious error trace for refining the abstract transformer, rather than the abstract domain.

To do so, we use the above-mentioned focus operator that is inspired by shape analysis. It refines the image of the abstract transformer by changing the presentation of its pre-image. We do not change the definition of the abstract transformer (with the Cartesian abstraction). However, when applied to the new presentation, the Cartesian abstraction does not lose as much precision as before. The novelty of our approach is that we devise a focus operator that is constructed on-demand and locally from the spurious error trace.

Figure 5 illustrates the effect of our counterexample-guided focus operator on Program CARTABSINIT. It only affects the body of the while loop. The focus operator uses an additional predicate $Y = \{ v \mid y = v \}$ in order to split disjuncts below the universal quantifier into multiple disjuncts before the application of the abstract transformer for the loop body. As a consequence, the correlation between variables $Iter$ and $Init$ is not lost in spite of the Cartesian abstraction of the abstract transformer. Note the different, but equivalent representation of the correlation by two assume statements. Our method and tool automatically infers the split predicate Y and the corresponding focus operator from a spurious counterexample of the abstract program CARTABSINIT. The inductive invariant obtained by the fixed point of the resulting abstract transformer implies the assertion at location ℓ_2. I.e., the analysis (which computes the set of reachable abstract states of the abstract program in Fig. 5) does succeed to prove the correctness of Program INIT.

3. Preliminaries

We now formalize the notion of formulas and programs used in this paper and formally introduce Boolean heap abstraction.

3.1 Logics and Programs.

Logical formulas and structures. For reasoning about programs we consider formulas in a sorted logic $\mathcal{L}$. We require that $\mathcal{L}$ provides the sorts obj (heap objects) and loc (program locations). Furthermore, we require that $\mathcal{L}$ provides logical constructs for equality over both sorts, Boolean connectives, and first-order quantification over variables of sort obj. Formulas are expressed over a signature Σ where Σ consists of the following constant symbols of sort loc: $\ell \in \mathit{Locs}$ (control locations), pc (the program counter), ℓ_0 (the initial location), and ℓ_E (the error location), as well as the following symbols of sort obj: unary function symbols $f \in \mathit{Flds}$ (data structure fields) and constant symbols $x \in \mathit{Vars}$ (program variables). We leave out sort annotations in formulas whenever this causes no confusion. In our running examples, the logic $\mathcal{L}$ is given by first-order logic with transitive closure.

We fix non-empty disjoints sets $\mathcal{O}$ and $\mathcal{L}$ for the interpretations of sort obj and loc. A Σ-structure $\mathcal{A}$ is a first-order interpretation that interprets each symbol in Σ by a function over, respectively, an element in the sets interpreting the associated sorts. We write $\mathcal{A}(f)$ for the interpretation of symbol $f \in \Sigma$ in structure $\mathcal{A}$. We further write $\mathcal{A}(t)$ for the denotation of a Σ-term t in structure $\mathcal{A}$. Hereby, $\mathcal{A}$ also provides interpretations for the free variables occurring in t. We use standard logical notation for satisfiability, validity, and entailment. Finally, for a formula F we write $[\![F]\!]$ to denote the set of all structures that satisfy F.

Programs. The set of commands Coms is defined by the following grammar where F is a formula in $\mathcal{L}$, $x, y \in \mathit{Vars}$ denote program variables, $f \in \mathit{Flds}$ a pointer field, and $\ell \in \mathit{Locs} \cup \{\ell_0, \ell_E\}$ a control location:

$$c ::= \quad c\,;c \mid \textbf{assume}\ F \mid pc{:=}\ell \mid x{:=}y \mid x{:=}y.f \mid x.f{:=}y$$

A *program* P is a finite set of commands. Consecutiveness of commands in a program is achieved by composing assume statements on the value of pc, updates of pc, and the actual commands using the sequential composition operator.

Program states. A program *state* is a Σ-structure. We denote by States the set of all program states. We call a state s *initial state* iff it satisfies $s \models pc{=}\ell_0$ and we call it *error state* iff it satisfies $s \models pc{=}\ell_E$. Let init be the formula $pc{=}\ell_0$ denoting all initial states and let safe be the formula $pc{\neq}\ell_E$ denoting all non-error states.

Null pointers and allocation. Note that we interpret data structure fields f as total functions. We treat null as a program variable that can neither be assigned nor dereferenced. We assume that for all fields $f \in \mathit{Flds}$ the equality $\mathsf{null}.f = \mathsf{null}$ holds in all program states. In order to ensure absence of null dereferences, every command c that contains a dereference of the form $x.f$ is guarded by a command **assume** $(x{=}\mathsf{null})\,; pc{:=}\ell_E$ that directs control to the error location if x is not defined. Allocation of fresh heap objects can be modelled by introducing a predicate symbol that keeps track of the current set of allocated objects. However, this requires the inclusion of **havoc** commands that nondeterministically update the value of a program variable. The techniques presented in this paper carry over to programs extended with **havoc** commands. For details see [51].

Transition relations. Each command c represents a relation $[\![c]\!]$ on pairs of states (s, s') that is defined recursively on the structure of commands as follows:

- If c is a sequential composition $c_1\,; c_2$ then there must exist a state s'' such that $(s, s'') \in [\![c_1]\!]$ and $(s'', s') \in [\![c_2]\!]$.
- If c is an assume command **assume** F, we require that $s \models F$ and $s' = s$.
- If c is an update of the program counter or a program variable, we have $s' = s[pc{\mapsto}s(\ell)]$ for $c = (pc{:=}\ell)$, $s' = s[x \mapsto s(y)]$ for $c = (x{:=}y)$ and $s' = s[x \mapsto s(f(y))]$ for $c = (x{:=}f.y)$.
- Finally, if c is a field update of the form $f.x{:=}y$, we have $s' = s[f \mapsto s(f)[s(x){\mapsto}s(y)]]$.

Computations and traces. A program *computation* of a program P is a (possibly infinite) sequence $\sigma = s_0 \xrightarrow{c_0} s_1 \xrightarrow{c_1} \ldots$ of states and commands such that s_0 is an initial state and for each pair of consecutive states s_i and s_{i+1} we have $(s_i, s_{i+1}) \in [\![c_i]\!]$ for some command $c_i \in P$. A *trace* is a sequence of commands and we call the projection of a computation $\sigma = s_0 \xrightarrow{c_0} s_1 \xrightarrow{c_1} \ldots$ to the sequence of commands $c_0 c_1 \ldots$ the trace of that computation. A trace is called *error trace* if it is the trace of some computation that reaches an error state. A program is *safe* if it has no error traces.

Predicate transformers. Given a set of states S and a binary relation R on states, we define strongest postcondition post and weakest (liberal) precondition wlp as usual:

$$\mathsf{post}(R)(S) \stackrel{\text{def}}{=} \left\{\, s' \mid \exists s.\, (s, s') \in R \wedge s \in S \,\right\}$$
$$\mathsf{wlp}(R)(S) \stackrel{\text{def}}{=} \left\{\, s \mid \forall s'.\, (s, s') \in R \Rightarrow s' \in S \,\right\} \quad .$$

We further introduce symbolic weakest preconditions on formulas. For any command c and formula F the formula $\mathsf{wlp}(c)(F)$ is a formula such that we have $\mathsf{wlp}([\![c]\!])([\![F]\!]) = [\![\mathsf{wlp}(c)(F)]\!]$. Note that we do not require F to be closed. We extend symbolic weakest preconditions from commands to sequences of commands as expected.

