Angelic Verification: Precise Verification Modulo Unknowns

Ankush Das[1], Shuvendu K. Lahiri[1], Akash Lal[1], and Yi Li[2]

[1] Microsoft Research, {t-ankdas, shuvendu, akashl}@microsoft.com
[2] University of Toronto, liyi@cs.toronto.edu

Abstract. Verification of open programs can be challenging in the presence of an unconstrained environment. Verifying properties that depend on the environment yields a large class of uninteresting false alarms. Using a verifier on a program thus requires extensive initial investment in modeling the environment of the program. We propose a technique called *angelic verification* for verification of open programs, where we constrain a verifier to report warnings only when no acceptable environment specification exists to prove the assertion. Our framework is parametric in a vocabulary and a set of angelic assertions that allows a user to configure the tool. We describe a few instantiations of the framework and an evaluation on a set of real-world benchmarks to show that our technique is competitive with industrial-strength tools even without models of the environment.

1 Introduction

Scalable software verifiers offer the potential to find defects early in the development cycle. The user of such a tool can specify a property (e.g. correct usage of kernel/security APIs) using some specification language and the tool validates that the property holds on all feasible executions of the program. There has been a significant progress in the area of software verification, leveraging ideas from model checking [13], theorem proving [34] and invariant inference algorithms [16,22,33]. Tools based on these principles (e.g. SDV [3], F-Soft [24]) have found numerous bugs in production software.

However, a fundamental problem still limits the adoption of powerful software verifiers in the hands of end users. Most (interprocedural) program verifiers aim to verify that a program does not fail assertions under all possible feasible executions of the program. This is a good match when the input program is "closed", i.e., its execution starts from a well-defined initial state, and external library methods are included or accurately modeled. Scalability concerns preclude performing monolithic verification that includes all transitive callers and library source code. In practice, a significant portion of verification tool development requires closing a program by (i) either providing a *harness* (a client program) or a *module invariant* [30] to constrain the inputs and (ii) *stubs* for external library procedures [3]. The effect of modeling is to *constrain* the set of *unknowns* in the program to rule out infeasible executions. Absence of such modeling results in numerous uninteresting alarms and deters a user from further interacting with the tool. "*A stupid false positive implies the tool is stupid*" [6]. The significant initial modeling overhead often undermines the value provided by verifiers. Even "bounded" versions

```
// inconsistency
procedure Bar(x:int) {
    if (x != NULL) { gs := 1; }
    else { gs := 2; }
    // possible BUG or dead code
    assert x != NULL;
    m[x] := 5;
}
// internal bug
procedure Baz(y:int) {
    assert y != NULL; //DEFINITE BUG
    m[y] := 4;
}
// entry point
procedure Foo(z:int) {
    call Bar(z);       // block + relax
    call Baz(NULL); // internal bug
    call FooBar();     // external calls
}
```

```
// globals
var gs:int, m:[int]int;

// external call
procedure FooBar() {
    var x, w, z:int;
    call z := Lib1();
    assert z != NULL;
    m[z] := NULL;
    call x := Lib2();
    assert x != NULL;
    w := m[x];
    assert w != NULL;
    m[w] := 4;
}
// library
procedure Lib1() returns (r:int);
procedure Lib2() returns (r:int);
```

Fig. 1: Running example.

of verifiers (such as CBMC [14]) suffer from this problem because these unknowns are present even in bounded executions.

Example 1. Consider the example program (written in the Boogie language [4]) in Figure 1. The program has four procedures Foo, Bar, Baz, FooBar and two external *library* procedures Lib1, Lib2. The variables in the programs can be scalars (of type int) or *arrays* (e.g. m) that map int to int. The Boogie program is an encoding of a C program [15]: pointers and values are uniformly modeled as integers (e.g. parameter x of Bar, or the return value of Lib1), and memory dereference is modeled as array lookup (e.g. m[x]). The procedures have assertions marked using assert statements. The entry procedure for this program is Foo.

There are several sources of *unknowns* or unconstrained values in the program: the parameter z to Foo, the global variable m representing the heap, and the return values of library procedures Lib1 and Lib2. Even a precise verifier is bound to return assertion failures for *each* of the assertions in the program. This is due to the fact that all the assertions, except the one in Baz (the only *definite bug* in the program) are assertions over unknowns in the program and (sound) verifiers tend to be conservative (over-approximate) in the face of unknowns. Such *demonic* nature of verifiers will result in several false alarms.

Overview Our goal is to push back on the demonic nature of the verifier by prioritizing alarms with higher evidence. In addition to the warning in Baz, the assertion in Bar is suspicious as the only way to avoid the bug is to make the "else" branch unreachable in Bar. For the remaining assertions, relatively *simple* constraints on the unknown values suffice to explain the correctness of these assertions. For example, it is *reasonable* to assume that calls to library methods do not return NULL, their dereferences (m[x]) store non-null values and calls to two different library methods do not return aliased pointers. We tone down the demonic nature of verifiers by posing a more *angelic* decision problem for the verifier (also termed as *abductive inference* [20,10]):

For a given assertion, does there exists an *acceptable* specification over the unknowns such that the assertion holds?

This forces the verifier to work harder to exhaust the space of *acceptable* specifications before showing a warning for a given assertion. Of course, this makes the verification problem less defined as it is parameterized by what constitutes “acceptable” to the end user of the tool. At the same time, it allows a user to be able to configure the demonic nature of the tool by specifying a vocabulary of acceptable specifications.

In this paper, we provide a user a few dimensions to specify a vocabulary $Vocab$ that constitutes a specification (details can be found in Section 4). The vocabulary can indicate a template for the atomic formulas, or the Boolean and quantifier structure. Given a vocabulary $Vocab$, we characterize an acceptable specification by how (a) *concise* and (b) *permissive* the specification is. Conciseness is important for the resulting specifications to be understandable by the user. Permissiveness ensures that the specification is not overly strong, thus masking out true bugs. The failure in Bar is an example, where a specification $\mathsf{x} \neq \mathsf{NULL}$ is not permissive as it gives rise to dead code in the “else” branch before the assertion. To specify desired permissiveness, we allow the users to augment the program with a set of *angelic assertions* $\hat{\mathcal{A}}$. The assertions in $\hat{\mathcal{A}}$ should *not* be provable in the presence of any inferred specification over the unknowns. An angelic assertion assert $e \in \hat{\mathcal{A}}$ at a program location l indicates that the user expects at least one state to reach l and satisfy $\neg e$. For Bar one can add two assertions assert false inside each of the branches. The precondition $\mathsf{x} \neq \mathsf{NULL}$ would be able to prove that assert false in the “else” branch is unreachable (and thus provable), which prevents it from being permissive. We describe a few such useful instances of angelic assertions in Section 3.1.

