# Almost-Correct Specifications

## A Modular Semantic Framework for Assigning Confidence to Warnings

Sam Blackshear
University of Colorado, Boulder
samuel.blackshear@colorado.edu

Shuvendu K. Lahiri
Microsoft Research, Redmond
shuvendu@microsoft.com

## Abstract

Modular assertion checkers are plagued with false alarms due to the need for precise environment specifications (preconditions and callee postconditions). Even the fully precise checkers report assertion failures under the most demonic environments allowed by unconstrained or partial specifications. The inability to preclude overly adversarial environments makes such checkers less attractive to developers and severely limits the adoption of such tools in the development cycle.

In this work, we propose a parameterized framework for prioritizing the assertion failures reported by a modular verifier, with the goal of suppressing warnings from overly demonic environments. We formalize *almost-correct specifications* as the minimal weakening of an angelic specification (over a set of predicates) that precludes any dead code intraprocedurally. Our work is inspired by and generalizes some aspects of semantic inconsistency detection. Our formulation allows us to lift this idea to a general class of warnings. We have developed a prototype ACSPEC, which we use to explore a few instantiations of the framework and report preliminary findings on a diverse set of C benchmarks.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification

***Keywords*** Program verifiers, false alarms, predicate abstraction

## 1. Introduction

*Talking about false positive rate is simplistic since false positives are not all equal. The initial reports matter inordinately; . . . Furthermore, you never want an embarrassing false positive. A stupid false positive implies the tool is stupid. ( "It's not even smart enough to figure* that *out?" ).* [2]

Automatic program verifiers are doomed to report false alarms. False alarms can be the result of analysis imprecision or the result of underspecified environment assumptions. The problem is relevant not only to modular checkers that analyze each procedure in isolation [12] (and therefore need specification of inputs and callees), but also to the more general setting when interprocedural specification inference (e.g., based on abstract interpretation [4]) converges with the best invariants expressible in the domain. Finally, this problem is inevitable when analyzing any open program that has under-constrained inputs and external procedures without source code.

This is unfortunate because excessive false alarms (especially stupid ones) act as a deterrent to the initial adoption of verification tools by average developers. Current best practices (often undocumented) in designing usable static analysis tools mitigate this issue by using several less-than-ideal techniques for reducing false positives. For example, a tool may bake unsound decisions into the analysis, which precludes the tool from finding a class of bugs. A tool may require user annotations, which often hurts initial adoption by average developers. Finally, a tool may suppress alarms using domain-specific and often ad-hoc heuristics, which makes the tool brittle and unpredictable. In spite of the severity of the problem, the subject of prioritizing warnings from program verifiers systematically is relatively under-studied.

In this work, we focus on a post-processing framework for prioritizing alarms in an intraprocedural setting. We assume that the interprocedural specification inference [4] has converged with a set of invariants expressible in the underlying analysis. This assumption allows us to decouple the analysis and the framework for displaying warnings, which is important because our goal is not to define a new static analysis technique. We seek to solve a problem similar to one addressed by statistical methods for ranking alarms [17] or user-guided classification of alarms [10]. Our method is based on deep semantic reasoning of a program (unlike [17]), and can complement the work in [10] by identifying the initial set of warnings to display to the user for further classification.

In this work, we propose a framework for reporting a high-confidence subset of the assertion failures reported by a modular verifier. Our framework is parameterized by a set of predicates $Q$. We use the following heuristic: if a warning can be suppressed by a *simple environment specification* over $Q$, it is likely to be a "stupid" false alarm. On the other hand, if a warning can only be suppressed by a complicated environment specification, it is more more likely to be a true alarm. In this work, we limit ourselves to three metrics for characterizing simple specifications: (i) the specification should not preclude the set of angelic environments, (ii) the specification should at least be permissive enough to allow all program locations in a procedure to be reachable (unless they are unreachable even in an unconstrained environment), and (iii) the representation of the specification should satisfy some quality measures such as (but not limited to) the number of disjunctions in a clausal representation.

This paper describes a framework for inferring these *almost-correct specifications*. We refer to the high-confidence warnings that these specifications reveal as *abstract inconsistency bugs*. When the set of predicates $Q$ contains all the atomic predicates in the representation of the weakest precondition for a procedure, our framework reports a set of warnings corresponding to *semantic*

*PLDI'13,* June 16–19, 2013, Seattle, WA, USA.
Copyright © 2013 ACM 978-1-4503-2014-6/13/06. . . $15.00

```
void Foo(int *c, char *buf, CMD cmd) {
1:  if (*) {
2:   free(c);//A1:assert(!Freed[c]);Freed[c] := true;
3:   free(buf);//A2:assert(!Freed[buf]);Freed[buf] := true;
4:   return;
5:  }
6:  if (cmd == READ) {
7:   if (*) {
8:    free(c);//A3:assert(!Freed[c]);Freed[c] := true;
9:    free(buf);//A4:assert(!Freed[buf]);Freed[buf] := true;
10:   /* ERROR: missing return */
11:  }
12: }
13: free(c);//A5:assert(!Freed[c]);Freed[c] := true;
14: free(buf);//A6:assert(!Freed[buf]);Freed[buf] := true;
15: return;
}
```

**Figure 1.** An example of a C double `free` bug.

*inconsistency bugs* [9, 11, 21], a class of high-confidence bugs that can be identified in a modular fashion. On the other hand, when the set of predicates $Q$ is empty, our framework reports all of the warnings from the underlying modular verifier. Our framework can be instantiated with other predicate sets between these extremes in order to construct various schemes for filtering warnings.

## 1.1 Overview

We begin by providing an informal overview of our work with the help of two illustrative examples.

### 1.1.1 Semantic inconsistency detection

Consider the C program in Figure 1, which is a simplified version of a real program [11]. Initially, we will ignore the "//"-commented text. The program frees the pointers `c` and `buf` by invoking the `free` procedure. The "*" denotes a non-deterministic choice. The procedure `free` has a precondition that the argument should not already be freed; the effect of the procedure is to free the pointer. The procedure has a double-free bug when the sequence of locations `1,6,7,8,9,10,11,13,14,15` is executed; the pointers `c` and `buf` would be freed twice. The error in the program is the missing return in line `10`.

The comments in "//" encode the program statements into a simple intermediate language (described in detail in § 2.1). The program makes the type-state for *freed* explicit by maintaining an array `Freed` that is indexed by an integer address. The `free` procedure is replaced by its precondition and the update to the `Freed` array.

**Demonic environments.** Given this program, let us consider what a sound and precise modular program verifier (such as BOOGIE [1]) would do: (1) It would report a violation of the assertion at `A1` by pretending that `Freed[c]` could be true at entry to `Foo`. Looking at the code, it seems likely that the programmer does not expect this scenario. (2) It would next report a violation of the assertion at `A2` by pretending that `Freed[buf]` holds on input, or that the `c` and `buf` pointers could be aliased (assuming we don't trust the static types of C). Looking at the code, there is no indication that the programmer expects any such aliasing (probably because he/she trusts the static types). (3) It would next complain about locations `A3` and `A4` for similar reasons. (4) Finally, it would complain about `A5` and `A6` by assuming that `c` and `buf` are not freed on entry and that they are not aliased, which corresponds to the real bug.

The exercise illustrates the most severe limitation of modular checking: the absence of precise environment assumptions yields a flood of stupid false alarms that obscure interesting alarms. For this example, the environment consists of inputs `c`, `buf`, `cmd` and the

```
void Bar(..) {
    twoints * data = NULL; /* Initialize data */
L1: data = (twoints *)calloc(100, sizeof(twoints));
L2: if(static_returns_t()) {
        /* FLAW: should check if memory allocation failed */
A1:     data[0].a = 1; ...
    } else {
        if (data != NULL) {
A2:         data[0].a = 1; ...
L3:     } else { }
    }
}
```

**Figure 2.** Example for abstract semantic inconsistency bug.

global `Freed`. Sound modular checkers assume the most demonic environments not precluded by the existing procedure contracts (preconditions and callee postconditions).

**Angelic environments.** Now consider the other extreme. The *weakest (liberal) precondition* $WP$(`Foo`) is the largest set of input states for `Foo` that do not fail any assertion; in other words, the weakest *angelic* environment specification. $WP$(`Foo`) can be represented as:

```
(cmd != READ  &&  !Freed[c] && !Freed[buf] && c != buf)
```

However, the weakest precondition for this example is too strong. It makes code unreachable (namely `A3`, `A4`), which prevents the bug from being reported. This intuition has been exploited by prior work on semantic inconsistency detection [9, 11, 21], and has inspired our work.