3.2 Boolean Heap Abstraction

We formalize the Boolean heap abstraction in terms of an abstract interpretation [15]. The concrete domain is given by sets of states ordered by set inclusion. We represent elements of the concrete domain by closed formulas in $\mathcal{L}$. The concrete fixed point functional is the operator post for the transition relation of the concrete program (i.e., the union of the transition relations of all its commands) and the initial states init. The abstract domain is a finite set of formulas that forms a sublattice of the concrete domain. The analysis is to compute the least fixed point of an abstraction of post that is defined on the abstract domain. The computed least fixed point is an inductive invariant of the concrete program.

Abstract domain. The abstract domain is parameterized by a finite set of predicates that denote sets on heap objects in a given state. In the following, we fix a particular finite set of predicates $\mathcal{P}$. We consider $\mathcal{P}$ to be given by a set of (closed) lambda terms of the form $\lambda v.\, G$ where G is a formula in $\mathcal{L}$, i.e., each predicate $p \in \mathcal{P}$ denotes a set of objects in a given state. If the formula G is itself closed we call the corresponding predicate *state predicate*. We denote by $\mathcal{P}(v)$ the set of formulas obtained by beta reduction of all formulas $p(v)$ for $p \in \mathcal{P}$. For notational convenience we assume that $\mathcal{P}$ is such that for all v the set $\mathcal{P}(v)$ is closed under negation. The following definitions are implicitly parameterized by the set $\mathcal{P}$.

The abstract domain AbsDom over $\mathcal{P}$ consists of all formulas of the form

$$\bigvee_{i=1}^{n} \forall v.\ \bigvee_{j_i=1}^{m_i} D_{j_i}(v)$$

where each $D_{j_i}(v)$ is a conjunction of formulas in $\mathcal{P}(v)$. We call the outer disjuncts of these formulas *abstract states* and the inner disjuncts *abstract objects*. We identify formulas up to logical equivalence. The partial order on the abstract domain is given by

the logical entailment relation "$\models$". Note that $AbsDom$ is finite (modulo logical equivalence) and closed under both conjunction and disjunction. Thus, it forms a complete lattice. The abstract domain can be easily generalized to formulas with quantification over more than one variable and predicates that denote relations on objects rather than just sets [51].

Abstraction function. The abstraction function α that maps a set of states represented by a closed formula F to a formula in the abstract domain is defined as follows

$$\alpha(F) \stackrel{\text{def}}{=} \bigwedge \left\{ F^{\#} \in AbsDom \mid F \models F^{\#} \right\} .$$

The function α is the lower adjoint of a Galois connection (α, γ) between the concrete and abstract domain, with γ being the identity function.

Abstract post operator. The most precise abstract post operator on the abstract domain of Boolean heaps $\mathsf{post}^{\#}_{\mathsf{BH}}$ and a command c is given by composition of the concrete post operator for c with the Galois connection (α, γ). The actual abstract post operator that we use in the fixed point computation of the analysis is an abstraction of $\mathsf{post}^{\#}_{\mathsf{BH}}$. We denote this operator by $\mathsf{post}^{\#}_{\mathsf{C{\cdot}BH}}$ and call it the *Cartesian abstract post operator*. Formally, the operator $\mathsf{post}^{\#}_{\mathsf{C{\cdot}BH}}$ is defined as a Cartesian abstraction of the operator $\mathsf{post}^{\#}_{\mathsf{BH}}$. In the following, we only show how $\mathsf{post}^{\#}_{\mathsf{C{\cdot}BH}}$ is computed. For further details see [45, 51].

We allow abstract states in the pre and post-images of operator $\mathsf{post}^{\#}_{\mathsf{C{\cdot}BH}}$ to range over different sets of predicates $\mathcal{P}_1$, respectively, $\mathcal{P}_2$. Let c be a command and $F^{\#}$ an abstract state over predicates $\mathcal{P}_1$ of the form

$$F^{\#} = \forall v. \bigvee_{j=1}^{m} D_j(v)$$

where the disjuncts D_j are monomials, i.e., each predicate in $\mathcal{P}_1$ occurs either positive or negative in each D_j. The operator $\mathsf{post}^{\#}_{\mathsf{C{\cdot}BH}}$ maps $F^{\#}$ to a single abstract state $F'^{\#}$ by mapping each disjunct D_j in $F^{\#}$ to a single disjunct D'_j in $F'^{\#}$. The mapping guarantees that if $s \in States$ is a concrete state that satisfies $F^{\#}$ and $o \in \mathcal{O}$ an object that satisfies disjunct D_j in s then for every c-successor s' of s, o satisfies D'_j in s'. Since this property holds for all objects o, every c-successor s' of s satisfies $F'^{\#}$. Formally, the image of $F^{\#}$ under the abstract post operator $\mathsf{post}^{\#}_{\mathsf{C{\cdot}BH}}$ for command c is given by:

$$\mathsf{post}^{\#}_{\mathsf{C{\cdot}BH}}[\mathcal{P}_1, \mathcal{P}_2](c)(F^{\#}) = \\ \forall v. \textstyle\bigvee_{j=1}^{m} \bigwedge \left\{ p(v) \in \mathcal{P}_2(v) \mid F^{\#} \wedge D_j(v) \models \mathsf{wlp}(c)(p(v)) \right\}$$

Thus, the image of the Cartesian abstract post is computed by checking entailments between conjunctions of predicates and weakest preconditions of predicates. The quantified formula $F^{\#}$ in the antecedent of these entailments can be replaced by any weaker formula, e.g., a conjunction of finitely many instantiations of $F^{\#}$. The operator $\mathsf{post}^{\#}_{\mathsf{C{\cdot}BH}}$ is extended to disjunctions of abstract states as expected.

4. Counterexample-Guided Focus

Before we formally define the counterexample-guided focus operator, it is instructive to fully understand the nature of the loss of precision that is induced by Cartesian abstraction.

Recall Program Init from Section 2. The left part of Figure 6 shows a program state s that may occur at location ℓ_1 during execution of Program INIT. The boxes represent abstract objects over the sets $Cont$, $Init$, and $Iter$. Hereby S^c stands for the set

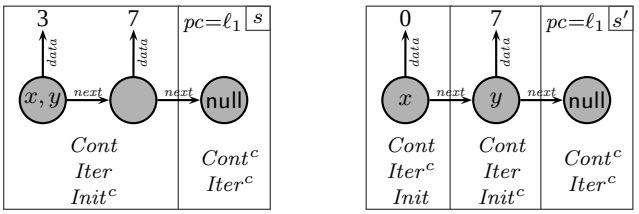

Figure 6. A reachable program state s of Program INIT and its successor state s' that is obtained after execution of the loop body.

complement of S. Formally the state s satisfies the abstract state

$$F^{\#} = \forall v. v \in Cont \wedge v \in Iter \wedge v \notin Init \vee v \notin Cont \wedge v \notin Iter$$

The right part of Figure 6 shows the post-state s' of s that is obtained at location ℓ_1 after execution of the loop body. The boxes indicate again the abstract objects associated with the concrete objects. The abstract state consisting of the disjunction of these abstract objects is the image of the most precise abstract post $\mathsf{post}^{\#}_{\mathsf{BH}}$ of the abstract state $F^{\#}$. Let $F'^{\#}$ be this abstract post-state. Note that the concrete objects in s that are represented by the abstract object

$$v \in Cont \wedge v \in Iter \wedge v \notin Init$$

end up in two disjoint abstract objects in $F'^{\#}$. The Cartesian abstract post operator *merges* these two disjuncts into a single conjunction that only contains the predicates on which both disjuncts agree, namely, $v \in Cont$. The correlations between predicates $Init$ and $Iter$ in the two inner disjuncts of $F'^{\#}$ are lost.

