# Detecting Inconsistencies via Universal Reachability Analysis

Aaron Tomb
Galois, Inc., USA

Cormac Flanagan
University of California, Santa Cruz, USA

## ABSTRACT

Recent research has suggested that a large class of software bugs fall into the category of *inconsistencies*, or cases where two pieces of program code make incompatible assumptions. Existing approaches to inconsistency detection have used intentionally unsound techniques aimed at bug-finding rather than verification. We describe an inconsistency detection analysis that extends previous work and is based on the foundation of the weakest precondition calculus. On a closed program, this analysis can serve as a full verification technique, while in cases where some code is unknown, a theorem prover is incomplete, or specifications are incomplete, it can serve as bug finding technique with a low false-positive rate.

**Categories and Subject Descriptors** D.2.4 [*Software Engineering*]: Software/Program Verification—Assertion checkers, programming by contract

**General Terms** Verification, Reliability, Languages

**Keywords** Defect detection, weakest preconditions

## 1. INTRODUCTION

Software is almost guaranteed to contain bugs. Programmers inevitably have trouble holding in mind every detail of a program's components and their many potential interactions. The time pressures of software development often exacerbate this problem, leading to an initial, rapid push to get *something* written, followed by a clean-up phase in which the developers attempt to improve the original codebase. Tools to verify code and help find bugs can be valuable in this second phase, but many existing tools, especially those that require information about the program's intended behavior, prove to be of limited use.

The problem occurs because specifications added during typical software development often take the form of independent assertions scattered throughout a program, rather than a single, coherent and interrelated contract. When testing is the primary form of quality assurance, these assertions can work well as runtime checks. However, testing has low coverage when typical programs have an effectively infinite state space.

Static program analysis tools can address the problem of coverage through mathematical reasoning about *all possible* program executions. However, when automated verification techniques are applied naïvely to programs that were not designed with verification in mind, they typically yield large numbers of spurious warnings because they lack sufficient information, computational power, or algorithmic sophistication to prove the necessary verification conditions. These warnings tend to be unhelpful because they indicate only failure to prove the absence of defects, rather than successful proof of the presence of defects.

### 1.1 Inconsistencies

We show how partial specification information in the form of assertions can be useful for static defect detection without incurring large numbers of false positives and while remaining scalable. While the approach described will not detect all errors that other tools can uncover, it will detect an important class of errors we will refer to as *inconsistencies*, with a low false-positive rate. In addition, once specification information becomes sufficiently comprehensive, the same implementation can be used for full verification.

Broadly, an inconsistency is any case in which an operation at one point in a program depends on one assumption to ensure correct operation, a second operation at a different point depends on another assumption, and these two assumptions are in some way incompatible. This idea has probably existed in the minds of programmers for decades, but was popularized by Dawson Engler in 2001 [5]. The idea was later successfully applied by Dillig *et al.* to find potential null pointer dereferences [4].

Our analysis, which we call *universal reachability analysis*, is a novel approach to inconsistency detection based on applying the weakest precondition operator to several modified versions of an input program. For a given program fragment, containing $n$ two-way conditionals, we generate $2n$ *wedges*. Each wedge is a variant of the initial fragment in which a single conditional is constrained to take only one of its two possible paths. One key characteristic of this approach is that every program point is unconditionally executed in at least one wedge, unless it is unreachable.

Once wedges have been generated, our analysis checks that each wedge can, for at least one input, terminate without violating any assertion, using the weakest precondition calculus. We say that a statement contains *fatal code* if one of its wedges is non-terminable.

Our thesis is that the application of weakest precondition analysis to the problem of inconsistency detection has advantages not present in other approaches. In particular, the ability to use the same implementation for both bug-finding (with few false alarms) and verification, depending on the completeness of the existing specifications, is novel.

### 1.2 Example

As an example, consider the C implementation of binary search shown in Fig. 1, where the second recursive call has the third and fourth arguments in the wrong order.[1]

[1] It also, incidentally, has an integer overflow bug in the calculation of mid which our analysis would not detect.