Our insight is that if the weakest precondition is too strong according to some metric (such as the absence of intraprocedurally dead code), then we can progressively weaken it until it satisfies the metric (makes all code reachable). The weakening process will reveal assertion failures that were previously suppressed by the weakest precondition. We formalize the notion of *almost-correct specifications* as the set of specifications that can be obtained by *minimally* weakening the representation of weakest precondition as a conjunction of *maximal clauses* (§3.3). For the current example, our method infers a single almost-correct specification:

```
(!Freed[c] && !Freed[buf] && c != buf)
```

which fails only `A5`, the assertion failure corresponding to the true bug.[1]

### 1.1.2 Abstract semantic inconsistency bug

Consider the following C example in Figure 2 from the SAMATE suite of benchmarks [20]. Let us assume that there is an assertion `data != 0` before the access to `data` in lines `A1` and `A2`. The environment for this example consists of the variables modified or returned by the two procedure calls `calloc` and `static_returns_t`. In the absence of any specification about the two procedures, a demonic environment would suggest that the assertion at line `A1` can fail. However, the code alone does not give us enough information to determine if this the environment is too demonic. On the angelic end, the weakest precondition conjures up a correlation between the two procedures and makes it a precondition to the procedure under analysis:

$$\theta_{\mathrm{L1}}.\texttt{static\_returns\_t.return} \implies \theta_{\mathrm{L2}}.\texttt{calloc.return} \mathrel{!}= 0$$

[1] In our semantics, assertion failures terminate execution and therefore two assertion failures cannot happen for the same input. Since every input that fails `A6` also fails `A5` under this environment, `A6` is unreachable and never reported as a failure.

Here $\theta_{\mathtt{L}}.\mathtt{pr.return}$ is a fresh constant that denotes the `return` value of the call to a procedure `pr` at some location `L`. The angelic weakest precondition tries to create correlations between various values in the program (parameters, returns, globals) in order to avoid an assertion failure.

Unlike the previous example, the metric of creating unreachable code cannot be used to counter the angelic specification. There is little evidence to suggest that either the angelic or demonic specification is (or isn't) the intended one.

Our insight is that we can report this warning (albeit with lesser confidence) by lifting semantic inconsistency detection to a more abstract setting. The idea is to take away some of the angelic power of the weakest precondition by restricting the vocabulary over which it can be expressed (i.e., the set of predicates $Q$). One natural abstraction that we describe in this paper is treating conditionals in a program as non-deterministic for the purpose of collecting the set of predicates in $Q$. This abstraction prevents the correlation between the return values of `calloc` and `static_returns_t` that the (concrete) weakest precondition inferred for this example. Instead, the most angelic specification possible under this abstraction of $Q$ is $\theta_{\mathtt{L2}}.\mathtt{calloc.return} \mathrel{!=} 0$, which creates dead code by making location `L3` unreachable. The almost-correct specification (over $Q$) for this example is **true**, which reveals the bug in location `A1`. We call such bugs *abstract semantic inconsistency bugs*, as they are parameterized by the set of predicates $Q$.

The example shows that we are able to lift the idea of semantic inconsistency detection from the concrete domain to an abstraction, which allows us to classify more failures using the same underlying principle. This greatly increases the applicability of semantic inconsistency detection.

### 1.2 Contributions

In this work, we propose a parameterized post-processing framework for prioritizing the assertion failures reported by a modular verifier. Specifically, we make the following contributions: (1) We generalize the notion of semantic inconsistency bugs ( [9, 11, 21]) by parameterizing it with a set of *predicates*. This allows us to apply the idea of semantic inconsistency to larger classes of warnings. (2) We formalize the concept of *almost-correct specifications* as the minimal weakening of the angelic specifications over a set of predicates. These specifications can be used to report high-confidence warnings to the user. (3) We provide a set of generic predicate choices that can be automatically constructed and instantiate our framework with them. (4) We have implemented our techniques in a tool called ACSPEC. Our tool is based on BOOGIE program verifier, but can be used with any off-the-shelf program verifier. (5) We have applied our tool on a diverse set of C programs measuring over 1.5 million LOC from open source and Windows software, and report our preliminary findings.

## 2. Background

In this section, we describe a simple programming language that we use to formalize the ideas in the paper.

### 2.1 Programs

Figure 3 defines the syntax of a simple loop-free and call-free programming language. A program consists of a set of procedures. Each program consists of a set of integer valued variables denoted by *Vars*. Variables are partitioned into globals, procedure parameters, returns, and locals. Integer expressions (denoted by *Expr*) can be constructed from variables, or by applying a function symbol $f$ to a (possibly empty) list of expressions. Boolean expressions (denoted by *Formula*) can be constructed from Boolean constants, or by applying a predicate symbol $p$ to a (possibly empty) list of expressions. The expressions are closed under Boolean connectives $\{\wedge, \neg\}$. Array expressions are modeled with the use of special function symbols *read* (to read a location in an array) and *write* (to return a new array updated at a location) that are constrained by the theory of arrays [6].

$$\begin{array}{llcll}
\mathtt{x},\mathtt{y} & \in & \mathit{Vars} & & \\
e & \in & \mathit{Expr} & :: & \mathtt{x} \mid f(e,\ldots,e) \\
\phi,\psi & \in & \mathit{Formula} & :: & \mathbf{true} \mid \mathbf{false} \mid p(e,\ldots,e) \mid \phi\wedge\phi \mid \neg\phi \\
s,t & \in & \mathit{Stmt} & :: & \mathtt{skip} \mid \mathtt{assert}\ \phi \mid \mathtt{assume}\ \phi \mid \mathtt{x} := e \mid \\
 & & & & \mathtt{havoc}\ \mathtt{x} \mid s;s \mid \mathtt{if}\ (\phi)\ \mathtt{then}\ s\ \mathtt{else}\ s
\end{array}$$

**Figure 3.** A simple programming language.

A (simple) statement can be either a skip (`skip`), an assertion (`assert` $\phi$), an assumption (`assume` $\phi$), or an assignment to a variable. In addition to the usual assignment $\mathtt{x} := e$, the statement `havoc x` assigns a non-deterministic value to the variable `x` — this is used to model a non-deterministic value "*" in an expression. A (compound) statement can be either a sequential composition of two statements or a conditional statement `if` $(\phi)$ `then` $s$ `else` $t$ that executes the statement $s$ if $\phi$ is true, or $t$ otherwise.

Procedure calls are not part of the programming language. Instead, a procedure call is replaced by its specification. Consider a procedure $pr(\mathtt{x})$ : `ret` that takes a parameter `x`, returns `ret`, and modifies a global $\mathtt{gl} \in \mathit{Vars}$. Further, it has a precondition $\psi_1$ (a formula over parameters and globals) and a postcondition $\psi_2$ (a formula over parameters, globals, and returns). A call to a procedure $\mathtt{r} := \mathtt{call}\ pr(e)$ at a control location $l$ is expressed as:

$$\mathtt{assert}\ \psi_1[e/\mathtt{x}]\ ;\ \mathtt{r},\mathtt{gl} := \theta_l.pr.\mathtt{r}, \theta_l.pr.\mathtt{gl}\ ;\ \mathtt{assume}\ \psi_2[\mathtt{r}/\mathtt{ret}]\ ;$$

where $\theta_l.pr.\mathtt{r}, \theta_l.pr.\mathtt{gl}$ are fresh constants unique to the location $l$.

The constructs in the programming language are fairly standard, and we refer the reader to earlier work for an operational semantics of the language [7]. The expressive power of the simple language suffices to precisely provide semantics to many imperative languages such as C, C#, and Java. Fields and objects can be modeled by a map indexed by object identifiers. Object allocation and deallocation can be simulated using extra ghost variables. We refer the reader to previous work on modeling Java [12] and C [3].