If we want to adapt the precision of the Cartesian abstract post we need to prevent it from merging disjuncts in the post-image of the most precise abstract post. This adaptation is performed by our focus operator. The focus operator adapts the precision of the Cartesian abstract post indirectly. Namely, it refines the abstract domain of the pre-image and splits disjuncts (i.e., both abstract states and abstract objects in abstract states) in the pre-image into more fine-grained disjunctions. The splitting ensures that individual disjuncts in the refined pre-image are mapped to individual disjuncts in the post-image under the most precise abstract post. This effectively prevents Cartesian abstraction from losing precision. Both the refinement of the abstract domain and the splitting of disjuncts are guided by spurious error traces that are produced by the analysis.

Counterexample-guided focus. We now formally define the counterexample-guided focus operator. In the following we fix a program P and a set of predicates $\mathcal{P}$. An *abstract computation* $\sigma^{\#}$ is a sequence

$$F_0^{\#} \xrightarrow{op_0} F_1^{\#} \xrightarrow{op_1} \dots \xrightarrow{op_{n-1}} F_n^{\#}$$

where the $F_i^{\#}$ are elements of the abstract domain $AbsDom[\mathcal{P}]$ and the op_i are abstract transformers $AbsDom[\mathcal{P}] \to AbsDom[\mathcal{P}]$. Moreover, the following two conditions hold: (1) $F_0^{\#} = \alpha[\mathcal{P}](\mathsf{init})$ and (2) for all i between 0 and $n-1$, $F_{i+1}^{\#} = op_i(F_i^{\#})$. We say that the abstract computation *ends in an error state* if $F_n^{\#} \not\models \mathsf{safe}$. We say that $\sigma^{\#}$ is *generated* by a trace $\pi = c_0 \dots c_{n-1}$ and an operator $op \in Coms \to AbsDom \to AbsDom$ if for all i, $op_i = op(c_i)$. We say that the generated abstract computation is *sound* if for all i between 0 and $n-1$, $F_{i+1}^{\#}$ is an over-approximation of the set of states that are reachable from an initial state by following the trace $c_0 \dots c_i$. Finally, we say that a trace π is a *spurious error trace* for op, if the abstract computation generated by π and op ends in an error state, yet, π is not an error trace of program P.

The counterexample-guided focus operator is used to eliminate spurious error traces for the Cartesian abstract post that are not

spurious error traces for the most precise abstract post. Let $\pi_0 = c_0 \dots c_{n-1}$ be such a trace.

Note that concrete error traces can be characterized in terms of symbolic weakest preconditions.

LEMMA 1. *A trace π is an error trace iff* $\mathsf{init} \not\models \mathsf{wlp}(\pi)(\mathsf{safe})$.

Since π_0 is not a concrete error trace, we know that π_0 satisfies

$$\mathsf{init} \models \mathsf{wlp}(\pi_0)(\mathsf{safe}) \tag{1}$$

Let $\mathsf{pref}_i(\pi_0)$ be the prefix of π_0 up to command c_{i-1}, respectively, $\mathsf{suff}_i(\pi_0)$ its suffix starting from command c_i. From (1) and the properties of predicate transformers post and wlp follows that for all i between 0 and $n-1$ we have

$$\mathsf{post}([\![\mathsf{pref}_i(\pi_0)]\!])([\![\mathsf{init}]\!]) \subseteq \mathsf{wlp}([\![\mathsf{suff}_i(\pi_0)]\!])([\![\mathsf{safe}]\!]) \tag{2}$$

In other words, for each i the formula $\mathsf{wlp}(\mathsf{suff}_i(\pi_0))(\mathsf{safe})$ is satisfied in all states that are reachable from an initial state by following the trace $\mathsf{pref}_i(\pi_0)$. The idea of our counterexample-guided focus operator is to use the formulas $\mathsf{wlp}(\mathsf{suff}_i(\pi_0))(\mathsf{safe})$ to guide the splitting of disjuncts in the pre-images of the Cartesian post operator.

The counterexample-guided focus operator focus takes a sequence of commands π (the suffix of a spurious error trace π_0) and an element of the abstract domain $AbsDom[\mathcal{P}]$ as arguments and maps the latter to an element of a refined abstract domain $AbsDom[\mathsf{preds}(\pi)]$ with $\mathcal{P} \subseteq preds(\pi)$. The operator focus is defined as follows

$$\mathsf{focus}(\pi)(F^\#) \stackrel{\text{def}}{=} \alpha[\mathsf{preds}(\pi)](\mathsf{wlp}(\pi)(\mathsf{safe})) \wedge F^\#$$

The set of predicates $\mathsf{preds}(\pi)$ is the union of the predicates $\mathcal{P}$ and predicates that are extracted from the weakest precondition of safe with respect to π. More precisely, if $\pi = c\pi'$ then preds extracts all atoms from the formula

$$\mathsf{wlp}(c)(\alpha[\mathcal{P}](\mathsf{wlp}(\pi')(\mathsf{safe}))) \tag{3}$$

The *adapted* Cartesian abstract post operator $\mathsf{post}^\#_{\mathsf{f{\cdot}C{\cdot}BH}}(\pi)$ for the suffix π of some spurious error trace is obtained by composition of the Cartesian post operator (for the refined pre-image domain) with the focus operator:

$$\mathsf{post}^\#_{\mathsf{f{\cdot}C{\cdot}BH}}(\pi) \stackrel{\text{def}}{=} \lambda c.\, \mathsf{post}^\#_{\mathsf{C{\cdot}BH}}[\mathsf{preds}(\pi), \mathcal{P}](c) \circ \mathsf{focus}(\pi)$$

Let $\sigma^\#$ be the abstract computation generated from path π_0 and the sequence of operators $[\mathsf{post}^\#_{\mathsf{f{\cdot}C{\cdot}BH}}(\mathsf{suff}_i(\pi))(c_i)]_{0 \le i < n}$.

PROPOSITION 2 (Soundness of Focus). *The abstract computation $\sigma^\#$ is sound.*

The proof of Proposition 2 follows from Property (2) and the fact that $\mathsf{post}^\#_{\mathsf{C{\cdot}BH}}$ is a sound approximation of the most precise abstract post operator $\mathsf{post}^\#_{\mathsf{BH}}$.

PROPOSITION 3 (Progress of Focus). *The abstract computation $\sigma^\#$ does not reach an error state.*

The proof of Proposition 3 relies on the fact that the trace π_o of computation $\sigma^\#$ is not an error trace and not spurious for the most precise abstract post. One can then show that focus performs sufficient splitting of disjuncts in abstract states of $\sigma^\#$ before each application of the Cartesian abstract post. This splitting ensures that Cartesian abstraction causes no loss of information that is crucial for proving that π_o is safe. Note, however, that the focused Cartesian abstract post is not guranteed to compute the most precise abstract post for the given trace. Its precision lies between the plain Cartesian abstract post and the most precise abstract post.

Practical considerations. In our actual analysis the focus operator is always applied in a very specific situation, namely, when the abstract domain for the post states of the refined abstract transformer already contains all the predicates that can be extracted from the spurious error trace used for the focus. Therefore the abstraction function α in (3) can be replaced by the identity function. The resulting focus operator is then polynomial in the number of extracted predicates and the size of the representation of the focused abstract states.

5. Additional Examples

We now illustrate how the counterexample-guided focus adapts the abstract transformers for the abstractions of three example programs. In all of these examples, the analysis without counterexample-guided focus would not be able to infer a sufficiently strong invariant that proves the correctness of the program.