We have implemented the angelic verification framework in a tool called *AngelicVerifier* for Boogie programs. Given a Boogie program with a set S of entrypoints, *AngelicVerifier* invokes each of the procedures in S with unknown input states. In the absence of any user-provided information, we assume that S is the set of all procedures in the program. Further, the library procedures are assigned a body that assigns a non-deterministic value to the return variables and adds an assume statement with a predicate unknown_i (Figure 2). This predicate will be used to constrain the return values of a procedure for *all possible call sites* (Section 4) within an entrypoint.

AngelicVerifier invokes a given (demonic) verifier on this program with all entrypoints in S. If the verifier returns a trace that ends in an assertion failure, *AngelicVerifier* tries to infer an acceptable specification over the unknowns. If it succeeds, it installs the specification as a precondition of the entry point and iterates. If it is unable to infer an acceptable specification, the trace is reported as a defect to the user.

Figure 3 shows the output of $AngelicVerifier$ applied to our example:

– For a trace that starts at Bar and fails the assert on line 6, we conjecture a specification $\mathsf{x} \neq \mathsf{NULL}$ but discover that it is not permissive. The line with “ANGELIC_WARNING” is a warning shown to the user.
– For the trace that starts at Baz and fails the assert on line 11, we block the assertion failure by installing the constraint $\mathsf{y} \neq \mathsf{NULL}$. The code of Bar does not have any indication that it expects to see NULL as input.

```
function unknown_0(a: int ): bool;
function unknown_1(a: int ): bool;

procedure Lib1() returns (r: int) {
    assume unknown_0(r);
    return;
}

procedure Lib2() returns (r: int) {
    assume unknown_1(r);
    return;
}
```

Fig. 2: Modeling of external procedures by *AngelicVerifier*. All variables are non-deterministically initialized.

```
// Trace: Bar → assert on line 6
SPEC :: x ≠ NULL, Spec not permissive
ANGELIC_WARNING: Assertion x != NULL fails in proc Bar
// Trace: Baz → assert on line 11
SPEC :: y ≠ NULL
// Trace: FooBar → assert on line 27
SPEC :: (∀ x_1: unknown_0(x_1) ⇒ x_1 ≠ NULL)
// Trace: FooBar → assert on line 30
SPEC :: (∀ x_2: unknown_1(x_2) ⇒ x_2 ≠ NULL)
// Trace: FooBar → assert on line 32
SPEC :: (∀ x_2, x_1: unknown_1(x_2) ∧
    unknown_0(x_1)⇒ (x_2 ≠ x_1 ∧ m[x_2] ≠ NULL))
// Trace: Foo → Baz → assert on line 11
ANGELIC_WARNING: Assertion y != NULL fails in proc Baz
```

Fig. 3: Output of *AngelicVerifier* on the program shown in Figure 1. A line with "SPEC" denotes an inferred specification to suppress a trace.

– For the three traces that start at FooBar and fail an assertion inside it, we block them using constraints on the return values of library calls. Notice that the return values are not in scope at the entry to FooBar; they get constrained indirectly using the unknown_i predicates. The most interesting block is for the final assertion which involves assuming that (a) the returns from the two library calls are never aliased, and (b) the value of the array m at the value returned by Lib2 is non-null. (See Section 4)
– The trace starting at Foo that calls Baz and fails on line 11 cannot be blocked (other than by using the non-permissive specification *false*), and is reported to the user.

Contributions In summary, the paper makes the following contributions: (a) We provide a framework for performing angelic verification with the goal of highlighting highest confidence bugs. (b) We provide a parametric framework based on $Vocab$ and $\hat{\mathcal{A}}$ to control the level of angelism in the tool that a user can configure. (c) We describe a scalable algorithm for searching specifications using *ExplainError* (Section 4). We show an effective way to deal with internal non-determinism resulting from calls to library procedures. (d) We have implemented the ideas in a prototype tool $AngelicVerifier$ and evaluated it on real-world benchmarks. We show that *AngelicVerifier* is competitive with industrial-strength tools even without access to the environment models.

2 Programming language

Syntax. We formalize the ideas in the paper in the context of a simple subset of the Boogie programming language [4]. A program consists of a set of *basic blocks* $Block$; each block consists of a *label* $BlockId$, a body $s \in Stmt$ and a (possibly empty) set of *successor* blocks. A program has a designated first block $Start \in Block$. Most statements are standard; the havoc x statement assigns a non-deterministic value to the variable x. An expression ($Expr$) can be a variable identifier or an application of function $f \in Functions$. A formula ($Formula$) includes Boolean constants, application of a predicate $p \in Predicates$, and closed under Boolean connectives and

$$\begin{array}{llll}
P & \in \mathit{Program} & ::= & \mathit{Block}^{+} \\
BL & \in \mathit{Block} & ::= & \mathit{BlockId} : s;\ \textsf{goto}\ \mathit{BlockId}^{*} \\
s, t & \in \mathit{Stmt} & ::= & \textsf{skip} \mid \textsf{assert}\ \phi \mid \textsf{assume}\ \phi \mid \mathtt{x} := e \mid \textsf{havoc}\ \mathtt{x} \mid s; s \\
\mathtt{x}, \mathtt{y} & \in \mathit{Vars} & & \\
e & \in \mathit{Expr} & ::= & \mathtt{x} \mid f(e, \ldots, e) \\
\phi, \psi & \in \mathit{Formula} & ::= & \textsf{true} \mid \textsf{false} \mid p(e, \ldots, e) \mid \phi \wedge \phi \mid \forall x :\ \phi \mid \neg\phi
\end{array}$$

Fig. 4: A simple programming language.

quantifiers. The constructs are expressive enough to model features of most programming languages such as C [15] or Java [1]. Conditional statements are modeled using assume and goto statements; heap is modeled using *interpreted* array functions $\{\mathit{read}, \mathit{write}\} \subseteq \mathit{Functions}$ [35].