### 2.2 Program to logic

DEFINITION 1 ($WP(pr)$). *For a procedure pr, the weakest precondition $WP(pr)$ is the largest set of input states from which no execution fails an assertion.*

A procedure satisfies its contracts if $WP(pr)$ is the set $\lambda x.\mathbf{true}$. For a loop-free and call-free procedure, the check for partial correctness can be reduced to checking satisfiability (modulo the background theories) of a logical formula by variants of Dijkstra's *weakest (liberal) precondition* predicate transformer [8]. The predicate transformer $wp(s,\psi)$ takes as inputs a statement $s \in \mathit{Stmt}$ and a formula $\psi \in \mathit{Formula}$, and constructs a formula. It is defined recursively over the structure of statements as follows:

$$\begin{array}{lcl}
wp(\mathtt{skip},\psi) & = & \psi \\
wp(\mathtt{assume}\ \phi,\psi) & = & \neg\phi \vee \psi \\
wp(\mathtt{assert}\ \phi,\psi) & = & \phi \wedge \psi \\
wp(\mathtt{x} := e,\psi) & = & \psi[e/\mathtt{x}] \\
wp(\mathtt{havoc}\ \mathtt{x},\psi) & = & \forall x.\psi[x/\mathtt{x}] \\
wp(s;t,\psi) & = & wp(s, wp(t,\psi)) \\
wp(\mathtt{if}\ (c)\ \mathtt{then}\ s\ \mathtt{else}\ t,\psi) & = & (\neg c \vee wp(s,\psi)) \wedge (c \vee wp(t,\psi))
\end{array}$$

To verify that $pr(\ldots)\{body\}$ does not fail any assertions, we check that the formula $\neg wp(body,\mathbf{true})$ is unsatisfiable. Since computing $wp(body,\mathbf{true})$ can incur an exponential blowup, program verifiers compute an equisatisfiable formula by first passifying the program [13]. We often overload the expression $wp(pr, \mathbf{true})$ to mean $wp(body, \mathbf{true})$, where *body* is the body of $pr$. The satisfiability of the resultant formula can be checked by suitable decision procedures such as modern Satisfiability Modulo Theories (SMT) solvers [6]. We will refer to the process of converting a program to a formula as verification condition (VC) generation.

### 2.3 Dead and fail set

For a procedure $pr$, let *Locs* denote the set of locations inside $pr$, and $Asserts$ denote the set of assertions inside $pr$. For rest of the discussion, the procedure $pr$, the set of assertions $Asserts$ and the set of locations *Locs* are implicit in every context, unless otherwise noted.

For a formula $\phi \in \mathit{Formula}$ representing a set of input states, we define the following concepts:

1. The set of *dead* locations $\mathit{Dead}(\phi) \doteq \{l \in \mathit{Locs} \mid l$ is not reachable for any state in $\phi\}$.
2. The set of *failed* assertions $\mathit{Fail}(\phi) \doteq \{a \in Asserts \mid a$ can fail on at least one execution from an input in $\phi\}$.

Note that for non-deterministic programs, both $\mathit{Dead}(\phi)$ and $\mathit{Fail}(\phi)$ account for all possible executions starting from a given input state.

We assume that $\mathit{Dead}(\mathbf{true}) = \{\}$; that is, that there are no dead locations from the local perspective of $pr$. In practice, we can ensure that this assumption holds by removing $\mathit{Dead}(\mathbf{true})$ from *Locs* before starting our analysis. To determine the set of dead locations, it suffices to restrict ourselves to only a subset of locations — namely the locations that appear immediately inside "then", and "else" branches as well as the locations that appear after each `assume` statement. The former ensures that each branch can be taken both ways, and the latter ensures that `assume` statements do not block subsequent statements.

Although we choose to define $\mathit{Dead}(\phi)$ in terms of branch coverage in this paper, the definition of $\mathit{Dead}(\phi)$ can be a parameter of our analysis. We can easily replace our definition with any computable metric for expressing when a specification is too strong. For example, we could have defined $\mathit{Dead}(\phi)$ as the set of paths that are infeasible under $\phi$ (i.e., in terms of path coverage rather than in terms of branch coverage), or as the set of observed runtime values at some locations that are precluded by $\phi$.

### 2.4 Clauses

A *predicate* $p$ is an atomic formula over some theory (say arithmetic) without any Boolean connectives ($\neg, \wedge, \vee$) at the outermost scope. A *literal* $l$ is either a predicate or its negation. A *clause* $c \doteq l_1 \vee l_2 \ldots l_k$ is a disjunction over literals. Dually, a *cube* $d \doteq l_1 \wedge l_2 \ldots l_k$ is a conjunction over literals. For a set of clauses $C$, we denote $\Pi(C)$ to be the formula $\left(\mathbf{true} \wedge \bigwedge_{c \in C} c\right)$.

LEMMA 1. *For any two sets of input clauses* $C_1 \subseteq C_2$,

- $\mathit{Dead}(\Pi(C_1)) \subseteq \mathit{Dead}(\Pi(C_2))$, *and*
- $\mathit{Fail}(\Pi(C_2)) \subseteq \mathit{Fail}(\Pi(C_1))$.

## 3. Semantic inconsistency and almost-correct specifications

In this section, we formalize the notion of abstract semantic inconsistency bugs (*SIB*). First, we capture the essential intuition behind semantic inconsistency [11] in the context of modular assertion checking (§ 3.1). We then lift the idea of inconsistency to a more general setting by parameterizing it with a set of predicates (abstract *SIB*). In § 3.3, we describe almost-correct specifications as a witness for abstract *SIB*s.

### 3.1 Semantic inconsistency bugs

Let us start by formalizing semantic inconsistency bugs in the context of modular assertion checking and explaining their relationship with weakest precondition.

DEFINITION 2 (*SIB*). *A procedure pr has a semantic inconsistency bug (SIB) if the largest set of input states $\phi$ that satisfies all assertions (i.e. $\mathit{Fail}(\phi) = \{\}$) creates dead code (i.e. $\mathit{Dead}(\phi) \neq \{\}$).*

We can establish a connection between inconsistency bugs and weakest precondition for a procedure:

PROPOSITION 1. *A procedure pr containing at least one assertion has a SIB if and only if* $\mathit{Dead}(\mathit{WP}(pr)) \neq \{\}$ .

Note that the above formulation is purely semantic and only relies on the assertions present in the code. For the double `free()` example in § 1.1.1, the weakest precondition can be expressed as:

```
(cmd != READ && !Freed[c] && !Freed[buf] && c != buf)
```

However, this causes `A3` and `A4` to become unreachable, and thus the procedure has a *SIB*.

The case where $\mathit{WP}(pr)$ is equivalent to $\{\}$ (that is, when every input fails at least one assertion) is a special case of *SIB* bugs where $\mathit{Dead}(\mathit{WP}(pr))$ contains every statement in the procedure.

### 3.2 Abstract semantic inconsistency bugs

Although *SIB*s are interesting and useful to detect, they represent only a small fraction of all possible bugs. As the example in § 1.1.2 shows, the concrete $\mathit{WP}(pr)$ is the most angelic specification on the environment, and therefore examining the dead locations it creates will not reveal any errors except *SIB*s. Our goal in this section is to lift the connection between $\mathit{WP}(pr)$ and *SIB*s to a more general setting. Our intuition is simple: by restricting the abstraction (or vocabulary) over which we can describe the environment specifications, some warnings can be categorized as *SIB* bugs *with respect to that abstraction*, even though there may be no *SIB* bugs by the concrete definition defined earlier in Proposition 1.

We lift the notion of *SIB* to *abstract* semantic inconsistency bugs by parameterizing the definition of inconsistency with respect to a *predicate abstraction* [4, 14]. Given a set of atomic formulas $Q$, the abstraction function $\alpha_Q : \mathit{Formula} \rightarrow \mathit{Formula}$ maps a formula $\phi$ to the strongest approximation of $\phi$ expressible as Boolean combinations of $Q$. For example, if $Q = \{x = 0, y = 0\}$ and $\phi \doteq (x = 0 \wedge y = 0) \vee (x = 1 \wedge y = 1)$, then $\alpha_Q(\phi) \doteq (x = 0 \wedge y = 0) \vee (x \neq 0 \wedge y \neq 0)$. For a given formula $\phi \in \mathit{Formula}$, we denote $\beta_Q$ as the weakest under-approximation of $\phi$ with respect to a given set of predicates $Q$. It is represented as $\neg\alpha_Q(\neg\phi)$. We can establish a partial order on the set of all predicate abstractions by a subset relation $\subseteq$ on the set of predicates. It is not hard to see that given a formula $\phi$ and two sets of predicates $Q \subseteq P$, $\phi \implies \alpha_P(\phi) \implies \alpha_Q(\phi)$ and $\beta_Q(\phi) \implies \beta_P(\phi) \implies \phi$. In other words, a larger set of predicates provides a more precise approximation (both over and under) of $\phi$.

DEFINITION 3 (Abstract *SIB*). *A procedure pr has an abstract semantic inconsistency bug (abstract SIB) with respect to a set of predicates Q if* $\mathit{Dead}(\beta_Q(wp(pr, \mathbf{true}))) \neq \{\}$.