***Program* INIT.** We first revisit Program INIT from Section 2. We explained the nature of the loss of precision under Cartesian abstraction for this example program in the previous section. This loss of precision causes that the analysis of Program INIT produces spurious error traces even though the abstract domain can express a sufficiently strong inductive invariant. The shortest such spurious error trace is the trace that starts with the commands at location ℓ_0 executes the while loop once and then goes to the error location via the failing assert statement at location ℓ_2. The weakest precondition $\mathsf{wlp}(\pi)(\mathsf{safe})$ for the suffix π of this spurious error trace that starts in location ℓ_1 is given by the formula

$$\begin{aligned} &y.next{=}\mathsf{null} \wedge y{\neq}\mathsf{null} \rightarrow \\ &\quad (\forall v.\, (x, v) \in next^* \wedge v{\neq}\mathsf{null} \rightarrow v.data{=}0 \vee y{=}v) \end{aligned}$$

Using this formula, the focus operator refines the pre-image of the abstract transformer for the loop body by adding, among other predicates, the predicate $Y = \{\, v \mid y{=}v \,\}$. In this particular example, simply adding predicate Y to the pre-image domain is sufficient to rule out the spurious error trace. Recall that the image of the Cartesian abstract post operator is computed by considering a normal form of the pre-image where in each inner disjunct every predicate occurs either positively or negatively. Thus, refining the abstract domain by adding predicate Y already enables the Cartesian abstract post to perform the necessary splitting of disjuncts in the original pre-image. However, this splitting is purely syntactic. Some of the split disjuncts might be unsatisfiable in all represented pre-states but might be mapped to satisfiable disjuncts in the post-image and, thus, cause imprecision. Also, without proper focus the image of the Cartesian abstract post will always be a single abstract state and never a proper disjunction. Our second example shows that, in general, the local refinement of the abstract domain of the pre-image alone, does not suffice to eliminate a spurious error trace for the Cartesian abstract post.

***Program* LISTREVERSE.** Consider program LISTREVERSE in Figure 7 that performs an in-place reversal of a singly-linked list. The list is pointed to by program variable r. Assume the heap is sharing-free before execution of the program (the first assume statement at location ℓ_0) and assume that r points to the actual root node of the list (the second assume statement at location ℓ_0). We would like to verify that under these assumptions r points again to the root node of the reversed list after termination of the program.

One part of the inductive invariant for location ℓ_1 that is needed to verify the assertion at location ℓ_2 is given by the formula

$$e{=}\mathsf{null} \vee (\forall v.\, v.next{\neq}e)$$

This formula expresses that e always points to the root of the part of the original list that has yet to be reversed. This formula can be

ℓ_0 : **assume** $(\forall u\, v\, w.\, v.next{=}w \wedge u.next{=}w \wedge v{\neq}u \rightarrow w{=}\mathsf{null})$
assume $(r{=}\mathsf{null} \vee (\forall v.\, v.next{\neq}r))$
$e{:=}\, r;\ r{:=}\,\mathsf{null}$
ℓ_1 : **while** $e{\neq}\mathsf{null}$ **do**
$t{:=}\, e;\ e{:=}\, e.next$
$t.next{:=}\, r$
ℓ_2 : **assert** $(r{=}\mathsf{null} \vee (\forall v.\, v.next{\neq}r))$

Figure 7. Program LISTREVERSE

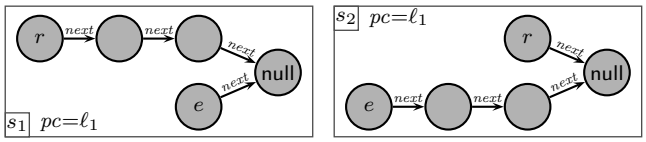

Figure 8. Two reachable states of Program LISTREVERSE

expressed by the disjunction of the following two abstract states

$$F_1^{\#} : \forall v.\, e{=}v \leftrightarrow \mathsf{null}{=}v$$
$$F_2^{\#} : \forall v.\, v.next{\neq}e$$

Figure 8 shows two states that may occur at location ℓ_1 during execution of the while loop. Both states satisfy the abstract state $F_2^{\#}$. Note that after execution of the loop body, the state s_1 satisfies only abstract state $F_1^{\#}$ while the second state satisfies only abstract state $F_2^{\#}$. The Cartesian abstract post operator will always merge the post states of $F_2^{\#}$ that result from execution of the loop body into a single abstract state. An analysis based on this operator therefore cannot infer a sufficiently strong inductive invariant. The counterexample-guided focus causes $F_2^{\#}$ to be split into two abstract states, one whose post states are covered by $F_1^{\#}$ and one whose post states are covered by $F_2^{\#}$. Thus, the Cartesian abstract post will not lose precision on the focused pre-image. Refining the abstract domain by adding additional predicates will not make up for the loss of precision on the unfocused pre-image, unless one adds a *state predicate* that expresses one of the outer disjuncts in terms of an inner disjunct of the other (e.g., the predicate $e{=}\mathsf{null}$ in the given example). In general, the state predicates that one needs to add to prevent loss of precision under Cartesian abstraction can be arbitrarily complex quantified formulas.

***Program* DLISTERASE.** The splitting of disjuncts that is performed by our counterexample-guided focus operator is closely related to *materialization* in shape analysis. Materialization refers to an intermediate step in the computation of the abstract post where a concrete object is extracted from an abstraction of a collection of objects. For instance, given an abstraction of a list, one needs to extract the head of the list in order to compute a precise abstract post-state for a command that iterates over the list. Our third example demonstrates that counterexample-guided focus performs materialization automatically, even in cases that require data-structure-specific manual adaptation of the abstract transformers in many existing shape analyses.

Consider Program DLISTERASE shown in Figure 9. This program erases all entries in an acyclic doubly-linked list. The list is pointed to by program variable r. The loop that erases the entries in the list iterates backwards over the data structure starting from the last entry l. In each iteration both the forward pointer $next$ and the backward pointer $prev$ of the current iterate are set to null. The task is to verify that the $prev$ and $next$ fields of all original list entries have indeed been set to null after the loop terminates. This property is expressed by the assert statement at location ℓ_2. The assume statements at location ℓ_0 express the precondition of the program.

ℓ_0 : **assume** $(\forall v\, w.\, v.prev{\neq}\mathsf{null} \wedge w.next{\neq}v \vee v.prev.next = v)$
assume $(\forall v.\, v \in Cont_0 \leftrightarrow (r, v) \in next^* \wedge v{\neq}\mathsf{null})$
assume $l \in Cont_0$
assume $l.next{=}\mathsf{null}$
ℓ_1 : **while** $l{\neq}\mathsf{null}$ **do**
$l.next{:=}\,\mathsf{null}$
$t{:=}\, l;\ l{:=}\, l.prev$
$t.prev{:=}\,\mathsf{null}$
ℓ_2 : **assert** $(\forall v.\, v \in Cont_0 \rightarrow v.next{=}\mathsf{null} \wedge v.prev{=}\mathsf{null})$

Figure 9. Program DLISTERASE

The first assume statement expresses that field $prev$ is the inverse of field $next$ which implies that the list r is doubly-linked. The second assume statements defines the set $Cond_0$, a *ghost variable* that denotes the set of elements that are originally stored in the list. The third and fourth assume statement together ensure that l points to the last entry in the list.

The important part of the inductive invariant at location ℓ_1 that is strong enough for proving the assertion at location ℓ_2 is given by the following formula

$$\forall v.\, v \in Cont_0 \wedge (l, v) \notin prev^* \rightarrow v.next{=}\mathsf{null} \wedge v.prev{=}\mathsf{null}$$

In order to infer this formula, the abstract transformer for the loop body needs to split some of the inner disjuncts that contain positive occurrences of the predicate $(l, v) \in prev^*$ in order to keep precise information about the object pointed to by program variable l in each iteration. This splitting corresponds to materialization from the back of the doubly-linked list. Many shape analyses, e.g., those based on separation logic [13, 19, 38], need special hand-crafted rules to perform materialization for specific data structures. With counterexample-guided focus, the abstract transformer is automatically adapted to perform materialization. The adaptation mechanism is independent of the data structures that the analyzed program manipulates and is only applied when the extra precision is needed to prove a particular property.