Semantics. A *program state* σ is a type-consistent valuation of variables in scope in the program. The set of all states is denoted by $\Sigma \cup \{\mathit{Err}\}$, where Err is a special state to indicate an *assertion failure*. For a given state $\sigma \in \Sigma$ and an expression (or formula) e, e_σ denotes the evaluation of e in the state. For a formula $\phi \in \mathit{Formula}$, $\sigma \models \phi$ holds if ϕ_σ evaluates to true. The semantics of a program is a set of *execution traces*, where a trace corresponds to a sequence of program states. We refer the readers to earlier works for details of the semantics [4]. Intuitively, an execution trace for a block BL corresponds to the sequence of states obtained by executing the body, and extending the *terminating* sequences with the traces of the successor blocks (if any). A sequence of states for a block does not terminate if it either executes an assume ϕ or an assert ϕ statement in a state $\sigma \in \Sigma$ such that $\sigma \not\models \phi$. In the latter case, the successor state is Err. The traces of a program is the set of traces for the start block Start. Let $\mathcal{T}(P)$ be the set of all traces of a program P. A program P is *correct* (denoted as $\models P$) if $\mathcal{T}(P)$ does not contain a trace that ends in the state Err. For a program P that is not correct, we define a *failure trace* as a trace τ that starts at Start and ends in the state Err.

3 Angelic verification

In this section, we make the problem of *angelic verification* more concrete. We are given a program P that cannot be proved *correct* in the presence of unknowns from the environment (e.g. parameters, globals and outputs of library procedures). If one takes a conservative approach, we can only conclude that the program P has a *possible* assertion failure. In this setting, verification failures offer no information to a user of the tool. Instead, one can take a more pragmatic approach. If the user can characterize a class of *acceptable* missing specifications Φ that precludes verification (based on experience), one can instead ask a weaker verification question: *does there exist a specification $\phi \in \Phi$ such $\phi \models P$?*. One can characterize the acceptability of a specification ϕ along two axes: (i) *Conciseness* — the specification should have a concise representation in some vocabulary that the user expects and can inspect. This usually precludes specifications with several levels of Boolean connectives, quantifiers, or complex atomic expressions. (ii) *Permissive* — the specification ϕ should not be too strong to preclude feasible states of P that are known to exist. We allow two mechanisms for an expert user to control the set of acceptable specifications:

- The user can provide a *vocabulary* Vocab of acceptable specifications, along with a checker that can test membership of a formula ϕ in Vocab. We show instances of Vocab in Section 4.
- The user can augment P with a set of *angelic* assertions $\hat{\mathcal{A}}$ at specific locations, with the expectation that any specification should not prove an assertion $\mathsf{assert}\ e \in \hat{\mathcal{A}}$.

We term the resulting verification problem *angelic* as the verifier co-operates with the user (as opposed to playing an adversary) to find specifications that can prove the program. This can be seen as a particular mechanism to allow an expert user to customize the *abductive inference* problem tailored to their needs [20]. If no such specification can found, it indicates that the verification failure of P cannot be categorized into previously known buckets of false alarms.

We make these ideas more precise in the next few sections. In Section 3.1, we describe the notion of *angelic correctness* given P, Vocab and $\hat{\mathcal{A}}$. In Section 3.2, we describe an algorithm to prove *angelic correctness* using existing program verifiers.

3.1 Problem formulation

Let $\phi \in \mathit{Formula}$ be a well-scoped formula at the block Start of a program P. We say that a program P is *correct under* ϕ (denoted as $\phi \models P$), if the augmented program $\mathit{Start}_0\ :\ \mathsf{assume}\ \phi\ ;\ \mathsf{goto}\ \mathit{Start}$ with "Start" block as Start_0 is correct. In other words, the program P is correct with a *precondition* ϕ.

Let $\mathcal{A}$ be the set of assertions in program P. Additionally, let the user specify an additional set $\hat{\mathcal{A}}$ of angelic assertions at various blocks in P. We denote the program P_{A_1,A_2} as the instrumented version of P that has two sets of assertions enabled:

- *Normal* assertions $A_1 \subseteq \mathcal{A}$ that constitute a (possibly empty) subset of the original assertions present in P, and
- *Angelic* assertions $A_2 \subseteq \hat{\mathcal{A}}$ that constitute a (possibly empty) subset of set of additional user supplied assertions.

Definition 1 (Permissive precondition) *For a program $P_{\mathcal{A},\hat{\mathcal{A}}}$ and formula ϕ, $\mathit{Permissive}(P_{\mathcal{A},\hat{\mathcal{A}}}, \phi)$ holds if for every assertion $s \in \hat{\mathcal{A}}$, if $\phi \models P_{\emptyset,\{s\}}$, then* $\mathsf{true} \models P_{\emptyset,\{s\}}$.

In other words, a specification ϕ is not allowed to prove any assertion $s \in \hat{\mathcal{A}}$ that was not provable under the unconstrained specification true.

Definition 2 (Angelic correctness) *Given (i) a program P with a set of normal assertions $\mathcal{A}$, (ii) an angelic set of assertions $\hat{\mathcal{A}}$, and (iii) a vocabulary Vocab constraining a set of formulas at Start, P is* angelically correct under $(\mathit{Vocab}, \hat{\mathcal{A}})$ *if there exists a formula $\phi \in \mathit{Vocab}$ such that: (i) $\phi \models P_{\mathcal{A},\emptyset}$, and (ii) $\mathit{Permissive}(P_{\emptyset,\hat{\mathcal{A}}}, \phi)$ holds.*

If no such specification ϕ exists, then we say that P has an *angelic* bug with respect to $(\mathit{Vocab}, \hat{\mathcal{A}})$. In this case, we try to ensure the angelic correctness of P with respect to a subset of the assertions in P; the rest of the assertions are flagged as angelic warnings.

Examples of angelic assertions $\hat{\mathcal{A}}$ If one provides $\mathsf{assert\ false}$ at $Start$ to be part of $\hat{\mathcal{A}}$, it disallows preconditions that are inconsistent with other preconditions of the program [20]. If we add $\mathsf{assert\ false}$ at the end of every basic block, it prevents us from creating preconditions that create dead code in the program. This has the effect of detecting *semantic inconsistency* or *doomed* bugs [21,19,23,36]. Further, we can allow checking such assertions interprocedurally and at only a subset of locations (e.g. exclude defensive checks in callees). Finally, one can encode other domain knowledge using such assertions. For example, consider checking the correct lock usage for $\mathsf{if}(*)\{\mathsf{L1}:\ \mathsf{assert}\ \neg\mathsf{locked}(\mathsf{l1});\ \mathsf{lock}(\mathsf{l1});\}\ \mathsf{else}\ \{\mathsf{L2}:\ \mathsf{assert}\ \mathsf{locked}(\mathsf{l2});\ \mathsf{unlock}(\mathsf{l2});\}$. If the user expects an execution where $\mathsf{l1} = \mathsf{l2}$ at $\mathsf{L2}$, the angelic assertion $\mathsf{assert}\ \mathsf{l1} \neq \mathsf{l2} \in \hat{\mathcal{A}}$ precludes the precondition $\neg\mathsf{locked}(\mathsf{l1}) \wedge \mathsf{locked}(\mathsf{l2})$, and reveals a warning for at least one of the two locations. As another example, if the user has observed a runtime value v for variable $\mathtt{x}$ at a program location l, she can add an assertion $\mathsf{assert}\ \mathtt{x} \neq v \in \hat{\mathcal{A}}$ at l to ensure that a specification does not preclude a known feasible behavior; further, the idea can be extended from feasible values to feasible intraprocedural path conditions.