The intuition behind this definition is quite simple: the set of environments $\beta_Q(wp(pr, \mathbf{true}))$ is the weakest set of angelic environments expressible using $Q$ that do not fail any assertion in the procedure. If this set of states gives rise to dead code, then the procedure has an abstract *SIB*. Henceforth, we will implicitly assume $Q$ as the set of predicates under consideration unless otherwise noted.

Note that when $Q$ contains all the atomic predicates from $wp(pr, \mathbf{true})$, then abstract *SIB*s (with respect to $Q$) and *SIB*s coincide. Unlike concrete *SIB*s, abstract *SIB*s are not guaranteed to correspond to either true bugs or dead code. However, the example in § 1.1.2 illustrates that some true bugs can be categorized as abstract *SIB*s, but not as concrete *SIB*s. At the other extreme when $Q \doteq \{\}$, any assertion failure will manifest itself as an abstract *SIB* since the only specifications allowed over $Q = \{\}$ are $\mathbf{true}$ and $\mathbf{false}$. We can establish the following relationship between two sets of predicates:

PROPOSITION 2. *For two sets of predicates $Q \subseteq P$, if pr has an abstract SIB under $P$, then $pr$ has an abstract SIB under $Q$.*

### 3.3 Almost-correct specifications

So far, we have only shown how to determine whether a procedure has an abstract $SIB$ or not. In this section, we explain how to find the assertions that might fail in a procedure with a $SIB$ (abstract or concrete). We provide a witness for each potential assertion failure by providing an *almost-correct* specification under which that assertion might fail. For this section, we assume that the procedure $pr$ is fixed and therefore implicit in the definitions.

Let $Formula_Q \subseteq Formula$ be the set of all formulas that can be constructed by a Boolean combination over the predicates in $Q$.

DEFINITION 4 ($AlmostCorrectSpecs(Q)$). *For a set of predicates $Q$ and an integer $k \geq 0$, $AlmostCorrectSpecsK(Q, k) \subseteq Formula_Q$ is defined as set of formulas $\phi$ where*

1. *$\beta_Q(wp(pr, \mathbf{true})) \implies \phi$, and*
2. *$Dead(\phi) = \{\}$, and*
3. *$|Fail(\phi)| = k$, and*
4. *for any $\psi \in Formula_Q$ such that $\beta_Q(wp(pr, \mathbf{true})) \implies \psi \implies \phi$, either $\phi \implies \psi$ or $Dead(\psi) \neq \{\}$.*

*$AlmostCorrectSpecs(Q)$ is $AlmostCorrectSpecsK(Q, k)$ for the smallest $k \geq 0$ such that $AlmostCorrectSpecsK(Q, k) \neq \{\}$.*

$AlmostCorrectSpecsK(Q, k)$ contains precisely those formulas $\phi$ that are at least as weak as the $\beta_Q(wp(pr, \mathbf{true}))$, do not create any dead code, induce $k$ assertion failures, and cannot be strengthened over the vocabulary $Q$ without creating dead code. $AlmostCorrectSpecs(Q)$ contains the set of specifications that induce the minimum number of failures. When a procedure has no abstract *SIB*s, then $AlmostCorrectSpecs(Q)$ is precisely the set of formulas representing $\beta_Q(wp(pr, \mathbf{true}))$, which (by definition) induce $k = 0$ failures.

**Algorithm 1** $FindAbstractSIBs(pr, Q)$

```
Require: A procedure $pr$ with a set of assertions $Asserts$
Require: A set of predicates $Q$
1: $\phi \leftarrow \beta_Q(wp(pr, \mathbf{true}))$
2: if $Dead(\phi) \neq \{\}$ then
3:     $s \leftarrow SIB$ /* abstract */
4: else
5:     $s \leftarrow MAYBUG$ /* low confidence warnings */
6: end if
7: $\Psi \leftarrow FindAlmostCorrectSpecs(pr, Q)$
8: $E \leftarrow \bigcup_{\psi \in \Psi} Fail(\psi)$
9: return $(s, E)$
```

Algorithm 1 shows how to find abstract *SIB*s. Line 7 invokes a method for generating a set of almost-correct specifications over $Q$; we describe this method in the next section. Finally, line 8 collects the set of assertion failures that are possible under the almost-correct specifications.

## 4. Computing almost-correct specifications

In this section, we describe one method to compute the set of almost-correct specifications formulated in Definition 4. The steps consist of obtaining a *canonical* representation of $\beta_Q(wp(pr, \mathbf{true}))$ in a conjunctive normal form (§ 4.1), and then performing a greedy search (with pruning) to find the almost-correct specifications (§ 4.2). We discuss Boolean simplification of the resultant specifications and further weakening the specifications based on some syntactic clause quality measures (§ 4.3). Finally, we provide different methods for constructing the set of predicates $Q$ starting with the precise set of predicates for representing $wp(pr, \mathbf{true})$.

### 4.1 Predicate cover $PredicateCover_Q(pr)$

For a procedure $pr$, let $VC(pr) \in Formula$ be the verification condition (VC) that is equisatisfiable with $\neg wp(pr, \mathbf{true})$ — the size of $VC(pr)$ is usually almost linear in the size of $pr$ [1] when $pr$ is converted to static single assignment (SSA) form. Given a set of predicates $Q$, the predicate cover $\beta_Q(wp(pr, \mathbf{true}))$ is computed by enumerating all the assignments (ALL-SAT) over $Q$ that satisfy $VC(pr)$ and then obtaining a set of clauses by negating these assignments. The clauses obtained this way are *maximal* — i.e., each predicate in $Q$ either appears in a positive or negative polarity in each clause. Let us denote this operation as $PredicateCover_Q(pr)$. Such a conjunctive normal form over maximal clauses provides a canonical representation of $\beta_Q(wp(pr, \mathbf{true}))$. The canonical representation is important for computing the almost-correct specifications, as we will describe in the next section. The algorithm is fairly standard in predicate abstraction and we refer readers to earlier work [19].

### 4.2 Weakening predicate cover

The algorithm $FindAlmostCorrectSpecs(pr, Q)$ finds the set of almost-correct specifications. It first computes the set of maximal clauses $C$ using the predicate cover operation described earlier. It performs a greedy search over the space of clauses by selecting a clause to drop and then inspecting the $Dead()$ and $Fail()$ counts of the resulting clause sets. The algorithm starts with the set of clauses $C$ satisfying $Fail(\Pi(C)) = \{\}$ and computes the minimum number of failures that can result from dropping clauses from $C$ to make all the code reachable. The procedure returns a set of formulas, each representing an almost-correct specification.

For the purpose of this section, we will treat the operations *Normalize* and *PruneClauses* as identity functions that simply return the input set of clauses — we will consider more interesting uses in § 4.3. The algorithm maintains a frontier set $S$ of sets of clauses that have a non-empty dead set and iterates (via the outer while loop) until $S$ is empty. It also maintains the minimum failure count (*MinFail*) for any clause set with an empty dead set. This count is initialized to the size of $Asserts$. At each step, the algorithm extracts a clause set $C_1 \in S$, and enumerates each subset of clauses $C_2$ (the inner for loop) by dropping a clause $c$ from $C_1$. Each sub-clause $C_2$ is either added to $S$ (if $Dead(C_2) \neq \{\}$ and $|Fail(C_2)| \leq MinFail$, or $Dead(C_2) = \{\}$ and $|Fail(C_2)| = 0$) or pruned (if $Dead(C_2) \neq \{\}$ and $|Fail(C_2)| > MinFail$). If $|Dead(C_2)| = 0$ and $|Fail(C_2)| \leq MinFail$, then we add $C_2$ to the output set $U$. When $|Fail(C_2)| < MinFail$, then the output set is flushed and the $MinFail$ is reduced to $|Fail(C_2)|$.