6. Lazy Nested Abstraction Refinement

We now present our lazy nested abstraction refinement loop that integrates lazy counterexample-guided refinement of the abstract domain and lazy adaptation of the abstract transformer via counterexample-guided focus.

The refinement loop is shown in Figure 10. The procedure LazyNestedRefine takes a program P as input and constructs an abstract reachability tree (ART) in the spirit of lazy abstraction [25]. An ART is a tree where each node r is labeled by a set of predicates $r.preds$ and abstract states $r.states$ in $AbsDom[r.preds]$. The root node r_0 of the ART is labeled by an abstract state denoting the set of all initial states. Each edge in the ART is labeled by a command c in program P and an abstract transformer op for the command c. We write $r \xrightarrow{c,op} r'$ to denote that there is an edge in the ART from node r to node r' which is labeled by c and op. Furthermore, we write $r \xrightarrow{\pi}{}^* r'$ to indicate that there is a (possibly empty) path from r to r' in the ART that is labeled by the trace π. Each path in the ART starting from the root node corresponds to an abstract computation that is generated from the trace labelling the path.

The lazy nested abstraction refinement algorithm iteratively extends and refines the ART until either a fixed point is reached, i.e., the disjunction of the abstract states contained in all ART nodes is an inductive invariant of program P, or until an error trace has been constructed. If a spurious error trace is encountered during the fixed point computation then this trace is used to refine the abstraction. We now describe the algorithm in detail.

```
proc LazyNestedRefine(P: program)
  begin
    let r_0 = ⟨preds: {init}, states: init, covered: false⟩
    let succ(r) =
      let Succ = ∅
      for all c ∈ T do
        let r' = ⟨preds: ∅, states: false, covered: false⟩
        let op = post#_{C.BH}(c)
        add edge r --c,op--> r'
        Succ := Succ ∪ {(r, op, r')}
      return Succ
    let U = succ(r_0)
    while U ≠ ∅ do
      choose and remove (r_1, op, r_2) ∈ U
      r_2.states := op[r_1.preds, r_2.preds](r_1.states)
      if r_2.states ⊨ ⋁_r { r.states | r ≠ r_2 } then
        r.covered := true
      else if r_2.states ⊨ safe then U := U ∪ succ(r')
      else let r_s, π such that π is maximal trace with r_s --π-->* r_2 ∧
                                  r_s.states ⊭ wlp(π)(safe)
        if r_s = r_0 then return counterexample(π)
        else let r_p, c, op such that r_p --c,op--> r_s
          let P_π = preds(wlp(π)(safe))
          let op' = if P_π ⊈ r_s.preds then
                      r_s.preds := r_s.preds ∪ P_π
                      op
                    else op ∘ focus(cπ)
          remove subtrees starting from r_s
          for all r_2 such that
            r_2.covered ∧ r_2.states ⊭ false ∧
            r_s.states older than r_2.states do
            let r_1, c, op such that r_1 --c,op--> r_2
            r_2.covered := false
            r_2.states := false
            U := U ∪ {(r_1, op, r_2)}
          r_s.states := false
          r_s.covered := false
          update edge r_p --c,op'--> r_s
          U := U ∪ {(r_p, op', r_s)}
    return "program is safe"
  end
```

Figure 10. Lazy nested abstraction refinement algorithm

The algorithm maintains a work set of unprocessed ART edges U. In each iteration one unprocessed ART edge (r_1, op, r_2) is selected. Then the image of the abstract states in r_1 and the abstract transformer op is computed and the resulting abstract states are stored in $r_2.states$. If the computed abstract states are already subsumed by other ART nodes then the node r_2 is marked as covered. Otherwise if $r_2.states$ contains no error states then the ART is extended with new nodes that are the successors of r_2 for all the commands in P. The edges to the successor nodes are labeled by the commands c and the initial abstract transformer given by the Cartesian abstract post for the command c. Then the new edges are inserted to the set of unprocessed edges.

If r_2 contains error states then the trace labelling the path from r_0 to r_2 is a potential error trace. The analysis now determines whether this trace is a spurious error trace. For this purpose, it performs a symbolic backward analysis of the error trace. This backward analysis finds the oldest ancestor node r_s of r_2 with $r_s \xrightarrow{\pi}{}^* r_2$ such that $r_s.states$ represents some concrete state that can reach an error state by executing the trace π, i.e., formally r_s is the oldest node on the path that still satisfies

$$r_s.states \not\models \mathsf{wlp}(\pi)(\mathsf{safe}) \ .$$

If r_s is the root node of the ART then π is a concrete error trace and the procedure returns the counterexample. If, however, r_s is not the root node then the trace is a spurious error trace. In this case we call r_s the *spurious node* of the trace. The algorithm then determines the immediate predecessor node r_p of the spurious node. We call r_p the *pivot node* of the spurious error trace. The pivot node is the youngest node on the given path in the ART that does not represent any concrete states that can reach an error state by following the commands in the trace. Depending on the refinement phase either the abstract domain of r_s is refined or the abstract transformer that labels the edge between r_s and r_p is adapted.

The refinement works as follows. The spurious part of the error trace starts from the spurious node r_s. Our abstraction refinement procedure first attempts to refine the abstract domain of node r_s by adding new predicates $\mathcal{P}_\pi$ that are extracted from the spurious part π of the spurious error trace. The predicate extraction function preds guarantees that the weakest precondition $\mathsf{wlp}(\pi)(\mathsf{safe})$ of the path is expressible in the abstract domain of the refined node r_s, i.e., formally the following entailment holds

$$\alpha[\mathcal{P}_\pi](\mathsf{wlp}(\pi)(\mathsf{false})) \models \mathsf{wlp}(\pi)(\mathsf{false}) \ .$$

Suppose the analysis was to compute the most precise abstract post operator for each command. Then we were guaranteed that after refining the predicate set of node r_s and reprocessing the ART edge between r_p and r_s, the node r_s would no longer be spurious for this spurious error trace. This would ensure that the spurious error trace would eventually be eliminated. However, our analysis uses the Cartesian abstract post operator. Thus, the same spurious error trace might be reproduced after the refinement of the abstract domain. The refinement procedure would then fail to derive new predicates for node r_s. In this case, the second refinement phase refines the current abstract transformer op for command c that labels the edge between r_p and r_s. The abstract transformer op is refined by composition with the counterexample-guided focus for the spurious part $c\pi$ of the trace.

After the abstraction has been refined, the spurious subtrees below r_s are removed from the ART. In order to ensure soundness, ART nodes that have potentially been marked as covered due to subsumption by nodes in the removed subtrees are uncovered and reinserted into the work set. Finally, the spurious ART edge between r_p and r_s is updated and also reinserted into the work set.

If the set of unprocessed ART edges becomes empty then all outgoing edges of inner ART nodes have been processed and all leaf nodes are covered, i.e., an inductive invariant that proves the absence of reachable error states has been computed. Thus, the procedure returns "program is safe".

THEOREM 4 (Soundness). *Procedure* LazyNestedRefine *is sound, i.e., for any program P if* LazyNestedRefine(P) *terminates with "program is safe" then program P is safe.*

The proof of Theorem 4 relies on Proposition 2 and follows the argumentation in [25]. The next theorem states that procedure LazyNestedRefine has the progress property. In the setting of lazy abstraction, progress means that procedure LazyNestedRefine cannot diverge because it gets stuck on refining a finite set of spurious error traces over and over again.