*ISSTA*'12, July 15–20, 2012, Minneapolis, MN, USA
Copyright 2012 ACM 978-1-4503-1454-1/12/05 ...$10.00

```
/*@ requires beg ≤ end; */
int bsearch(int a[], int n, int beg, int end) {
    int mid = (beg + end) / 2;
    if(beg == end) return −1;
    if(a[mid] == n) {
        return mid;
    } else if(a[mid] < n) {
        return bsearch(a, n, mid, end);
    } else {
        return bsearch(a, n, mid, beg);
    }
}
```

**Figure 1: Example function containing an inconsistency.**

This function has four wedges, one for each of the predicates beg == end, a[mid] == n, a[mid] < n, and a[mid] > n.

Now let us extend the language with an `assume` statement. We use `assume` statements to describe what conditions are imposed by each wedge. The third wedge is then as follows, where we replicate the precondition as an `assume` statement.

```
assume(beg ≤ end);
int mid = (beg + end) / 2;
assume(beg ≠ end);
assume(a[mid] ≠ n);
assume(a[mid] ≥ n);
assume(a[mid] > n);
return bsearch(a, n, mid, beg);
```

This wedge is guaranteed to crash because the recursive call is always invoked with its third argument greater than its fourth argument, which violates the precondition of the callee.

## 1.3 Benefits of Universal Reachability

Universal reachability analysis has several attributes that make it a valuable addition to the arsenal of software defect-detection tools.

First, it has a low false-positive rate. In theory, universal reachability analysis should generate no false positives, and warn only about either possible crashes or dead code. In practice, engineering compromises lead to approximations in some cases. In addition, some true inconsistencies are not serious defects (*e.g.*, explicit checks for situations that can never occur).

In our experiments (Sec. 6), the rate of false positives that the analysis reported due to imprecision was fairly low, but the rate of unnecessary checks it reported was fairly high. We consider these unnecessary checks to be bad programming style, and fixing code to avoid them is feasible and likely advantageous.

A second benefit of universal reachability analysis is that it can be used on partial programs — even programs that cannot be executed. For example, it can analyze library code, separate from any specific client.

A third benefit is the freedom to adjust computational resource usage. At any point in the analysis, giving up is a sound (though incomplete) option. It is possible to simply ignore a particular wedge or a whole function, or to approximate proof obligations before attempting to solve them. In the presence of approximations or timeouts, verification may no longer be possible, but inconsistency detection still is, though some inconsistencies may be missed.

The ability to approximate is connected to universal reachability's most significant shortcoming, however. The analysis is not guaranteed to detect all possible assertion violations, only those that are guaranteed to occur as a consequence of passing through any single program point. While a test suite with full branch coverage would detect the same set of assertion violations, universal reachability has this defect detection power even if no tests exist.

A final point is that, from one perspective, universal reachability detects dead code. It may seem that detecting unreachable code is a minor benefit. However, dead code may point to subtle failures in other areas, and therefore some programming guidelines, such as parts of the DO178B standard used for software in airborne systems [11], require that all code be reachable, and that both paths of any conditional be meaningful and executable.

## 1.4 Contributions

This paper makes two key contributions to the state of the art in automated software defect detection:

- A novel formalization of the notion of inconsistency, called universal reachability analysis, built on the foundation of the weakest precondition calculus.
- A collection of experimental results showing the ability of this new analysis to detect bugs with a low false positive rate using reasonable computational resources.

# 2. LANGUAGE

Fig. 2 shows the syntax of a language we call $GC$ that we use to describe universal reachability. $GC$ is a first-order imperative language similar to those languages considered by Dijkstra, Floyd, and Hoare [3, 9, 10].

The language includes *variables*, *values*, and *stores*. Variables (written $x$, $y$, or $z$) are unique identifiers drawn from an infinite supply. Values are left unspecified, and indicated by the metavariable $v$. A store $\theta$ is a functional mapping used to describe the current values of program variables in a particular concrete execution. We use the standard functional update operation to model changes to the store.

*Predicates* (written $P$, $Q$, $N$, $X$, or $W$) are functional mappings from stores to Boolean truth values. A predicate in our context is a characteristic function that describes the subset of all possible stores that satisfy a particular condition.