THEOREM 1. *The following statements are true:*

1. *$FindAlmostCorrectSpecs(pr, Q) \subseteq AlmostCorrectSpecs(Q)$.*
2. *For each $\phi \in AlmostCorrectSpecs(Q)$, there exists a $\psi \in FindAlmostCorrectSpecs(pr, Q)$ such that $\phi \Leftrightarrow \psi$.*

The set $AlmostCorrectSpecs(Q)$ may contain different syntactic representations of the same specification and therefore may con-

**Algorithm 2** *FindAlmostCorrectSpecs*$(pr, Q)$

```
Require: A procedure $pr$ with a set of assertions $Asserts$
Require: A set of predicates $Q$
Ensure: Set of formulas representing $AlmostCorrectSpecs(C)$
 1: $C \leftarrow PredicateCover_Q(pr)$
 2: if $Dead(\Pi(C)) = \{\}$ then
 3:     return $\{\Pi(PruneClauses(Normalize(C)))\}$
 4: end if
 5: $S \leftarrow \{C\}$ /*Frontier set */
 6: $T \leftarrow \{\}$ /* Visited set */
 7: $U \leftarrow \{\}$ /* Output set */
 8: $MinFail \leftarrow |Asserts|$ /* Smallest set of failures*/
 9: while $S \neq \{\}$ /* Frontier is non-empty */ do
10:     $C_1 \leftarrow Choose(S)$
11:     for each clause $c \in C_1$ do
12:         $C_2 \leftarrow C_1 \setminus \{c\}$ /* Weaken by one clause */
13:         if $C_2 \in T$ then
14:             continue /* Visited already */
15:         end if
16:         $T \leftarrow T \cup \{C_2\}$ /* Add to Visited */
17:         if $|Fail(C_2)| > MinFail$ then
18:             continue /* $MinFail$ can only decrease */
19:         end if
20:         if $|Dead(C_2)| \neq 0$ then
21:             $S \leftarrow S \cup \{C_2\}$ /* Add to the frontier set */
22:         else if $|Fail(C_2)| = 0$ then
23:             $S \leftarrow S \cup \{C_2\}$ /* Add to the frontier set */
24:         else if $|Fail(C_2)| = MinFail$ then
25:             $U \leftarrow U \cup \{C_2\}$ /* Add it to $U$ */
26:         else
27:             /* $|Fail(C_2)| < MinFail$ */
28:             $MinFail \leftarrow |Fail(C_2)|$ /* Decrease $MinFail$ */
29:             $U \leftarrow \{C_2\}$ /* Clear $U$ and add $C_2$ */
30:         end if
31:     end for
32: end while
33: return $\bigcup_{C_1 \in U} \Pi(PruneClauses(Normalize(C_1)))$
```

tain more formulas than $FindAlmostCorrectSpecs(pr, Q)$. The proof follows from the observations that (a) the set of maximal cubes (negation of maximal clause) over $Q$ partitions the set of all states, and (b) dropping a maximal clause from the set of maximal clauses representing a formula weakens the formula by exactly one maximal cube.

## 4.3 Clause simplification and pruning

We can additionally parameterize the almost-correct specifications by providing a syntactic quality measure for the set of clauses. For instance, the number of literals in a clause can be a measure of goodness of a specification. Under the hypothesis that good specifications are usually simple and do not contain many disjunctions, we can prune clauses that contain a large number of literals. Alternately, we might try to avoid clauses that correlate the return values of multiple procedure calls.

However, such quality measures cannot be applied directly on the maximal clauses. Consider the maximal clauses $(a \vee b) \wedge (a \vee \neg b)$ over $Q = \{a, b\}$. These clauses have two literals per clause and may be undesirable, whereas an equivalent clause $(a)$ has only one literal. We address this problem by first performing Boolean simplification of the maximal clauses.

The operation $Normalize(C)$ takes a set of clauses $C$ and performs Boolean clause simplification on the set of clauses. It applies the following three rules to a set of clauses (starting with $C$) until a fix-point is reached: (1) *Resolution:* If there are two clauses $(c \vee l), (d \vee \neg l) \in C$, then add $(c \vee d)$ to $C$. (2) *Subsumption:* If there are two clauses $c, (c \vee l) \in C$, then remove $(c \vee l)$ from $C$. (3) *Tautologies:* If there is a clause $(c \vee l \vee \neg l) \in C$, then remove it from $C$.

The function $PruneClauses$ takes a set of clauses and returns a subset of clauses that satisfy the quality measure. Note that the pruning makes the almost-correct specifications weaker and can reveal more warnings — it is not just a syntactic transformation. For the example in § 1.1.2, both schemes (limiting the number of literals per clause to one, or removing clauses containing returns from multiple procedures) will reveal the warning by pruning the clause $\theta_{\mathtt{L1}}.\mathtt{static\_returns\_t.return} \implies \theta_{\mathtt{L2}}.\mathtt{calloc.return\,!=\,0}$.

## 4.4 Mining predicates

We now describe a few choices for constructing the set of predicates in $Q$ automatically for any given procedure $pr$. We start with a method for collecting all the atomic predicates that appear in the expression representing the weakest precondition (§ 4.4.1), followed by two methods that generate smaller sets of predicates (§ 4.4.2 and § 4.4.3).

### 4.4.1 Predicates in $wp(pr, \mathbf{true})$

Recall that if $Q$ contains all the atomic predicates that constitute $wp(pr, \mathbf{true})$, then $\beta_Q(wp(pr, \mathbf{true}))$ is equivalent to $wp(pr, \mathbf{true})$, and abstract *SIB*s and *SIB*s coincide. For a procedure $pr$ with body $body$, the set of predicates collected is $Preds(body, \{\})$, where$Preds(,)$ is defined recursively as follows:

$$\begin{array}{lcl}
Preds(\mathtt{skip}, Q) & = & Q \\
Preds(\mathtt{assume}\ \phi, Q) & = & Atoms(\phi) \cup Q \\
Preds(\mathtt{assert}\ \phi, Q) & = & Atoms(\phi) \cup Q \\
Preds(\mathtt{x} := e, Q) & = & Atoms(Q[e/\mathtt{x}]) \\
Preds(\mathtt{havoc\ x}, Q) & = & Drop(Q, \mathtt{x}) \\
Preds(s; t, Q) & = & Preds(s, Preds(t, Q)) \\
Preds(\mathtt{if}\ (c)\ \mathtt{then}\ s\ \mathtt{else}\ t, Q) & = & Atoms(c) \cup Preds(s, Q) \\
 & & \cup\ Preds(t, Q)
\end{array}$$

Here $Atoms(\phi)$ collects the set of atomic predicates (those do not contain any Boolean connectives), and $Drop(Q, \mathtt{x})$ removes any atomic predicate in $Q$ that contains the variable $\mathtt{x}$. Note the similarity with the $wp(s, \phi)$ transformer from § 2.2, as the goal of the predicate collection above is to collect the atomic predicates that may appear in the $wp(pr, \mathbf{true})$. We describe a few important details about the predicates in $Preds(body, \{\})$.

First, recall that a variable $\mathtt{x}$ that is modified by a call to a procedure $pr_1$ at line $l$ is assigned a special constant $\theta_l.pr_1.\mathtt{x}$ (§ 2.1). These constants appear in the set of predicates $Q$ and in the specifications (see the example in § 1.1.2).

Second, consider the problem of generating atomic predicates in the presence of arrays or maps. For example,

$$wp(\mathtt{x} := write(\mathtt{x}, e_1, e_2), p(read(\mathtt{x}, e_3), e_4))$$

results in the formula $p(read(write(\mathtt{x}, e_1, e_2), e_3), e_4)$ that contains $write$ symbol. We apply rewrite rules (omitted for brevity) to eliminate the $write$ symbols in the predicates in order to make input specifications more readable and intuitive. After eliminating $write$, the above expression corresponds to $e_1 = e_3\ ?\ p(e_2, e_4) : p(read(\mathtt{x}, e_3), e_4)$. The set of atomic predicates in this expression are $\{e_1 = e_3,\ p(e_2, e_4),\ p(read(\mathtt{x}, e_3), e_4)\}$. This explains the presence of the predicate $\mathtt{c} = \mathtt{buf}$ in the weakest precondition for the example of double-free (§ 1.1.1).

### 4.4.2 Ignoring conditionals

Consider the following example:

| Alias | Havoc returns | Ignore conditionals |
|---|---|---|
| *Conc* | F | F |
| $A_0$ | T | F |
| $A_1$ | F | T |
| $A_2$ | T | T |

$$\begin{tikzcd} & \boxed{Conc} \arrow[dl] \arrow[dr] & \\ \boxed{A_0} \arrow[dr] & & \boxed{A_1} \arrow[dl] \\ & \boxed{A_2} & \end{tikzcd}$$

**Figure 4.** The four abstract configurations defined by combining our two abstractions, along with their aliases. Arrows flow from higher precision to lower precision.

```
void Foo(bool c1, bool c2, int *x) {
    if (c1) {...; if (x) {*x = 1;} ... }
    if (c2) {... ; *x = 2; ...}
}
```

The weakest precondition avoids non-null errors by conjuring $\texttt{c2} \implies \texttt{x} \neq 0$ as a possible precondition. This precondition does not create any concrete *SIB*s. One way to take away this angelic power (and reveal warnings as abstract *SIB*s) is to avoid considering the predicates that guard conditionals. We can obtain this by treating a conditional $\texttt{if}\ (c)\ \texttt{then}\ s\ \texttt{else}\ t$ as $\{\texttt{havoc x}; \texttt{if}\ (\texttt{x})\ \texttt{then}\ s\ \texttt{else}\ t;\}$ during predicate collection (where `x` is a fresh program variable). The set of predicates collected for this examples is $Q = \{\texttt{x} \neq 0\}$.