THEOREM 5 (Progress). *A run Δ of procedure* LazyNestedRefine *terminates, unless the set of spurious error traces encountered in Δ is infinite.*

benchmark	checked properties	DP	time (in s)	CGF
List.traverse	AC, SF	MONA	0.11	no
List.init	AC, SF, PC	MONA	0.69	no
List.create	AC, SF	MONA	0.78	yes
List.getLast	AC, SF, PC	MONA	0.53	no
List.contains	AC, SF, PC	MONA	0.53	no
List.insertBefore	AC, SF	MONA	2.48	yes
List.append	AC, SF	MONA	8.95	no
List.filter	AC, SF	MONA	5.31	yes
List.partition	AC, SF	MONA	149.16	yes
List.reverse	AC, SF	MONA	5.52	yes
DList.addLast	AC, SF, DL, PC	MONA	2.05	yes
DList.erase	AC, SF, DL, PC	MONA	17.98	yes
SortedList.add	AC, SF, SO, PC	MONA, Z3	9.88	no
SkipList.add	AC, SF, PC	MONA	10.82	yes
Tree.add	AC, SF, PC	MONA	18.51	no
ParentTree.add	AC, SF, PL, PC	MONA	20.48	no
ThreadedTree.add	AC, SF, TH, SO, PC	MONA, Z3	445.93	no
Client.move	CS	Z3	3.11	no
Client.createMove	CS, PC	Z3	41.07	yes
Client.partition	CS, FC, PC	Z3	108.15	no

Properties: CS = call safety, AC = acyclic, SF = sharing free, DL = doubly linked, PL = parent linked, TH = threaded, SO = sorted, FC = frame condition, PC = post condition

Table 1. Summary of experiments. Column DP lists the used decision procedures. Column CGF indicates whether counterexample-guided focus was required to successfully verify the corresponding program.

benchmark	#appl. of abs. post	#ref. steps	final ART size	final ART depth	final ART st./loc.	#predicates total	#predicates avrg.	#predicates max.
List.traverse	3	0	4	4	1.00	4	2.8	3
List.init	5	1	5	5	2.00	7	5.4	7
List.create	11	6	6	6	1.67	11	6.7	9
List.getLast	7	1	6	6	2.00	7	6.0	7
List.contains	5	1	5	5	2.00	6	5.2	6
List.insertBefore	8	2	5	5	13.50	10	7.4	8
List.append	5	1	4	4	1.50	13	8.2	11
List.filter	31	5	14	5	2.50	12	7.1	10
List.partition	62	21	40	7	3.50	15	10.8	12
List.reverse	9	3	5	5	2.00	11	7.0	9
DList.addLast	7	3	5	5	1.50	8	7.2	8
DList.erase	23	6	10	5	5.00	10	7.6	8
SortedList.add	21	3	13	5	1.33	9	6.2	9
Skiplist.add	19	4	16	6	3.67	12	9.6	11
Tree.add	11	0	12	5	3.00	11	10.5	11
ParentTree.add	11	0	12	5	3.00	11	10.5	11
ThreadedTree.add	151	4	82	6	4.33	17	7.8	17
Client.move	8	0	9	9	1.00	16	8.4	11
Client.createMove	46	6	21	18	1.00	33	10.1	14
Client.partition	118	18	24	19	1.00	32	11.9	15

Table 2. Analysis details for experiments. The columns list the number of applications of the abstract post, the number of refinement steps, the size and depth of the final ART that represents the computed fixed point, the average number of abstract states per location in the fixed point, the total number of predicates, and the average and maximal number of predicates in a single ART node.

7. Implementation and Case Studies

We implemented our analysis in our tool Bohne. Bohne is implemented in Objective Caml and distributed as part of the Jahob system [30, 51, 56, 57]. The input to Bohne are Java programs annotated with special comments that specify procedure contracts and representation invariants of data structures. We represent elements of the abstract domain as sets of BDDs [11] and use the weaker order of propositional implication for the fixed point test in the abstraction refinement loop. Abstract transformers are also represented as BDDs to enable efficient post-image computation. We represent the program counter explicitely and not implicitely in abstract states. Our tool uses a few simple heuristics to guess an initial set of predicates from the input program and its specification. All additional predicates are inferred in the nested abstraction refinement procedure. Since we syntactically extract new predicates from weakest preconditions of finite traces in the program, we cannot infer reachability predicates (i.e., predicates with transitive closure operators) if they do not already occur in the specification. We therefore use a widening technique to infer new reachability predicates from the predicates that are extracted from weakest preconditions.

Case studies. We applied Bohne to verify operations on a diverse set of data structures implementations, checking a variety of properties. No manual adaptation of the abstract domain or the abstract transformers was required for the successful verification of these example programs. In particular, we were able to verify preservation of data structure invariants for operations on threaded binary trees (including sortedness and the in-order traversal invariant). We are not aware of any other analysis that can verify these properties with a comparable degree of automation. Further experiments cover data structures such as (sorted) singly-linked lists, doubly-linked lists, two-level skip lists, trees, and trees with parent pointers. The verified properties include: absence of runtime errors such as null dereferences complex data structure consistency properties, such as preservation of the tree structure and sortedness. Finally, we verified procedure contracts expressing functional correctness properties, e.g., how the set of elements stored in a data structure is affected by the procedure.

We further performed modular verification of data structure clients that use the interface for sets with iterators from the `java.util` library. For this purpose, we annotated procedure contracts for all set operations and then used our tool to infer invariants for the client. These invariants ensure that all preconditions of the set operations are satisfied at call sites in the client. Furthermore we verified functional correctness properties of the client code.

Table 1 shows a summary for a collection of benchmarks running on a 2.66 GHz Intel Core2 with 3 GB memory using one core. Each program satisfied all of the checked properties listed for the respective program and for each program all of the checked properties have been verified in a single run of the analysis. For reasoning about transitive closure of fields in tree-like data structures we used MONA [28] in combination with our field constraint analysis technique [52] for reasoning about non-tree fields. We further used the SMT solver Z3 [18] for verifying sortedness properties. For the the data structure clients we used only Z3. The last column in Table 1 indicates that for many of our benchmarks the verification would not succeed without the use of counterexample-guided focus, i.e., without counterexample-guided focus the analysis would not be able to rule out some spurious error trace and get stuck.

Note that our examples are not limited to stand-alone programs that build and then traverse their own data structures. Instead, our examples verify procedures with non-trivial preconditions, postconditions and representation invariants that can be part of arbitrarily large code.

Further details of the benchmarks are given in Tables 2 and 3. Table 3 gives details on the calls to the validity checker and its underlying decision procedures. One immediately observes that the calls to the validity checker are the main bottleneck of the analysis. On average, 98% of the total running time is spent in the validity checker. The reasons for the high running times are diverse. First, communication with decision procedures is currently implemented via files which is slower than passing data directly. Second, we use expensive decision procedures such as MONA. In

benchmark	#VC calls			rel. time spent in VC			time/DP call	
	total	DP	cache	total	abstr.	refine.	avrg.	max.
List.traverse	41	20	51.22%	92.59%	92.59%	0.00%	0.005	0.012
List.init	132	69	47.73%	95.93%	72.67%	23.26%	0.010	0.024
List.create	189	68	64.02%	95.36%	57.22%	38.14%	0.011	0.016
List.getLast	158	56	64.56%	97.74%	54.89%	42.86%	0.009	0.028
List.contains	114	52	54.39%	95.45%	55.30%	40.15%	0.010	0.028
List.insertBefore	246	143	41.87%	97.25%	80.61%	16.64%	0.017	0.052
List.append	311	254	18.33%	99.46%	97.10%	2.37%	0.035	0.080
List.filter	820	273	66.71%	97.36%	87.20%	10.17%	0.019	0.060
List.partition	7650	3027	60.43%	99.17%	95.63%	3.54%	0.049	0.088
List.reverse	615	312	49.27%	98.55%	89.05%	9.50%	0.017	0.048
DList.addLast	161	89	44.72%	97.86%	62.57%	35.28%	0.023	0.040
DList.erase	749	484	35.38%	98.15%	81.42%	16.73%	0.036	0.224
SortedList.add	470	190	59.57%	97.89%	65.65%	32.24%	0.051	0.120
Skiplist.add	679	241	64.51%	97.52%	43.84%	53.68%	0.044	0.076
Tree.add	390	124	68.21%	99.52%	67.59%	31.94%	0.149	0.624
ParentTree.add	428	141	67.06%	99.36%	63.41%	35.94%	0.144	0.596
ThreadedTree.add	2882	619	78.52%	99.65%	91.80%	7.85%	0.720	3.816
Client.move	111	82	26.13%	97.17%	85.48%	11.70%	0.037	0.136
Client.createMove	662	393	40.63%	96.35%	33.35%	63.00%	0.101	5.428
Client.partition	2138	896	58.09%	94.92%	27.13%	67.79%	0.115	5.540

Table 3. Statistics for validity checker (VC) calls. The columns list the total number of calls to the VC, the number of actual calls to decision procedures and the corresponding cache hit ration, the time spent in the VC relative to the total running time, and the average and maximal time spent for a single VC call.

some of the examples individual calls to these decision procedures can take up to several seconds. Running times can be improved by incorporating more efficient decision procedures for reasoning about specific data structures [7, 33, 54].