3.2 Finding angelic bugs

Algorithm 1 describes a (semi) algorithm for proving angelic correctness of a program. In addition to the program, it takes as inputs the set of angelic assertions $\hat{\mathcal{A}}$, and a vocabulary $Vocab$. On termination, the procedure returns a specification E and a subset $\mathcal{A}_1 \subseteq \mathcal{A}$ for which the resultant program is angelically correct under E. Lines 1 and 2 initialize the variables E and $\mathcal{A}_1$, respectively. The loop from line 3 — 16 performs the main act of blocking failure traces in P. First, we verify the assertions $\mathcal{A}_1$ over P. The routine tries to establish $E \models P$ using a sound and complete program verifier; the program verifier itself may never terminate. We return in line 6 if verification succeeds and P contains no failure traces (`NO_TRACE`). In the event a failure trace τ is present, we query a procedure $ExplainError$ (see Section 4) to find a specification ϕ that can prove that none of the executions along τ fail an assertion. Line 10 checks if the addition of the new constraint ϕ still ensures that the resulting specification E_1 is permissive. If not, then it suppresses the assertion a that failed in τ (by removing it from $\mathcal{A}_1$) and outputs the trace τ to the user. Otherwise, it adds ϕ to the set of constraints collected so far. The loop repeats forever until verification succeeds in Line 4. The procedure may fail to terminate if either the call to $Verify$ does not terminate, or the loop in Line 3 does not terminate due to an unbounded number of failure traces.

Theorem 1 *On termination, Algorithm 1 returns a pair of precondition E and a subset $\mathcal{A}_1 \subseteq \mathcal{A}$ such that (i) $E \models P$ when only assertions in $\mathcal{A}_1$ are enabled, and (ii) $Permissive(P_{\mathcal{A},\hat{\mathcal{A}}}, E)$.*

The proof follows directly from the check in line 4 that establishes (i), and line 10 that ensures permissiveness.

```
Input: Program $P$ with assertions $\mathcal{A}$,
Input: Angelic Assertions $\hat{\mathcal{A}}$,
Input: Vocabulary $Vocab$,
Output: A permissive specification $E$,
Output: A set of assertions $\mathcal{A}_1 \subseteq \mathcal{A}$ for
    which $E \models P_{\mathcal{A}_1,\emptyset}$
 1: $E \leftarrow \emptyset$
 2: $\mathcal{A}_1 \leftarrow \mathcal{A}$
 3: loop
 4:     $\tau \leftarrow Verify(P_{\mathcal{A}_1,\emptyset}, E)$ /* $E \models P$ */
 5:     if $\tau =$ NO_TRACE then
 6:         return $(E, \mathcal{A}_1)$
 7:     end if
 8:     $\phi \leftarrow ExplainError(P, \tau, Vocab)$
 9:     $E_1 \leftarrow E \cup \{\phi\}$
10:     if $\neg Permissive(P_{\emptyset,\hat{\mathcal{A}}}, E_1)$ then
11:         Let $a$ be the failing assert in $\tau$
12:         $\mathcal{A}_1 \leftarrow \mathcal{A}_1 \setminus \{a\}$ /* Report $a$ */
13:     else
14:         $E \leftarrow E_1$
15:     end if
16: end loop
```

Algorithm 1: *AngelicVerifier*

```
Input: Program $P$, failure trace $\tau$, vocabulary
    $Vocab$
Output: A formula that blocks $\tau$
 1: $\tau_1 \leftarrow ControlSlice(P, \tau)$
 2: $\phi_1 \leftarrow wlp(\tau_1, \mathsf{true})$
 3: $\phi_2 \leftarrow EliminateMapUpdates(\phi_1)$
 4: $atoms_1 \leftarrow FilterAtoms\ (GetAtoms(\phi_2),$
    $Vocab.Atoms)$
 5: if $Vocab.Bool =$ MONOMIAL then
 6:     $S \leftarrow \{a \mid a \in atoms_1,\ and\ a \models \phi_2\}$
 7:     return $S = \emptyset\ ?\ \mathsf{false}\ :\ (\bigvee_{a \in S} a)$
 8: else
 9:     return $ProjectAtoms(\phi_2, atoms_1)$
10: end if
```

Algorithm 2: *ExplainError*

4 ExplainError

Problem Given a program P that is not correct, let τ be a failure trace of P. Since a trace can be represented as a valid program ($Program$) in our language (with a single block containing the sequence of statements ending in an assert statement), we will treat τ as a program with a single control flow path.

Informally, the goal of ExplainError is to return a precondition ϕ from a given vocabulary $Vocab$ such that $\phi \models \tau$, or false if no such precondition exists. ExplainError takes as input the following: (a) a program P, (b) a failure trace τ in P represented as a program and (c) a *vocabulary* $Vocab$ that specifies syntactic restrictions on formulas to search over. It returns a formula ϕ such that $\phi \models \tau$ and $\phi \in Vocab \cup \{\mathsf{false}\}$. It returns false either when (a) the vocabulary does not contain any formula ϕ for which $\phi \models \tau$, or (b) the search does not terminate (say due to a timeout).

Note that the *weakest liberal precondition* (wlp) of the trace [18] is guaranteed to be the weakest possible blocking constraint; however, it is usually very specific to the trace and may require enumerating all the concrete failing traces inside Algorithm 1. Moreover, the resulting formula for long traces are often not suitable for human consumption. When ExplainError returns a formula other than false, one may expect ϕ to be the *weakest* (most permissive) constraint in $Vocab$ that blocks the failure path. However, this is not possible for several reasons (a) efficiency concerns preclude searching for the weakest, (b) $Vocab$ may not be closed under disjunction and therefore the weakest constraint may not be defined. Thus the primary goals of ExplainError are to be (a) scalable (so that it can be invoked in the main loop in Algorithm 1), and (b) the resulting constraints are concise even if not the weakest over $Vocab$.