A program is a collection of functions, each of which consists of a name, a precondition $P$, a normal postcondition $N$, an exceptional postcondition $X$, and a body $s$. The annotations $P$, $N$, and $X$ represent the claim that when invoked in a state satisfying $P$, the body $s$ will evaluate either to `skip` in a state satisfying $N$ or to `raise` in a state satisfying $X$. We use the global variable $p$ in the following to refer to the program under consideration, to avoid including it as an extra parameter to every definition. We will then say $(\texttt{fun}\ f()\ P\ \{\ s\ \}\ N\ X) \in p$ to refer to definitions within the current program.

Intermediate specifications within function bodies are encoded using static assumptions (written `assume` $P$) and dynamic assertions (written `assert` $P$). An assumption trims the execution state space to that in which $P$ is true, providing a way to encode conditional execution. An assertion is a runtime check, and the program will *go wrong* if $P$ is not true during execution. For either an assertion or assumption statement, $P$ will be true after the execution of the statement, if execution proceeds. We also have a generalization to assignment: the $\texttt{havoc}(x)$ instruction assigns an arbitrary, unknown value to $x$. The primary purpose of this instruction is to help support our analysis of loops and procedure calls.

Alternative paths of execution are encoded using the nondeterministic choice operator (written $s_1 \ \square \ s_2$). This operator chooses arbitrarily to execute either $s_1$ or $s_2$. The "if" construct

$$
\begin{array}{rlll}
\textit{Program}: & p & ::= & d^* \\
\textit{Function}: & d & ::= & \texttt{fun}\, f()\ P\ \{\ s\ \}\ N\ X \\
\textit{Instruction}: & i & ::= & x := e \\
 & & \mid & \texttt{assert}\ P \\
 & & \mid & \texttt{assume}\ P \\
 & & \mid & f() \\
 & & \mid & \texttt{havoc}(x) \\
\textit{Statement}: & s & ::= & i \\
 & & \mid & s\ \square\ s \\
 & & \mid & \texttt{skip} \\
 & & \mid & s\ ;\ s \\
 & & \mid & \texttt{raise} \\
 & & \mid & s\,!\,s \\
 & & \mid & s^* \\
\textit{Variable}: & x, y, z & & \\
\textit{Value}: & v & ::= & \ldots \\
\textit{Store}: & \theta & : & \textit{Variable} \to \textit{Value} \\
\textit{Expression}: & e & : & \textit{Store} \to \textit{Value} \\
\textit{Predicate}: & P, N, X, W & : & \textit{Store} \to \textit{Boolean} \\
\textit{Context}: & C & ::= & \bullet \\
 & & \mid & C\ ;\ s \mid s\ ;\ C \\
 & & \mid & C\,!\,s \mid s\,!\,C \\
\textit{Evaluation Context}: & & & \\
 & E & ::= & \bullet \mid E\ ;\ s \mid E\,!\,s
\end{array}
$$

**Figure 2: Syntax for $GC$**

found in most languages can be encoded using a combination of assumption and non-deterministic choice.

$$\texttt{if}\ P\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \stackrel{\text{def}}{=} (\texttt{assume}\ P\ ;\ s_1)\ \square\ (\texttt{assume}\ \neg P\ ;\ s_2)$$

Iteration is encoded using the loop operator (written $s^*$). This operator executes the statement $s$ an arbitrary number of times. Loops that terminate when a particular condition (or its negation) occur, as found in most programming languages, can be encoded using assumptions.

$$\texttt{while}\ P\ \texttt{do}\ s \stackrel{\text{def}}{=} (\texttt{assume}\ P\ ;\ s)^*\ ;\ \texttt{assume}\ \neg P$$

We include procedure definitions and procedure calls so that we can reason about procedure specifications. For simplicity in the formal semantics, all variables are global; there are no local variables or arguments. Locals and arguments (and the rest of the C language) are supported in the implementation described in Sec. 5, however.