### 4.4.3 Havoc returns

Another way to restrict the set of predicates is to avoid predicates about callee modifications. Instead of assigning fresh constants (e.g. $\theta_l.pr_1.\texttt{x}$) to modified variables (as done by default in § 2.1), we can `havoc` these variables. However, this can also lead to undesired imprecision in cases such as:

```
void Bar(...) {
   havoc x;   // x = getValidPointer(..);
   M[x] := 1; // *x = 1;
}
```

For this example, $wp(\texttt{Bar}, \textbf{true})$ will be **false** because the predicate set is empty.

We henceforth refer to the first abstraction described as "ignore conditionals" (§ 4.4.2) and the second as "havoc returns" (§ 4.4.3). The product of the these two abstractions naturally yields $2^2 = 4$ abstract configurations which we name and describe in Figure 4. The configuration *Conc* refers to concrete *SIB*s (§ 4.4.1). The rest of the configurations $A_0, A_1, A_2$ correspond to abstract *SIB*s resulting from combining the schemes in the previous two sections (§ 4.4.2 and § 4.4.3).

## 5. Implementation and evaluation

***Implementation.*** We have constructed a prototype tool ACSPEC (Almost-Correct SPECifications) that implements the ideas presented in the paper. The tool consists of around 2000 lines of C# code. It accepts a source file in the BOOGIE language [7] and a list of abstractions as input. It outputs whether the procedure has a *SIB* under the abstractions, searches for the set of almost-correct specifications in the predicate vocabulary allowed by the abstractions, and prints the set of errors induced by the specifications. We leverage BOOGIE's optimized verification condition generation along with the Z3 SMT solver [6] for checking the satisfiability of the verification conditions. The current prototype does not yet use the incremental interface to the Z3 prover and regenerates VC for every call to Z3 — this is a major source of inefficiency in the current implementation.

The main modules implemented in the tool are (a) computing the $Fail()$ and $Dead()$ sets, (b) generating the set of predicates under the different options from Figure 4, (c) computing the predicate cover and checking for *SIB*s, (d) searching for almost-correct specifications over a set of clauses, and (e) normalizing clauses and performing clause pruning. We only mention the computation of $Dead()$ briefly here. For each location $l \in Locs$ (one for each block that has an `assume` statement), we introduce an assertion $\texttt{assert}\ b_l$. To compute $Dead(\phi)$, we ask for all assertion failures (over the location asserts) under $\phi$ — the set of assertions that *do not fail* constitute $Dead(\phi)$. For computing $Dead()$, we rely on the completeness of the theorem prover. For the logics we use (equalities, arithmetic, arrays), Z3 ensures completeness.

| Bench | LOC (C) | LOC (BPL) | Procs | Asserts |
|---|---|---|---|---|
| CWE476 | 24582 | 267510 | 561 | 228 |
| CWE690 | 85855 | 941045 | 1865 | 1009 |
| ansicon | 2972 | 38145 | 289 | 325 |
| space | 9566 | 185878 | 263 | 2144 |
| cancel | 827 | 7411 | 9 | 9 |
| event | 872 | 6430 | 7 | 4 |
| firefly | 622 | 8669 | 9 | 22 |
| moufilter | 631 | 7073 | 7 | 25 |
| vserial | 1396 | 21617 | 23 | 102 |
| Drv1 | 97615 | 1401949 | 799 | 14402 |
| Drv2 | 77180 | 1536078 | 1218 | 13843 |
| Drv3 | 31011 | 77760 | 67 | 692 |
| Drv4 | 82720 | 542049 | 401 | 3582 |
| Drv5 | 151516 | 981682 | 661 | 11250 |
| Drv6 | 18577 | 562477 | 490 | 7381 |
| Drv7 | 1171514 | 21845268 | 21626 | 227573 |
| Lib1 | 96158 | 1121121 | 1158 | 10051 |
| Total | 1853614 | 29552162 | 29453 | 292642 |

**Figure 5.** Benchmark statistics. For each benchmark, we give the lines of code in both C and BOOGIE, the number of procedures, and number of assertions.

***Benchmarks.*** We chose a set of 17 C benchmarks drawn from open source programs and the Windows codebase (Figure 5). The CWE benchmarks are from the NIST SAMATE test suite for static analysis[2], space is a flight control software[3], ansicon is a console text processor, and the rest of the small benchmarks are sample drivers from Windows Driver Kit[4]. The set of larger Windows benchmarks has been anonymized. Each Drv* is a set of multiple drivers in Windows and Lib1 is a core component of the Windows kernel. All benchmarks were compiled from C to the BOOGIE programming language using the HAVOC tool [3]. Although our technique works for arbitrary assertions, the only assertions contained in these programs are assertions of the form `x != null` automatically inserted by HAVOC before each pointer dereference `*x`. For each procedure, we unroll the loops twice. This allows us to focus on the (still substantial) subset of warnings from program verifiers that do not arise from analysis imprecision in handling loops.

***Experiments.*** We performed experiments with the goal of evaluating both the reduction in warnings (Section 5.1.1) and the precision/completeness trade offs (Section 5.1.2) induced by the abstractions and clause pruning. Finally, we evaluated the usability of ACSPEC on larger programs by running the tool on over 1M lines of Windows code (Section 5.1.3). We triaged the highest priority warnings and compiled statistics on the performance of the tool.

***Experimental setup.*** For each benchmark we checked for *SIB*s with the *Conc* configuration and abstract *SIB*s with the $A_0$, $A_1$, and $A_2$ configurations. *Cons* refers to the conservative verifier we

[2] `http://samate.nist.gov/SRD/testsuite.php`

[3] `http://sir.unl.edu/portal/index.html`

[4] `http://www.microsoft.com/whdc/devtools/ddk/default.mspx`

| Bench | Proc | Asrt | $Conc$ | k=3 | k=2 | k=1 | $A_1$ | k=3 | k=2 | k=1 | $A_2$ | k=3 | k=2 | k=1 | $Cons$ | TO |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CWE476 | 561 | 228 | 32 | 32 | 32 | 34 | 32 | 32 | 32 | 34 | 36 | 36 | 36 | 36 | 126 | 0 |
| CWE690 | 1865 | 1009 | 36 | 36 | 36 | 60 | 54 | 54 | 54 | 60 | 105 | 105 | 105 | 105 | 348 | 0 |
| ansicon | 289 | 325 | 1 | 4 | 5 | 5 | 2 | 17 | 18 | 20 | 5 | 26 | 26 | 28 | 60 | 9 |
| space | 263 | 2144 | 2 | 94 | 131 | 150 | 6 | 27 | 18 | 243 | 9 | 22 | 28 | 30 | 313 | 39 |
| cancel | 9 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 2 | 2 | 1 |
| event | 7 | 4 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| firefly | 9 | 22 | 0 | 2 | 4 | 5 | 0 | 0 | 2 | 2 | 4 | 4 | 4 | 4 | 7 | 0 |
| moufilter | 7 | 25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 5 | 5 | 5 | 8 | 0 |
| vserial | 23 | 102 | 0 | 10 | 2 | 11 | 0 | 0 | 10 | 11 | 8 | 10 | 12 | 12 | 24 | 1 |
| Total | 3033 | 3868 | 71 | 178 | 210 | 266 | 94 | 130 | 134 | 372 | 172 | 208 | 216 | 222 | 889 | 50 |

**Figure 6.** Comparison of abstract configurations and clause pruning on small benchmarks. For each abstract configuration ($Conc$, $A_1$, $A_2$), we show results with no clause pruning ($k = \infty$) and with pruning from $k$ = 3 to 1."TO" denotes the number of timeouts.

use (BOOGIE). We omit results for $A_0$ because it performed the same as $A_2$ on all benchmarks we tried. Timeout was set to 10 seconds; we omit procedures that timed out in any configuration. Experiments were run on a Windows 7 desktop machine with a 2.66 GHz processor and 6 GB of RAM and a Windows 7 laptop with a 1.3 GHz processor and 4 GB of RAM.