Algorithm Algorithm 2 provides the high-level flow of ExplainError. Currently, the algorithm is parameterized by $Vocab$ that consists of two components:

- $Vocab.Atoms$: a template for the set of atomic formulas that can appear in a blocking constraint. This can range over equalities ($e_1 = e_2$), difference constraints ($e_1 \leq e_2 + c$), or some other syntactic pattern.
- $Vocab.Bool$: the complexity of Boolean structure of the blocking constraint. One may choose to have a *clausal* formula ($\bigvee_i e_i$), *cube* formulas ($\bigwedge_i e_i$), or an arbitrary *conjunctive normal form* (CNF) ($\bigvee_j (\bigwedge_i e_i)$) over atomic formulas e_i.

Initially, we assume that we do not have internal non-determinism in the form of havoc or calls to external libraries in the trace τ – we will describe this extension later in this section.

Let $wlp(s, \phi)$ be the weakest liberal precondition transformer for a $s \in Stmt$ and $\phi \in Formula$ [18]. $wlp(s, \phi)$ is the weakest formula representing states from which executing s does not lead to assertion failure and on termination satisfies ϕ. It is defined as follows on the structure of statements: $wlp(\mathsf{skip}, \phi) = \phi$, $wlp(\mathtt{x} := e, \phi) = \phi[e/\mathtt{x}]$ (where $\phi[e/\mathtt{x}]$ denotes substituting e for all free occurrences of $\mathtt{x}$), $wlp(\mathsf{assume}\ \psi, \phi) = \psi \Rightarrow \phi$, $wlp(\mathsf{assert}\ \psi, \phi) = \psi \wedge \phi$, and $wlp(s;t, \phi) = wlp(s, wlp(t, \phi))$. Thus $wlp(\tau, \mathsf{true})$ will ensure that no assertion fails along τ. Our current algorithm (Algorithm 2) provides various options to create *predicate (under) covers* of $wlp(\tau, \mathsf{true})$ [22], formulas that imply $wlp(\tau, \mathsf{true})$. Such formulas are guaranteed to block the trace τ from failing.

The first step $ControlSlice$ performs an optimization to prune conditionals from τ that do not control dominate the failing assertion, by performing a variant of the *path slicing* approach [25]. Line 2 performs the wlp computation on the resulting trace τ_1. At this point, ϕ_1 is a Boolean combination of literals from arithmetic, equalities and array theories in *satisfiability modulo theories* (SMT) [34]. $EliminateMapUpdates$ (in line 3) eliminates any occurrence of $write$ from the formula using rewrite rules such as $read(write(e_1, e_2, e_3), e_4) \rightarrow e_2 = e_4\ ?\ e_3 : read(e_1, e_4)$. This rule introduces new equality (aliasing) constraints in the resulting formula that are not present directly in τ. Line 4 chooses a set of atomic formulas from ϕ_2 that match the vocabulary. Finally, the conditional in Line 5 determines the Boolean structure in the resulting expression.

The `MONOMIAL` option specifies that the block expression is a disjunction of atoms from $atoms_1$. Line 7 collects the set of atoms in $atoms_1$ that imply ϕ_2, which in turn implies $wlp(\tau, \mathsf{true})$. We return the clause representing the disjunction of such atoms, which in turn implies $wlp(\tau, \mathsf{true})$. The more expensive $ProjectAtoms(\phi_2, atoms_1)$ returns a formula ϕ_3 that is a CNF expression over $atoms_1$, such that $\phi_3 \Rightarrow \phi_2$, by performing Boolean quantifier elimination of the atoms not present in $atoms_1$. We first transform the formula ϕ_2 into a *conjunctive normal form* (CNF) by repeatedly applying rewrite rules such as $\phi_1 \vee (\phi_2 \wedge \phi_3) \rightarrow (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$. We employ a theorem prover at each step to try simplify intermediate expressions to true or false. Finally, for each clause c in the CNF form, we remove any literal in c that is not present in the set of atoms $atoms_1$.

Example Consider the example FooBar in Figure 1, and the trace τ that corresponds to violation of assert $\mathsf{w} \neq \mathsf{NULL}$. The trace is a sequential composition of the following statements: $\mathsf{z} := \mathsf{x\_1}$, $\mathsf{m[z]} := \mathsf{NULL}$, $\mathsf{x} := \mathsf{x\_2}$, $\mathsf{w} := \mathsf{m[x]}$, assert $\mathsf{w} \neq \mathsf{NULL}$, where we have replaced calls to Lib1 and Lib2 with x_1 and x_2 respectively. $wlp(\tau, \mathsf{true})$ is $\mathsf{read(write(m, x\_1, NULL), x\_2)} \neq \mathsf{NULL}$, which after applying $EliminateMapUpdates$ would result in the expression $(\mathsf{x\_1} \neq \mathsf{x\_2} \wedge \mathsf{m[x\_2]} \neq \mathsf{NULL})$. Notice that this is nearly identical to the blocking clause (except the quantifiers and triggers) returned while analyzing FooBar in Figure 3. Let us allow any disequality $e_1 \neq e_2$ atoms in $Vocab$. If we only allow MONOMIAL Boolean structure, there does not exist any clause over these atoms (weaker than false) that suppresses the trace.

Internal non-determinism In the presence of only *input* non-determinism (parameters and globals), the $wlp(\tau, \mathsf{true})$ is a well-scoped expression at entry in terms of parameters and globals. In the presence of *internal* non-determinism (due to havoc statements either present explicitly or implicitly for non-deterministic initialization of local variables), the target of a havoc is universally quantified away ($wlp(\mathsf{havoc}\ \mathtt{x}, \phi) = \forall u : \phi[u/\mathtt{x}]$). However, this is unsatisfactory for several reasons: (a) one has to introduce a fresh quantified variable for different call sites of a function (say Lib1 in Figure 1). (b) Moreover, the quantified formula does not have good *trigger* [17] to instantiate the universally quantified variables u. For a quantified formula, a trigger is a set of sub-expressions containing all the bound variables. To address both these issues, we introduce a distinct predicate unknown_i after the i-th syntactic call to havoc and introduce an assume statement after the havoc (Figure 2): assume $\mathsf{unknown\_i(x)}$, The wlp rules for assume and havoc ensure that the quantifiers are more well-behaved as the resultant formulas have $\mathsf{unknown\_i(x)}$ as a trigger (see Figure 3).