Finally, $GC$ supports exceptions and recovery from exceptions. The `raise` statement throws an exception. The statement $s_1\ !\ s_2$ executes $s_2$ if $s_1$ evaluates to `raise`. Note that exceptions are not the same as assertion failures. We consider any failed assertion to indicate a bug in the program, whereas a program that reduces to `raise` is acceptable.

Programs can terminate (or fail to terminate) in several ways. If application of the evaluation rules yields a state where the statement is `skip`, the program has *terminated normally*. If the statement is `raise`, the program has *terminated exceptionally*. the evaluation rules will not allow execution to proceed in either of these cases.

A program is *stuck* if it is in any state where the statement is neither `raise` nor `skip`, and none of the evaluation rules apply. A stuck program can arise in any state where the current statement is an assertion or assumption, and the associated predicate is not true when applied to the current store. If the program is stuck when the current statement is an assertion, we call it an *assertion failure*. Finally, a program *diverges* if there is no finite sequence of evaluation rule applications that will yield an assertion failure, normal termination, or exceptional termination. Divergence includes stuck programs in which the current statement is an assumption.

In some cases, it can be useful to describe statements within the context of a larger, enclosing statement. For this, we define two forms of statement context in Fig. 2, normal contexts, $C$, and evaluation contexts, $E$, both of which contain a hole where a substatement can fit. A normal context can be empty ($\bullet$), can refer to either substatement of a sequencing statement ($C\ ;\ s$ or $s\ ;\ C$) or of an exception handler ($C\ !\ s$ or $s\ !\ C$). Evaluation contexts are the subset of normal contexts in which the statement filling the hole can be immediately evaluated.

Finally, the $\textit{targets}(s)$ function returns the set of variables assigned by $s$ (where the targets of a call are the targets of the callee's body). It is used in conjunction with $\texttt{havoc}(x)$ to obscure the values of the variables assigned by $s$.

## 3. UNIVERSAL REACHABILITY ANALYSIS

Intuitively, a program is universally reachable if every statement (and sub-statement) within it can possibly be executed, starting from at least one initial state. To reason about this property, we first introduce a formal notion of evaluation.

**Definition 1** (Evaluation). *The notation $\theta, s \to \theta', s'$ means that, starting in a state $\theta$, statement $s$ can evaluate in one step to $s'$, yielding a new store $\theta'$. The notation $\theta, s \to^* \theta', s'$ is the reflexive, transitive closure, indicating that $\theta, s$ can evaluate to $\theta', s'$ in zero or more steps. The full definition of this relation is available in the first author's dissertation [17].*

Next, we consider the conditions in which it is possible for a sub-statement $s'$ of $s$ be evaluated during the evaluation of $s$. For this to happen, it must be possible for $s'$ to find its way into an evaluation context. If this is impossible then $s'$ is dead code.

**Definition 2** (Dead Code). *A sub-statement $s'$ of $s$ is* dead *in $s$ if there do not exist stores $\theta$, $\theta'$ and an evaluation context $E$ such that $\theta, s \to^* \theta', E[s']$.*

The notion of *fatal code* generalizes dead code. A substatement $s'$ of $s$ is fatal if an execution of $s$ cannot execute $s'$ and then terminate properly, either in `skip` or `raise`.

**Definition 3** (Fatal Code). *A sub-statement $s'$ of $s$ is* fatal *if there do not exist stores $\theta$, $\theta'$, $\theta''$, an evaluation context $E$, and a statement $s'' \in \{\texttt{skip}, \texttt{raise}\}$ such that*

$$\theta, s \to^* \theta', E[s'] \to^* \theta'', s''$$

Finally, we extend this notion to consider the case where none of the sub-statements of a given statement are fatal code.

**Definition 4** (Universally Reachable). *A statement $s$ is* universally reachable *if no sub-statement in $s$ is fatal.*

### 3.1 Wedge Calculation

We compute universal reachability by rewriting each function body into several versions, or *wedges*. Informally, a wedge is a modified form of a statement in which one branch of a chosen choice statement is forced to be taken. This modification requires also that any enclosing choice statements are forced to take the path that leads to the chosen choice statement. For every choice statement there are two wedges. Note that wedges are not the same as paths: the number of wedges in a program grows linearly with the number of choice statements in a program while the number of paths grows exponentially in the number of choice statements.