## 5.1 Evaluation

### 5.1.1 Warning reduction

In our first experiment, we compared the the number of warnings reported by $Conc$, $A_1$, and $A_2$ abstract configurations both without clause pruning and with $k$-clause pruning for $k = 1$ to 3 (Figure 6). In $k$-clause pruning, we prune clauses with $> k$ literals out of specifications (§ 4.3). We observe that:

- Without clause pruning, all abstract configurations report many fewer warnings than $Cons$. Even the coarsest abstract configuration ($A_2$) reported at least 2X fewer alarms than the conservative verifier on almost all benchmarks.
- Clause pruning steadily increases the number of alarms reported as $k$ decreases. This effect is relatively stable across abstract configurations, but is much more dramatic for $Conc$ and $A_1$. Even with clause pruning, all configurations still report less than half as many alarms as $Cons$ on most benchmarks.

Interestingly, combining aggressive clause pruning with a coarser abstraction can sometimes result in a *lower* number of errors than a finer abstraction at the same level of clause pruning (for example, the firefly benchmark under 1-clause pruning in the $Conc$ and $A_1$ configurations). This is because abstractions remove predicates and consequently force the generation of simpler (and less disjunctive) specs. An example illustrates how this can occur:

```
L1: grid_ptr = malloc();
L2: if (grid_ptr == NULL) return;
L3: x = *key;
```

The weakest specification for $Conc$ ($\theta_{L1}$`.malloc.return == 0 || key != 0`) has a disjunction. However, $A_1$'s predicate vocabulary does not allow predicates from conditionals, so the angelic specification is the simpler `key != 0`. Though both specs prove the program correct without creating dead code, the former contains one disjunction, so it is pruned to **true** by 1-clause pruning and reports a warning.

### 5.1.2 Precision and completeness trade offs

Our next experiment investigated whether the additional alarms revealed by adding abstractions and/or clause pruning to the $Conc$ configuration were real bugs or false positives. We performed a complete classification of the error reports for $Cons$, $Conc$, and the abstract configurations for two benchmarks: CWE476 and CWE690. We chose these because they are from a NIST test suite for static analysis tools in which dereferences are labeled as safe or buggy (with 36% and 27% assertions buggy for CWE476 and CWE690 respectively). We omit the results for clause pruning because they were nearly identical to the results shown. Figure 7 shows the results. We can observe the following:

|  |  | $Conc$ |  |  | $A_1$ |  |  | $A_2$ |  |  | $Cons$ |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bnch | Asrt | C | FP | FN | C | FP | FN | C | FP | FN | C | FP | FN |
| CWE476 | 228 | 179 | 0 | 49 | 179 | 0 | 49 | 179 | 2 | 47 | 183 | 45 | 0 |
| CWE490 | 1009 | 775 | 0 | 234 | 793 | 0 | 216 | 844 | 0 | 165 | 931 | 78 | 0 |
| Total | 1237 | 954 | 0 | 283 | 972 | 0 | 265 | 1023 | 2 | 212 | 1114 | 123 | 0 |

**Figure 7.** Full classification of alarms for the SAMATE benchmark suite across abstract configurations with no clause pruning. We give the number of assertions classified correctly (C), number of false positives (FP), and number of false negatives (FN).

- Adding abstractions (such as $A_1$ and $A_2$) to $Conc$ allows us to report more real bugs than the concrete domain while barely increasing the number of false positives. The few false positives our abstract domains report are due to the "havoc return values" abstraction as shown in § 4.4.2 and § 4.4.3.
- Even the coarsest abstraction fails to report lots of real bugs. This is expected — our stated goal is to report a small number of high confidence bugs with a low false positive rate. Many false negatives occur because there is no (abstract) inconsistency when the procedures bodies are simple, but buggy (e.g. `void Foo(x) { *x = 1;}`). To catch such bugs, we plan to extend our current method to assert the weakest precondition of simple procedures at call sites.

### 5.1.3 Larger benchmarks

In this section, we evaluate ACSPEC on the set of core Windows benchmarks, measuring over a million lines. The results are shown[5] in Figure 8.

| Bench | Proc | Asrt | $Conc$ | $A_1$ | $A_2$ | $Cons$ | TO |
|---|---|---|---|---|---|---|---|
| Drv1 | 799 | 14402 | 2 | 5 | 44 | 399 | 401 |
| Drv2 | 1218 | 13843 | 0 | 0 | 61 | 766 | 370 |
| Drv3 | 67 | 692 | 0 | 0 | 1 | 63 | 18 |
| Drv4 | 401 | 3582 | 0 | 0 | 12 | 262 | 61 |
| Drv5 | 661 | 11250 | 0 | 0 | 31 | 353 | 294 |
| Drv6 | 490 | 7381 | 0 | 0 | 84 | 287 | 114 |
| Drv7 | 16753 | 172973 | 1 | 11 | 876 | 12596 | 1369 |
| Lib1 | 1158 | 10051 | 1 | 3 | 171 | 552 | 317 |
| Total | 21547 | 234174 | 4 | 19 | 1280 | 15278 | 2944 |

**Figure 8.** Comparison of abstract configurations on large benchmarks. "TO" denotes the number of timeouts.

We briefly comment on trends in the data:

- As on the smaller benchmarks, the abstract configurations provide a "knob" through which gradually more errors can be viewed by adding abstractions to $Conc$.
- With the possible exception of $A_2$, the abstract configurations show a number of warnings small enough that a willing user could feasibly examine them. As expected, the conservative verifier reports more warnings than most users would reasonably want to examine.

[5] Our tool did not analyze around 5000 procedures in Drv7 due to an out-of-memory error. The results do not include these procedures or their assertions.

We investigated most of the warnings reported by the $Conc$ and $A_1$ configurations. All of the reports we examined were false positives, which is perhaps not entirely surprising since the property we are testing for is very shallow and these benchmarks are old and well-tested Windows code.

Many false positives were due to expansion of macros (a problem also encountered by [11] [21]). One extensively used macro pattern defensively checks for `NULL` before dereferencing a field:

```
#define CheckFieldF(x, a) (x != NULL && x->f == a)
```

The short-circuiting semantics of "&&" causes us to view this as a conditional expression, which means that a code snippet such as `y = *x; if(CheckFieldF(x, a)) ...` is expanded to `y = *x; if (x != NULL) {...} else {L1: ...}`. $Conc$ flags this as a *SIB* since the location `L1` is unreachable for the specification `x != NULL`. We manually validated that `x` can never be never be `NULL` in each of these cases, which means that this check is too defensive, but not buggy. Another cause for the $Conc$ warnings is the presence of macros that encode an assertion in terms of **assert false** as follows:

```
#define SL_ASSERT(e)  if (!e) assert(false)
```

Our tool insists that the "then" branch of such code be reachable, although the user expects it to be reachable only when the assertion fails. Macro expansion corresponds to a form of inlining callees, and the heuristic of absence of dead code can sometimes be too strong interprocedurally [9] (due to defensive coding in the callees).

Most of the $A_1$ warnings happen either due to one of the reasons above, or due to the removal of correlations that appear to be true preconditions. A common pattern that we observe (names anonymized) is the following:

```
void Process(size_t mBufferLength, char *mBuffer, ..){ ...
1:  if (mBufferLength >= 0) { ...
2:     for (size_t i = 0; i < mBufferLength; ++i) { ...
3:         assert(mBuffer != null); mBuffer[i] = ..;
       }
     ....
    }
5:  if (mBuffer != null) { .... }
```

where the tool avoids the error in Line 3 during $Conc$ analysis by inferring the correct precondition: `mBufferLength >= 0` $\Longrightarrow$ `mBuffer != 0`, which does not create any dead code. However, $A_1$ results in a stronger specification `mBuffer != 0`, which creates dead code for the "else" branch for Line 5 and reveals a *SIB*.

A vast majority of the $A_2$ warnings are due to an overly conservative modeling of calls in HAVOC, where all fields (modeled as maps) are present in the set of modified globals for a call. Any nested dereference of a field `x->f->g` (modeled as `g[f[x]]`) after a call to (say) `bar` results in a warning since $A_2$ can't capture that `x->f != 0` (expressed as `f[x] != 0`) after the call to `bar`. On the other hand, both $Conc$ and $A_1$ can add a specification $\theta$`.bar.f[x] != 0` since the modified values have associated symbolic constants.