5 Evaluation

We have implemented the ideas described in this paper (Algorithms 1 and 2) in a tool called *AngelicVerifier*, available with sources.[3] *AngelicVerifier* uses the *Corral* verifier [31] as a black box to implement the check $Verify$ used in Algorithm 1. *Corral* performs interprocedural analysis of programs written in the Boogie language; the Boogie program can be generated from either C [15], .NET [5] or Java programs [1]. As an optimization, while running ExplainError, *AngelicVerifier* first tries the MONOMIAL option and falls back to $ProjectAtoms$ when the former returns false.

We empirically evaluate $AngelicVerifier$ against two industrial tools: the *Static Driver Verifier* (SDV) [3] and *PREfix* [9]. Each of these tools come packaged with models of the environment (both harness and stubs) of the programs they target. These models have been designed over several years of testing and tuning by a product team. We ran $AngelicVerifier$ with none of these models and compared the number of code defects found as well as the benefit of treating the missing environment as *angelic* over treating it as *demonic*.

[3] At corral.codeplex.com, project AddOns\AngelicVerifierNull.

5.1 Comparison with SDV

Benchmarks	Procedures	KLOC	CPU(Ks)
Correct (5)	71-235	2.0-19.1	1.1
Buggy (13)	23-139	1.5-6.7	1.7

Fig. 5: SDV Benchmarks

SDV is a tool offered by Microsoft to third-party driver developers. It checks for type-state properties (e.g., locks are acquired and released in strict alternation) on Windows device drivers. SDV checks these properties by introducing monitors in the program in the form of global variables, and instrumenting the property as assertions in the program. We chose a subset of benchmarks and properties from SDV's verification suite that correspond to drivers distributed in the Windows Driver Kit (WDK); their characteristics are mentioned in Figure 5. We picked a total of 18 driver-property pairs, in which SDV reports a defect on 13 of them. Figure 5 shows the range for the number of procedures, lines of code (contained in C files) and the total time taken by SDV (in 1000s of seconds) on all of the buggy or correct instances.

We ran various instantiations of $\mathit{AngelicVerifier}$ on the SDV benchmarks:

- DEFAULT: The vocabulary includes aliasing constraints ($e_1 \neq e_2$) as well as arbitrary expressions over monitor variables.
- NOTS: The vocabulary only includes aliasing constraints.
- NOALIAS: The vocabulary only includes expressions over the monitor variables.
- NOEE: The vocabulary is empty. In this case, all traces returned by Corral are treated as bugs without running ExplainError. This option simulates a demonic environment.
- DEFAULT+HARNESS: This is the same as DEFAULT, but the input program includes a stripped version of the harness used by SDV. This harness initializes the monitor variables and calls specific procedures in the driver. (The actual harness used by SDV is several times bigger and includes initializations of various data structures and flags as well.)

Example: Figure 6 contains code snippets inspired from real code in our benchmarks. We use it to highlight the differences between the various configurations of $\mathit{AngelicVerifier}$ described above.

- The assertion in Figure 6(a) will be reported as a bug by NOTS but not DEFAULT because $\mathsf{LockDepth} > 1$ is not a valid atom for NOTS.
- The assertion in Figure 6(c) will be reported as a bug by NOALIAS but not DEFAULT because it requires a specifiction that constrains aliasing in the environment. For instance, DEFAULT constrains the environment by imposing $(x \neq \mathsf{irp} \wedge y \neq \mathsf{irp}) \vee (z \neq \mathsf{irp} \wedge y \neq \mathsf{irp})$, where x is $\mathsf{devobj} \rightarrow \mathsf{DeviceExtension} \rightarrow \mathsf{FlushIrp}$, y is $\mathsf{devobj} \rightarrow \mathsf{DeviceExtension} \rightarrow \mathsf{LockIrp}$ and z is $\mathsf{devobj} \rightarrow \mathsf{DeviceExtension} \rightarrow \mathsf{BlockIrp}$.
- The procedures called Harness in Figure 6 are only available under the setting DEFAULT+HARNESS. The assertion in Figure 6(a) will not be reported by DEFAULT as it is always possible (irrespective of the number of calls to KeAcquireSpinLock and KeReleaseSpinLock) to construct an initial value of LockDepth that suppresses the assertion. When the (stripped) harness is present, this assertion will be reported.

```
// monitor variable
int LockDepth;

// This procedure is only
// available under the option
// default +harness
void Harness() {
  LockDepth = 0;
  IoCancelSpinLock();
}

void IoCancelSpinLock() {
  KeReleaseSpinLock();
   ...
  KeReleaseSpinLock();
   ...
  KeAcquireSpinLock();
   ...
  KeCheckSpinLock();
}

void KeAcquireSpinLock()
{ LockDepth ++; }

void KeReleaseSpinLock()
{ LockDepth −−; }

void KeCheckSpinLock()
{ assert LockDepth > 0; }
```

(a)

```
const int PASSIVE = 0;
const int DISPATCH = 2;
//monitor variable
int irqlVal ;

// This procedure is only
// available under the option
// default +harness
void Harness() {
   irqlVal = PASSIVE;
   KeRaiseIrql ();
}

void KeRaiseIrql () {
   ...
   irqlVal = DISPATCH;
   ...
   KeReleaseIrql ();
}

void KeReleaseIrql () {
   assert irqlVal == PASSIVE;
   irqlVal = DISPATCH;
}
```

(b)

```
int completed;
IRP ∗ global_irp ;

void DispatchRoutine(DO ∗devobj,
                     IRP ∗irp) {
  completed = 0;
   global_irp = irp ;
  DE ∗de = devobj→DeviceExtension;
   ...
  IoCompleteRequest(de→FlushIrp);
   ...
  IoCompleteRequest(de→BlockIrp);
   ...
  IoCompleteRequest(de→LockIrp);
}

void IoCompleteRequest(IRP ∗p) {
  if (p == global_irp ) {
     assert completed != 1;
    completed = 1;
  }
}
```

(c)

Fig. 6: Code snippets, in C, illustrating the various settings of *AngelicVerifier*

Note that the assertion failure in Figure 6(b) will be caught by both DEFAULT and DEFAULT+HARNESS.

The results on SDV benchmarks are summarized in Table 1. For each *AngelicVerifier* configuration, we report the cumulative running time in thousands of seconds (**CPU**), the numbers of bugs reported (**B**), and the number of false positives (**FP**) and false negatives (**FN**). The experiments were run (sequentially, single-threaded) on a server class machine with two Intel(R) Xeon(R) processors (16 logical cores) executing at 2.4 GHz with 32 GB RAM.

NOEE reports a large number of false positives, confirming that a demonic environment leads to spurious warnings. The DEFAULT configuration, on the other hand, reports no false positives! It is overly-optimistic in some cases resulting in missed defects. It is clear that the out-of-the-box experience, i.e., before environment models have been written, of *AngelicVerifier* (low false positives, few false negatives) is far superior to a demonic verifier (very high false positives, few false negatives).