We formalize the preceding intuitive notion of wedges with the function $\textit{wedges}(C, s)$ shown in Fig. 3. This function computes all wedges, $s_1 \ldots s_n$ of $s$ and inserts each of them into the enclosing context $C$, yielding the collection of wedges $C[s_1] \ldots C[s_n]$. Thus, $\mathit{wedges}(\bullet, s)$ computes all wedges of $s$.

$$
\begin{aligned}
\mathit{wedges} &: \mathit{Context} \times \mathit{Statement} \to 2^{\mathit{Statement}} \\
\mathit{wedges}(C, i) &= \{C[i]\} \\
\mathit{wedges}(C, \texttt{skip}) &= \{C[\texttt{skip}]\} \\
\mathit{wedges}(C, \texttt{raise}) &= \{C[\texttt{raise}]\} \\
\mathit{wedges}(C, s_1 \ ; \ s_2) &= \mathit{wedges}(C[\bullet \ ; \ s_2], s_1) \\
&\quad \cup \mathit{wedges}(C[(s_1 \ ! \ \bot) \ ; \ \bullet], s_2) \\
\mathit{wedges}(C, s_1 \ ! \ s_2) &= \mathit{wedges}(C[\bullet \ ! \ s_2], s_1) \\
&\quad \cup \mathit{wedges}(C[(s_1 \ ; \ \bot) \ ! \ \bullet], s_2) \\
\mathit{wedges}(C, s_1 \ \Box \ s_2) &= \mathit{wedges}(C, s_1) \cup \mathit{wedges}(C, s_2) \\
\mathit{wedges}(C, s^*) &= \mathit{wedges}(C[(s^* \ ! \ \bot) \ ; \ \bullet \ ; \ s^*], s)
\end{aligned}
$$

**Figure 3: Wedge Computation**

To calculate the wedges of an instruction, `skip`, or `raise`, in a context $C$, we simply insert the statement into $C$.

To calculate the wedges of a sequence, $s_1 \ ; \ s_2$, we calculate the wedges of $s_1$ in extended context where it is followed by $s_2$. To calculate the wedges of $s_2$, however, we use the context $C[s_1 \ ! \ \bot \ ; \ \bullet]$ to ensure that $s_1$ does not raise an exception; if it does, the wedge will diverge, since $\bot$ represents the divergent computation `assume false`.

To calculate the wedges of an exception handler, $s_1 \ ! \ s_2$, we use a similar approach. In this case, we need to account for the fact that $s_1$ may evaluate to `skip`, in which case the exception handler will be skipped.

To calculate the wedges of a choice statement, $s_1 \ \Box \ s_2$, we calculate the wedges of each of its branches in the same context as the entire statement. This has the effect of discarding $s_2$ in one subset of wedges, and discarding $s_1$ in the other subset of wedges.

To calculate wedges in the presence of loops, we need to ensure that the selection of a particular choice branch occurs on an *arbitrary* loop iteration. Otherwise, the selection of a particular choice may introduce artificial inconsistencies. Often, loop bodies contain conditional branches that may be possible only on certain loop iterations, such as the first iteration or alternating iterations.

To understand how we perform wedge calculation at an arbitrary iteration, consider a loop $s^*$ with body $s$. To ensure that a wedge of $s$ occurs in an arbitrary iteration of $s^*$, we first calculate the wedges of $s$. For each of these wedges, $s'$, we add an arbitrary number of loop iterations occurring both before and after $s'$, and require that the preceding iterations terminate normally:

$$(s^* \ ! \ \bot) \ ; \ s' \ ; \ s^*$$

After computing all wedges of a function body $s$, we next ensure that each wedge is *terminable*, that is, it can be fully reduced to `skip` or `raise`.

**Definition 5** (Terminable). *A statement $s$ is* terminable *if there exist stores $\theta$, $\theta'$, and a statement $s' \in \{\texttt{skip}, \texttt{raise}\}$ such that*

$$\theta, s \to^* \theta', s'$$

Though similar in style to universal reachability, this property is easier to check. In particular, it does not refer to sub-statements, but only to the overall statement in question.