Limitations. The set of data structures that our implementation can handle is limited by the decision procedures that we have currently incorporated into our system. The use of monadic second-order logic over trees as our main logic for reasoning about transitive closure makes it more difficult to use our tool for verifying data structures that admit cycles or sharing. Furthermore, our widening technique for inferring new reachability predicates only works for flat tree-like structures. It is not appropriate for handling nested data structures such as lists of lists which may require the analysis to infer nested reachability predicates.

Costs and gains of automation. In order to estimate the costs and gains of an increased degree of automation, we compared Bohne to TVLA [35], the implementation of three-valued shape analysis [48]. We used TVLA version 3.0 alpha [8] for our comparison. We ran both tools on a set of singly-linked list benchmarks. For each example program we used the same precondition in both tools: heaps that form a forest of acyclic, sharing free lists. For TVLA we provided preconditions in the form of sets of three-valued logical structures. Bohne automatically computed the abstraction of preconditions given as logical formulas. We did not use finite differencing [46] to automatically compute predicate updates in TVLA. With finite differencing TVLA was unable to prove preservation of acyclicity of lists in some of the examples. We therefore used the standard abstract domain and abstract transformers for singly-linked lists that are shipped with TVLA (with standard focus as described in [48]). This abstract domain provides high precision for analyzing list-manipulating programs. We checked for properties that require such high precision, in order to get a meaningful comparison. We checked for absence of null dereferences as well as preservation of acyclicity and absence of sharing. All properties where checked in a single run of each analysis. Both tools were able to verify these properties for all our benchmarks.

The results of our experiments are summarized in Table 4. The running times for Bohne are between one and two orders of magnitude higher than for TVLA. Observe that almost all time is spent in the decision procedure. Thus, the increase in running time is the price that we pay for automation.

benchmark	running time (in s)			avrg. #abs. states		#predicates	
	Bohne	w/o VC	TVLA	Bohne	TVLA	Bohne	TVLA
traverse	0.11	0.008	0.179	1.0	8	4	12
create	0.78	0.036	0.133	1.7	6	11	12
getLast	0.53	0.012	0.214	2.0	10	7	14
insertBefore	2.48	0.068	0.503	13.5	15	10	18
append	8.95	0.048	0.462	1.5	23	13	18
filter	5.31	0.140	0.600	2.5	19	12	18
partition	149.16	1.238	1.508	3.5	72	15	18
reverse	5.52	0.080	0.331	2.0	12	11	14

Table 4. Comparison between Bohne and TVLA. The columns list total running times, average number of abstract states per location in the fixed point, and total number of predicates (we refer to the total number of unary predicates used by TVLA.). The third column shows the running time of Bohne without the time spent in the validity checker, i.e., this would be the total running time if we had an oracle for checking validity of formulae that would always return instantaneously.

More important in the context of this work is the fact that the space consumption of Bohne (measured in number of explored abstract states) is smaller than TVLA's, in some examples significantly. TVLA's focus operator eagerly splits abstract states (and summary nodes) during fixed point computation in order to retain high precision. This potentially leads to an explosion in the number of explored abstract states. Instead, our counterexample-guided focus splits abstract states and abstract objects on demand, only if the additional precision is required to rule out some spurious error trace. We believe that this is the main reason for the smaller space consumption of Bohne. This believe is supported by our experience with a uniform focus operator similar to TVLA's that we used in an earlier implementation.

However, there are other factors that play a role, such as the fact that Bohne's abstraction refinement loop infers a smaller number of predicates, compared to the fixed set of predicates that TVLA uses. A smaller predicate set results in a smaller abstract domain. Furthermore, the abstract domains of the two analyses are very similar, but not equivalent. In particular, abstract objects in our abstract states can be empty. This is in contrast to summary nodes in TVLA's three-valued structures which are bound to be non-empty. The presence of empty abstract objects can result in more compact abstractions. Hence, a more sturdy conclusion as to why the space consumption of Bohne is smaller requires further experiments.

8. Further Related Work

Shape analysis. Most shape analyses infer quantified invariants of heap programs, either explicitly or implicitly. We discuss some of these techniques in the following.

Our analysis shares many ideas with three-valued shape analysis [48] which inspired our idea of the counterexample-guided focus operator. In the previous section we presented an experimental comparison of both analyses. We now discuss additional related work. Recent approaches enable the automatic computation of transfer functions [36, 46] for three-valued shape analysis. Some of these approaches are using decision procedures [55]. A method for automated generation of predicates using inductive learning has been presented in [37], but is not based on counterexamples. A recent direction is the development of parameterized focus operators that are fined-tuned to specific verification tasks [40, 41, 47]. Our counterexample-guided focus is not only fine-tuned to a specific verification task but also to the individual steps of the analysis

of a specific verification task. Furthermore, this fine-tuning is performed automatically. However, the above techniques are explored in the more challenging setting of the verification of concurrent heap programs. Here, user-provided domain-specific knowledge is often invaluable for obtaining an efficient analysis.

Shape analyses based on separation logic such as [13, 19, 38] are typically tailored towards specific data structures and properties. This makes them scale to programs of impressive size [53], but also limits their application domain. Recent techniques introduce some degree of parametricity and allow the analysis to automatically adapt to specific data structure classes [4, 24]. All of these techniques still require manual adaptation of abstract transformers to perform materialization for different classes of data structures. Our focus operator performs this adaptation automatically.

Shape analysis based on abstract regular tree model checking [10] encodes heap programs into tree transducers which can be analyzed using automata-based program analysis techniques. The encoding into tree transducers loses precision if the structures observed in the heap program do not exhibit some regularity. This shape analysis can take advantage of abstraction refinement techniques that have been developed for abstract regular tree model checking [9]. In particular, there is an automata-based version of predicate abstraction that can be combined with abstraction refinement and provide progress guarantees. However, these refinement techniques cannot prevent any loss of precision that is caused by the initial encoding of a heap program into tree transducers. Also, this approach focuses on shape invariants of data structures and does not apply to properties such as sortedness.

Predicate abstraction. Classical predicate abstraction [22] can be seen as an instance of Boolean heap abstraction where all abstraction predicates are closed formulas. Our technique of counterexample-guided focus therefore carries over to classical predicate abstraction.

The advantages of combining predicate abstraction with shape analysis are clearly demonstrated in lazy shape analysis [6]. Lazy shape analysis performs independent runs of a shape analysis algorithm, whose results are then used to improve the precision of predicate abstraction. The combined analysis implicitly infers quantified invariants. In contrast, our analysis transcends the lazy abstraction technique to the point where it itself becomes effective as a shape analysis. Thus, our analysis offers the benefits of lazy abstraction (i.e., a high degree of automation and targeted precision) also for the heap-aware analysis.