The DEFAULT+HARNESS configuration shows that once the tool could use the (stripped) harness, it found all bugs reported by SDV. The configurations NOTS and NOALIAS show that the individual components of the vocabulary were necessary for inferring the right environment specifiction in the DEFAULT configuration. We also note that the running time of our tool is several times higher than that of SDV; instead of the tedious manual environment modeling effort, the cost shifts to higher running time of the automated verifier.

	DEFAULT				DEFAULT+HARNESS				noEE				noTS				noAlias			
Bench	**CPU(Ks)**	**B**	**FP**	**FN**	**CPU(Ks)**	**B**	**FP**	**FN**	**CPU(Ks)**	**B**	**FP**	**FN**	**CPU(Ks)**	**B**	**FP**	**FN**	**CPU(Ks)**	**B**	**FP**	**FN**
Correct	9.97	0	0	0	16.8	0	0	0	0.28	12	12	0	4.20	2	2	0	15.1	0	0	0
Buggy	3.19	9	0	4	3.52	13	0	0	0.47	21	13	5	2.58	14	3	2	1.42	10	3	6

Table 1: Results on SDV benchmarks

	stats		*PREfix*	DEFAULT							DEFAULT-AA	
Bench	**Procs**	**KLOC**	**B**	**CPU(Ks)**	**B**	**PM**	**FP**	**FN**	**PRE-FP**	**PRE-FN**	**CPU(Ks)**	**B**
Mod 1	453	37.2	14	2.7	26	14	4	0	0	1	1.8	26
Mod 2	64	6.5	3	0.2	0	0	0	3	0	0	0.2	0
Mod 3	479	56.6	5	5.8	11	3	4	2	0	1	1.7	6
Mod 4	382	37.8	4	1.8	3	0	0	0	4	3	1.1	2
Mod 5	284	30.9	6	0.8	12	6	1	0	0	0	0.4	11
Mod 6	37	8.4	7	0.1	10	7	0	0	0	0	0.1	10
Mod 7	184	20.9	10	0.6	11	10	0	0	0	1	0.4	11
Mod 8	400	43.8	5	2.9	15	5	1	0	0	1	1.0	15
Mod 9	40	3.2	7	0.1	8	7	0	0	0	0	0.1	8
Mod 10	998	76.5	7	24.9	8	3	1	4	0	4	16.0	4
total	–	321	68	39.9	104	54	11	9	4	11	22.8	93

Table 2: Comparison against PREfix on checking for null-pointer dereferences

5.2 Comparison against PREfix

PREfix is a production tool used internally within Microsoft. It checks for several kinds of programming errors, including checking for null-pointer dereferences, on the Windows code base. We targeted *AngelicVerifier* to find null-pointer exceptions and compared against PREfix on 10 modules selected randomly, such that PREfix reported at least one defect in the module. Table 2 reports the sizes of these modules. (The names are hidden for proprietary reasons.)

We used two *AngelicVerifier* configurations: DEFAULT-AA uses a vocabulary of only aliasing constraints. DEFAULT uses the same vocabulary along with angelic assertions: an assert false is injected after any statement of the form assume e == null. This enforces that if the programmer anticipated an expression being *null* at some point in the program, then *AngelicVerifier* should not impose an environment specification that makes this check redundant.

Scalability. This set of benchmarks were several times harder than the SDV benchmarks for our tool chain. This is because of the larger codebase, but also because checking *null*-ness requires tracking of pointers in the heap, whereas SDV's type-state properties are mostly control-flow based and require minimal tracking of pointers. To address the scale, we use two standard tricks. First, we use a cheap alias analysis to prove many of the dereferences safe and only focus *AngelicVerifier* on the rest. Second, *AngelicVerifier* explores different entrypoints of the program in parallel. We used the same machine as for the previous experiment, and limited parallelism to 16 threads (one per available core). Further, we optimized ExplainError to avoid looking at assume statements along the trace, i.e., it can only block the failing assertion. This can result in ExplainError returning a stronger-than-necessary condition but improves the convergence time of *AngelicVerifier*. This is a limitation that we are planning to address in future work.

Table 2 shows the comparison between PREfix and *AngelicVerifier*. In each case, the number of bug reports is indicated as **B** and the running time as **CPU** (in thousands of seconds). We found *AngelicVerifier* to be more verbose than PREfix, producing a higher number of reports (104 to 68). However, this was mostly because *AngelicVerifier* reported multiple failures with the same cause. For instance,

$\mathsf{x} = \mathsf{null}; \mathsf{if}(...)\{*\mathsf{x} = ...\}\mathsf{else}\{*\mathsf{x} = ...\}$ would be flagged as two buggy traces by *AngelicVerifier* but only one by PREfix. Thus, there is potential for post-processing *AngelicVerifier*'s output, but this is orthogonal to the goals of this paper.

We report the number of PREfix traces matched by some trace of *AngelicVerifier* as **PM**. To save effort, we consider all such traces as true positives. We manually examined the rest of the traces. We classified traces reported by *AngelicVerifier* but not by PREfix as either false positives of *AngelicVerifier* (**FP**) or as false negatives of PREfix (**PRE-FN**). The columns **FN** and **PRE-FP** are the duals, for traces reported by PREfix but not by *AngelicVerifier*.

PREfix is not a desktop application; one can only invoke it as a background service that runs on a dedicated cluster. Consequently, we do not have the running times of PREfix. *AngelicVerifier* takes 11 hours to consume all benchmarks, totaling 321 KLOC, which is very reasonable (for, say, overnight testing on a single machine).

Most importantly, *AngelicVerifier* is able to find most (80%) of the bugs caught by PREfix, without any environment modeling! We verified that under a demonic environment, the Corral verifier reports 396 traces, most of which are false positives.

AngelicVerifier has 11 false positives; 5 of these are due to missing stubs (e.g., a call to the KeBugCheck routine does not return, but *AngelicVerifier*, in the absence of its implementation, does not consider this to be a valid specification). All of these 5 were suppressed when we added a model of the missing stubs. The other 6 reports turn out to be a bug in our compiler front-end, where it produced the wrong IR for certain features of C. (Thus, they are not issues with *AngelicVerifier*.) *AngelicVerifier* has 9 false negatives. Out of these, 1 is due to a missing stub (where it was valid for it to return a *null* pointer), 4 due to Corral timing out, and 5 due to our front-end issues.