Our key theorem is then that we can use terminability of the wedges of a statement to decide whether the original statement is universally reachable.

**Theorem 1.** *The following are equivalent for any statement $s$:*

1. *$s$ is universally reachable*
2. *$\forall s' \in \mathit{wedges}(\bullet, s)$, $s'$ is terminable*

*Proof.* By induction on the structure of $s$. □

## 3.2 Termination Analysis via Weakest Preconditions

It now remains to compute whether or not each wedge of a statement is terminable. We decide the terminability of each wedge with a combination of a weakest precondition calculation and an automated decision procedure.

The weakest precondition operation we use for termination analysis (Fig. 4) differs slightly from the traditional definition. We add a third predicate argument, $W$, representing the postcondition for the case where the statement fails an assertion (or *goes wrong*), and modify the interpretation of assertions to take this extra postcondition into account.

$\mathit{wp}$ : *Statement × Predicate × Predicate × Predicate → Predicate*

| $s$ | $\mathit{wp}(s, N, X, W)$ |
|---|---|
| $x := e$ | $N[x := e]$ |
| $\texttt{havoc}(x)$ | $\forall x'. N[x := x']$ |
| `skip` | $N$ |
| `assert` $P$ | $(P \wedge N) \vee (\neg P \wedge W)$ |
| `assume` $P$ | $P \Rightarrow N$ |
| $s_1 \ ; \ s_2$ | $\mathit{wp}(s_1, \mathit{wp}(s_2, N, X, W), X, W)$ |
| $s_1 \ \Box \ s_2$ | $\mathit{wp}(s_1, N, X, W) \wedge \mathit{wp}(s_2, N, X, W)$ |
| `raise` | $X$ |
| $s_1 \ ! \ s_2$ | $\mathit{wp}(s_1, N, \mathit{wp}(s_2, N, X, W), W)$ |
| $s^*$ | $\mathit{lfp}(\lambda P. \ N \wedge \mathit{wp}(s, P, X, W))$ |
| $f()$ | $\mathit{wp}(s', N, X, W)$<br>if `fun` $f() \ P \ \{ \ s \ \} \ N' \ X' \in p$<br>and $s' = \texttt{assert} \ P \ ; \ \texttt{havoc}(\mathit{targets}(s));$<br>`assume` $N' \ \Box$ (`assume` $X'$ ; `raise`) |

**Figure 4: Weakest Preconditions for Universal Reachability**

Intuitively, $W$ describes the permitted states for an assertion violation. If $W$ is `false` then the treatment of assertions matches that in the traditional definition of weakest preconditions. However, if $W$ is `true`, $\mathit{wp}(\texttt{assert}\ P, N, X, W)$ simplifies to $P \Rightarrow N$, and treats assertions with the traditional semantics for assumptions.

If we treat assertions with the semantics of assumptions, we cannot prove that a statement will fail an assertion, but we can prove that a statement will either fail an assertion or an assumption. An instance of this situation indicates either an assertion that will always fail or a branch that cannot be taken.

While either of these cases is fairly uninteresting when considering an entire function body, they are more likely to indicate interesting bugs when considering wedges. While a function that always fails during execution would almost certainly be discovered during testing, it may be more difficult to discover a particular branch that will always lead to failure, especially when the conditions that lead to the branch being taken are complex and hard to induce on demand. A significant fraction of the code in typical software systems is devoted to handling errors, and this code is often the least well-tested portion of a system, depending heavily on details of the execution environment [2].

Next, to decide the terminability of a statement $s$, we compute the predicate $\mathit{wp}(s, \texttt{false}, \texttt{false}, \texttt{true})$. We call the result of this particular weakest precondition calculation a *failure condition*. A failure condition which is always true indicates fatal code.

**Theorem 2.** *If $wp(s, \texttt{false}, \texttt{false}, \texttt{true})$ is valid and $s$ contains no call instructions then $s$ is not terminable.*

*Proof.* A direct consequence of Theorem 2. □

The proof of the above theorem relies in part on the following standard result that the weakest precondition calculation correctly reflects the underlying operational semantics. Here, we assume the absence of function calls, since function specifications generally only overapproximate the actual behavior of the function body.