### 5.1.4 Performance

Figure 9 describes the performance of our tool on the large benchmark set. We do not report any statistics for procedures that the conservative verifier labels as correct. We note that: (1) As expected, $A_1$ and $A_2$ collect fewer predicates than $Conc$. (2) Interestingly, the number of clauses in the predicate cover is relatively stable across all three configurations even though $Conc$ runs noticeably slower than the other two domains.

| | $Conc$ | | | $A_1$ | | | $A_2$ | | |
|---|---|---|---|---|---|---|---|---|---|
| Bench | P | C | T | P | C | T | P | C | T |
| Drv1 | 3.5 | 1.1 | 2.7 | 2.0 | 1.0 | 2.2 | 1.8 | 0.9 | 2.1 |
| Drv2 | 4.5 | 1.1 | 2.0 | 2.4 | 1.1 | 1.3 | 2.2 | 1.0 | 1.2 |
| Drv3 | 3.6 | 1.3 | 2.3 | 2.8 | 1.3 | 1.6 | 2.6 | 1.3 | 1.6 |
| Drv4 | 4.1 | 1.6 | 2.7 | 2.6 | 1.5 | 1.9 | 2.5 | 1.5 | 1.9 |
| Drv5 | 3.6 | 1.0 | 2.3 | 1.7 | 0.9 | 1.4 | 1.4 | 0.8 | 1.4 |
| Drv6 | 5.3 | 1.6 | 2.8 | 2.8 | 1.5 | 1.7 | 2.8 | 1.3 | 1.9 |
| Drv7 | 3.3 | 1.3 | 1.1 | 2.5 | 1.2 | 0.8 | 2.4 | 1.1 | 0.8 |
| Lib1 | 6.1 | 1.5 | 2.3 | 3.1 | 1.2 | 1.3 | 2.0 | 1.0 | 1.2 |

**Figure 9.** Performance on large benchmarks expressed as per-procedure averages. P is av. predicates/procedure, C is av. clauses in the predicate cover/procedure, and T is av. time/procedure in seconds.

Finally, as we can see from Figure 8, we time out on 14% of the procedures that the conservative verifier does not label as correct[6]. We found that the sources of timeouts are mixed. Some of the larger procedures take more than 10 seconds to compute $Fail()$ and $Dead()$; others time out during the predicate cover generation, and a smaller number time out during the search for almost-correct specifications. In the latter case, we can at least answer the question about the existence of a *SIB* in the procedure. We believe that using the incremental Z3 interface will significantly reduce timeouts by preventing redundant VC generation.

## 6. Related work

Engler *et al.* introduced inconsistency detection [11] as a practical approach for finding bugs in real programs. They identify a set of templates for inconsistent programmer beliefs such as "`*<p> ... if (<p> != null)`" that can be matched to patterns in real code in order to find likely bugs. They rank occurrences of these patterns using statistical analysis and use this simple but powerful combination to discover hundreds of new bugs in open source software. They build on this work with techniques for suppressing false alarms stemming from "polluted" analysis results [17] and a combination of this approach with factor-based statistical techniques to automatically infer rich function specifications, such as which functions allocate and de-allocate memory [18].

Dillig *et al.* provide the first semantic formulation of inconsistency errors [9]. In their framework, an inconsistency is a discrepancy between two related "checks" (i.e., a dereference): an inconsistency exists when there are two checks on the same predicate such that one check always succeeds, but the other may fail. They formally distinguish inconsistency errors from *source-sink* errors which require directly tracking the flow of a bad value across the program. Finally, they provide a scheme for interprocedural inconsistency detection based on inlining failure summaries for procedures at their call sites.

The recent work of Tomb and Flanagan [21] is the closest to identifying a connection between weakest precondition and (concrete) inconsistency checking in terms of assertions. Their inconsistency detection works by splitting a procedure $pr$ with $c$ conditionals into $2c$ *wedge programs*, where each wedge blocks one of the two branches of a conditional statement. An inconsistency is reported if any wedge program fails on all inputs. Both of these works [9, 21] validate that numerous bugs can be found using semantic inconsistency checking.

Our approach is inspired by both of these formulations of inconsistency checking, although our goal is not limited to finding inconsistency bugs. There are several differences between our work and

[6] We ignore the 5000 procedures that were not analyzed due to the memory problem.

theirs. First, our formulation of inconsistency checking can be parameterized with an abstraction, which allows us to apply the idea to a more general class of errors. Second, we compute the almost-correct specifications for a procedure, which allows us to witness inconsistency bugs as the failures induced by these specifications.

Finally, there are subtle discrepancies in the definition of inconsistency due to differences in formalism. For example, [21] would not report an inconsistency error for the procedure `Foo` in § 4.4.2, whereas [9] would [21]. Our formulation agrees with [21] in that we would characterize this program as an abstract *SIB* rather than a concrete *SIB*. On the other hand, we would report a concrete *SIB* for the program `if` $(*)$ `then assert` $e$ `else assert` $\neg e$, since there are no inputs that satisfy both assertions. [21] would not flag this program since neither of the two wedge programs is guaranteed to fail. We believe that [9] would not report this as an error since neither assertion is guarded by a check that guarantees its success. Similar reasoning (due to the presence of non-determinism) shows that [21] will miss reporting a *SIB* for the example in Figure 1.

In addition to prior work on semantic inconsistency, there are other generic approaches to reducing false alarms in static analysis. *Necessary preconditions* [5] are conditions that are shared by all non-faulting execution traces. Even without abstraction, the preconditions inferred by our approach are (in general) incomparable to necessary preconditions. For the program `if (x) { assert x; } assert x`, the necessary precondition is `x`, but the almost-correct specification is `true`, and thus the necessary precondition is stronger. For the program `if (*) assert x`, the necessary precondition is `true`, and the almost-correct specification is `x`, and so the almost-correct specification is stronger. The work on *doomed program points* [15] reduces false alarms by looking for assertions that will fail for all possible inputs. Such assertions are a special case of our *SIB*s. Recent work by Dillig *et al.* [10] use *abductive inference* with user feedback to classify alarms. We believe that the two approaches are largely complementary. The framework in [10] mainly addresses the problem of false alarms resulting from analysis imprecision instead of imprecision from the environment. Our framework can aid this work by identifying the initial set of warnings to display to the user for further classification. On the other hand, the use of quantifier elimination for abduction can provide an alternate way to discover almost-correct specifications. The work on *interleaved bugs* [16] addresses the underspecified precondition problem for the specific case of checking concurrency bugs by using the "simpler" sequential executions as a filter. Our approach applies in the sequential setting and does not use any simpler behaviors as oracles.

## 7. Conclusions and future work

In this paper, we make the case for using semantic methods to make program verifiers less demonic in the face of uncertainty. We introduce the concepts of *abstract* semantic inconsistency bugs and almost-correct specifications, which allow program verifiers to avoid reporting a class of stupid false alarms without being overly angelic. We show that our framework can be instantiated to find the demonstrably useful class of semantic inconsistency bugs. Our preliminary results demonstrate that the technique can be useful in suppressing many obviously demonic environments and improving the quality of warnings.

There are several directions for extending this work: We would like to discover more interesting abstractions, possibly by using user feedback for guidance. Extending our current work to perform limited interprocedural analysis [9] by asserting failure preconditions at call sites will increase the scope of analysis and increase the set of abstract *SIB*s. Finally, we plan to conduct a user study to understand the quality of the alarms we report initially.

## References

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, LNCS, 2005.

[2] 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.

[3] 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.

[4] 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.

[5] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *VMCAI*, pages 128–148, 2013.

[6] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, 2008.

[7] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[8] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 1975.

[9] 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.

[10] 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.

[11] 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.

[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI'02)*, 2002.

[13] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Symposium on Principles of Programming Languages (POPL '01)*, pages 193–205. ACM, 2001.

[14] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV '97)*.

[15] 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.

[16] 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.

[17] 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.

[18] T. Kremenek, P. Twohey, G. Back, A. Y. Ng, and D. R. Engler. From uncertainty to belief: Inferring the specification within. In *OSDI*, 2006.

[19] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. Smt techniques for fast predicate abstraction. In *Computer Aided Verification (CAV '06)*, Lecture Notes in Computer Science, 2006.

[20] NIST SAMATE Benchmarks. `http://samate.nist.gov/SRD/testsuite.php`.

[21] A. Tomb and C. Flanagan. Detecting inconsistencies via universal reachability analysis. In *International Symposium on Software Testing and Analysis (ISSTA '12)*, 2012.