Indexed predicate abstraction [31] uses predicates with free variables to infer quantified invariants and is similar to our analysis. However, indexed predicate abstraction has not yet been used for the analysis of heap programs. Heuristics for automatic discovery of indexed predicates are described in [32]. The abstract domain of our analysis is more general than the indexed predicate abstraction domain, because it contains disjunctions of universally quantified statements. The presence of disjunctions avoids loss of precision at join points of the control flow graph. This is important in the context of abstraction refinement because it enables the analysis to precisely identify spurious error traces in the abstract system.

The SLAM tool [3] uses Cartesian abstraction [2] on top of predicate abstraction. The loss of precision under Cartesian abstraction is not a decisive factor in standard predicate abstraction [1]. However, there are other approximations of abstract transformers that are used for performance reasons in actual implementations. A side-effect of such approximations is that spurious error traces may not be eliminated by sole refinement of the abstract domain. SLAM incorporates an algorithm developed by Das and Dill [17] as a countermeasure. This algorithm gradually refines the abstract transformer towards the most precise abstract post for a fixed predicate abstraction. In SLAM this algorithm is applied whenever no new predicates can be extracted from a spurious error trace. The refinement of the abstract transformer guarantees that the spurious error trace is eventually eliminated. However, this algorithm is not appropriate in the context of an abstract domain of quantified assertions. It relies on a greedy elimination of spurious abstract transitions in the abstract transformer. In predicate abstraction these abstract transitions are simple conjunctions of predicates. In our setting they are quantified Boolean combinations of predicates. The enumeration of abstract transitions is therefore infeasible.

Quantified invariants over arrays. For programs over arrays the problem of how to treat disjunctions in the inference of quantified invariants is not as accentuated as in the context of heap programs. Techniques for inferring quantified invariants that only apply to programs over arrays include [20, 27, 29, 49]. Noticeable exceptions are [23, 50] which also apply to heap programs. Here the user specifies predicates and templates for the quantified invariants that partially fix the Boolean structure of the inferred invariants. The analysis then automatically instantiates the template parameters. The templates can significantly reduce the search-space. However, finding the right predicates and the right templates is non-trivial in the context of heap programs.

Extracting predicates from counterexamples. The problem of extracting good predicates from counterexamples for a domain of quantified assertions is orthogonal to the contribution of this paper. In the setting of quantified invariant inference, the question of how to infer predicates from a finite set of spurious counterexamples that rule out infinitely many similar spurious counterexamples (e.g., resulting from the traversal of a recursive data structure) is open. In our implementation we followed a practical approach to solve this problem. Techniques for the inference of quantified interpolants [43] offer a promising alternative. Such techniques could be integrated in our approach following the line of interpolation-based abstraction refinement [5, 27].

9. Conclusion

In this paper, we have addressed the automated inference of quantified invariants for software verification. The fine-tuning of the focus operator (from the related area of shape analysis) is central to finding the right efficiency-precision tradeoff in the underlying program analysis. We have put forward the idea to use counterexamples to guide the fine-tuning of the focus operator. We have shown how this idea can be realized in a method and tool; preliminary experiments indicate its practical potential.

An interesting line of future research is the extension of the presented work to the verification of concurrent programs. It seems that here one should seek the integration of counterexample-guided focus with existing mechanisms for manually fine-tuning parameterized versions of the focus operator[39]. This would allow the user to incorporate domain-specific knowledge (e.g., about synchronization) into the analysis.

References

[1] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *TACAS'04*, pages 388–403, 2004.

[2] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *TACAS'01*, pages 268–283, 2001.

[3] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL'02*, pages 1–3, 2002.

[4] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV'07*, pages 178–192, 2007.

[5] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007.

[6] D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *CAV'06*, pages 532–546, 2006.

[7] J. D. Bingham and Z. Rakamaric. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *VMCAI'06*, pages 207–221, 2006.

[8] I. Bogudlov, T. Lev-Ami, T. W. Reps, and M. Sagiv. Revamping TVLA: Making Parametric Shape Analysis Competitive. In *CAV'07*, pages 221–225, 2007.

[9] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. *ENTCS*, 149(1):37–48, 2006.

[10] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *SAS'06*, pages 52–70, 2006.

[11] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *TC*, 35(10):677–691, 1986.

[12] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE'03*, pages 385–395, 2003.

[13] B.-Y. E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In *SAS'07*, pages 384–401, 2007.

[14] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV'00*, pages 154–169, 2000.

[15] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252, 1977.

[16] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *FPCA'95*, pages 170–181, 1995.

[17] S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *LICS'01*, pages 51–60, 2001.

[18] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, 2008.

[19] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS'06*, pages 287–302, 2006.

[20] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL'02*, pages 191–202, 2002.

[21] D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *POPL'05*, pages 338–350, 2005.

[22] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV'97*, pages 72–83, 1997.

[23] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL'08*, pages 235–246, 2008.

[24] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *PLDI'07*, pages 256–265, 2007.

[25] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL'02*, pages 58–70, 2002.

[26] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *SPIN 03: Model Checking of Software*, pages 235–239. 2003.

[27] R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV'07*, pages 193–206, 2007.

[28] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, University of Aarhus, January 2001.

[29] L. Kovacs and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE'09*, pages 470–485, 2009.

[30] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[31] S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *VMCAI'04*, pages 267–281, 2004.

[32] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV'04*, 2004.

[33] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *POPL'08*, pages 171–182, 2008.

[34] P. Lam, V. Kuncak, and M. Rinard. Hob: A tool for verifying data structure consistency. In *CC'05*, 2005.

[35] T. Lev-Ami. TVLA: A Framework for Kleene Based Logic Static Analyses. Master's thesis, Tel-Aviv University, Israel, 2000.

[36] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *CAV'06*, pages 533–546, 2006.

[37] A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement via inductive learning. In *CAV'05*, 2005.

[38] S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS'07*, pages 419–436, 2007.

[39] R. Manevich. *Partially Disjunctive Shape Analysis*. PhD thesis, Tel-Aviv University, Israel, 2009.

[40] R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In *TACAS'07*, pages 3–18, 2007.

[41] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *SAS'04*, pages 265–279, 2004.

[42] M. Marron, D. Kapur, D. Stefanovic, and M. V. Hermenegildo. A static heap analysis for shape and connectivity. In *LCPC*, pages 345–363, 2006.

[43] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS'08*, volume 4963, pages 413–427, 2008.

[44] A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL'07*, pages 245–259, 2007.

[45] A. Podelski and T. Wies. Boolean Heaps. In *SAS'05*, pages 267–282, 2005.

[46] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *ESOP'03*, pages 380–398, 2003.

[47] N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. In *ESOP'07*, pages 220–236, 2007.

[48] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.

[49] M. N. Seghir, A. Podelski, and T. Wies. Abstraction refinement for quantified array assertions. In *SAS'09*, 2009.

[50] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI'09*, 2009.

[51] T. Wies. *Symbolic Shape Analysis*. PhD thesis, University of Freiburg, Freiburg, Germany, 2009.

[52] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field Constraint Analysis. In *VMCAI'06*, 2006.

[53] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV'08*, pages 385–398, 2008.

[54] G. Yorsh, A. M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A Logic of Reachable Patterns in Linked Data-Structures. In *FOSSACS'06*, pages 94–110, 2006.

[55] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS'04*, 2004.

[56] K. Zee, V. Kuncak, and M. Rinard. Full Functional Verification for Linked Data Structures. In *PLDI'08*, pages 349–361, 2008.

[57] K. Zee, V. Kuncak, and M. C. Rinard. An integrated proof language for imperative programs. In *PLDI'09*, pages 338–351, 2009.