Interestingly, PREfix misses 11 valid defects that *AngelicVerifier* reports. Out of these, 6 are reported by *AngelicVerifier* because it finds an inconsistency with an angelic assertion; we believe PREfix does not look for inconsistencies. We are unsure of the reason why PREfix misses the other 5. We have reported these new defects to the product teams and are awaiting a reply. We also found 4 false positives in PREfix's results (due to infeasible path conditions).

A comparison between DEFAULT and DEFAULT-AA reveals that 11 traces were found because of an inconsistency with an angelic assertion. We have already mentioned that 6 of these are valid defects. The other 5 are again due to front-end issues.

In summary, *AngelicVerifier* matched 80% of PREfix's reports, found new defects, and reported very few false positives.

6 Related work

Our work is closely related to previous work on abductive reasoning [20,7,10,11] in program verification. Dillig et al. [20] perform abductive reasoning based on quantifier elimination of variables in wlp that do not appear in the minimum satisfying assignment of $\neg wlp$. The method requires quantifier elimination that is difficult in the presence of richer theories such as quantifiers and uninterpreted functions. Our method $ProjectAtoms$ can be seen as a (lightweight) method for performing Boolean quantifier elimination (without interpreting the theory predicates) that we have found to be effective in practice. It can be shown that the specifications obtained by the two methods can be incomparable, even for arithemtic programs. Calcagno et al. use bi-abductive reasoning to perform bottom-up shape analysis [10] of programs, but performed only in the context of intraprocedural reasoning. In comparison of this work, we provide configurability by being able to control parts of vocabulary and the check for permissiveness using $\hat{\mathcal{A}}$. The work on *almost-correct specifications* [7] provides a method for *minimally* weakening the wlp over a set of predicates to construct specifications that disallow dead code. However, the method is expensive and can be only applied intraprocedurally.

Several program verification techniques have been proposed to detect semantic inconsistency bugs [21] in recent years [19,23,36]. Our work can be instantiated to detect this class of bugs (even interprocedurally); however, it may not be the most scalable approach to perform the checks. The work on *angelic non-determinism* [8] allows for checking if the non-deterministic operations can be replaced with deterministic code to succeed the assertions. Although similar in principle, our end goal is bug finding with high confidence, as opposed to program synthesis. The work on *angelic debugging* [12] and BugAssist [26] similarly look for relevant expressions to relax to fix a failing test case. The difference is that the focus is more on debugging failing test cases and repairing a program.

The work on ranking static analysis warnings using statistical measures is orthogonal and perhaps complementary to our technique [28]. Since these techniques do not exploit program semantics, such techniques can only be used as a post-processing step (thus offering little control to users of a tool). Finally, work on *differential static analysis* [2] can be leveraged to suppress a class of warnings with respect to another program that can serve as a specification [29,32]. Our work does not require any additional program as a specification and therefore can be more readily applied to standard verification tasks. The work on CBUGS [27] leverages sequential interleavings as a specification while checking concurrent programs.

7 Conclusions

We presented the angelic verification framework that constrains a verifier to search for warnings that cannot be precluded with acceptable specifications over unknowns from the environment. Our framework is parameterized to allow a user to choose different instantiations to fit the precision-recall tradeoff. Preliminary experiments indicate that such a tool can indeed be competitive with industrial tools, even without any modeling effort. With subsequent modeling (e.g. adding a harness), the same tool can find more interesting warnings.

References

1. S. Arlt and M. Schäf. Joogie: Infeasible code detection for java. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 767–773, 2012.

2. T. Ball, B. Hackett, S. K. Lahiri, S. Qadeer, and J. Vanegue. Towards scalable modular checking of user-defined properties. In *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*, pages 1–24, 2010.
3. T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.
4. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis For Software Tools and Engineering (PASTE '05)*, pages 82–87, 2005.
5. M. Barnett and S. Qadeer. BCT: A translator from MSIL to Boogie. Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation, 2012.
6. A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, Feb. 2010.
7. S. Blackshear and S. K. Lahiri. Almost-correct specifications: a modular semantic framework for assigning confidence to warnings. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 209–218, 2013.
8. R. Bodík, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *Principles of Programming Languages (POPL '10)*, pages 339–352, 2010.
9. W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper.*, 30(7):775–802, 2000.
10. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 289–300, 2009.
11. S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Programming Language Design and Implementation (PLDI '09)*, pages 363–374, 2009.
12. S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 121–130, New York, NY, USA, 2011. ACM.
13. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
14. E. M. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, pages 368–371, 2003.
15. J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *Principles of Programming Languages (POPL '09)*, pages 302–314, 2009.
16. P. Cousot and R. Cousot. Abstract interpretation : A Unified Lattice Model for the Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Symposium on Principles of Programming Languages (POPL '77)*. ACM Press, 1977.
17. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
18. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 1975.
19. I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Programming Language Design and Implementation (PLDI '07)*, pages 435–445, 2007.

20. I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 181–192, New York, NY, USA, 2012. ACM.
21. D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, 2001.
22. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV '97)*.
23. J. Hoenicke, K. R. M. Leino, A. Podelski, M. Schäf, and T. Wies. Doomed program points. *Formal Methods in System Design*, 37(2-3):171–199, 2010.
24. F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft: Software verification platform. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, pages 301–306, 2005.
25. R. Jhala and R. Majumdar. Path slicing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 38–47, 2005.
26. M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 437–446, 2011.
27. S. Joshi, S. K. Lahiri, and A. Lal. Underspecified harnesses and interleaved bugs. In *Principles of Programming Languages (POPL '12)*, pages 19–30. ACM, 2012.
28. T. Kremenek and D. R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Static Analysis Symposium (SAS '03)*, LNCS 2694, pages 295–315, 2003.
29. S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 345–355, 2013.
30. S. K. Lahiri, S. Qadeer, J. P. Galeotti, J. W. Voung, and T. Wies. Intra-module inference. In *Computer Aided Verification, 21st International Conference, CAV 2009*, LNCS 5643, pages 493–508, 2009.
31. A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 427–443, 2012.
32. F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear. Verification modulo versions: towards usable verification. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 32, 2014.
33. K. L. McMillan. An interpolating theorem prover. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*.
34. Satisfiability Modulo Theories Library (SMT-LIB). Available at `http://goedel.cs.uiowa.edu/smtlib/`.
35. A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *IEEE Symposium of Logic in Computer Science(LICS '01)*, 2001.
36. A. Tomb and C. Flanagan. Detecting inconsistencies via universal reachability analysis. In *International Symposium on Software Testing and Analysis (ISSTA '12)*, 2012.