**Theorem 3.** *If $wp(s, N, X, W) = P$ and $s$ contains no call instructions and there exist stores $\theta$ and $\theta'$ and a statement $s'$ such that $\theta, s \to^* \theta', s'$, where $s' \in \{\texttt{skip}, \texttt{raise}\}$, and $\theta$ satisfies $P$ then one of the following cases holds:*

1. *$s' = \texttt{skip}$ and $\theta'$ satisfies $N$*
2. *$s' = \texttt{raise}$ and $\theta'$ satisfies $X$*

*Proof.* By induction on the structure of $s$. □

We can now combine the previous three theorems to arrive at the following primary result that states that an algorithmic approach to detecting fatal code is sound.

**Theorem 4.** *If a statement $s$ contains no call instructions and there exists an $s' \in wedges(\bullet, s)$ such that $wp(s', \texttt{false}, \texttt{false}, \texttt{true})$ is valid then $s$ is not universally reachable.*

*Proof.* A straightforward consequence of Theorems 1 and 2. □

To determine validity automatically, we depend on a decision procedure that is sound (though not necessarily complete).

## 3.3 Verification

Although we have presented our analysis as a defect-detection mechanism so far, one advantage of using an approach based on weakest preconditions is that it generalizes to full verification when sufficient function specifications are available.

For example, if the specifications present in a program are sufficiently complete to prove $P \Rightarrow wp(s, N, X, \texttt{false})$ is valid for a function $\texttt{fun}\ f()\ P\ \{\ s\ \}\ N\ X$, then we can guarantee that the function meets its specification. When we cannot verify this strong correctness property, universal reachability analysis can still determine whether the function is guaranteed to violate the known components of its specification.

## 3.4 Loops and Approximation

Sec. 3.2 provided a semantics for loops that depends on a fixpoint construct in the logic:

$$wp(s^*, N, X, W) \;=\; \mathit{lfp}(\lambda P.\ N \wedge wp(s, P, X, W))$$

This is a precise encoding; the theorems from the previous sections all hold. However, existing automated decision procedures do not typically support fixpoint operations. Since we want to support automatic defect detection, we need a more tractable approximation.

We begin by defining an approximation relation over statements that allows us to say when the behavior of one statement overapproximates the behavior of another.

**Definition 6** (Statement Approximation)**.** *A statement $s$ overapproximates a statement $s'$, written $s \sqsupseteq s'$, if and only if for all predicates $N$, $X$, and $W$, $wp(s, N, X, W) \Rightarrow wp(s', N, X, W)$.*

Note that if $s$ overapproximates a wedge $s'$ and $s$ is not terminable (*i.e.*, $wp(s, \texttt{false}, \texttt{false}, \texttt{true})$ is valid), then $s'$ is also not terminable (*i.e.*, $wp(s', \texttt{false}, \texttt{false}, \texttt{true})$ is therefore also valid). Here we can safely overapproximate wedges without introducing false inconsistencies. Moreover, the notion of overapproximation is closed under contexts: if $s \sqsupseteq s'$ then $C[s] \sqsupseteq C[s']$.

Given a notion of statement approximation, we can then rewrite $s^*$ with an approximate version that guarantees that we get no false positives. One common loop approximation is to "unroll" some small number of times. For example, unrolling $s^*$ two times:

$$\texttt{skip} \;\Box\; s \;\Box\; (s; s)$$

In the context of other analyses, unrolling is often a reasonable approximation. However, it is an *under-approximation*, and therefore can lead to false positives in universal reachability analysis.

$$s^* \sqsupseteq \texttt{skip} \;\Box\; s \;\Box\; (s; s)$$

Intuitively, the reason unrolling a finite number of iterations can lead to false positives is that the body of the loop may contain code that executes, for example, only on the fifth iteration. Our preferred approach, with no false positives, is use $\texttt{havoc}(\mathit{targets}(s))$ to replace all store elements updated by the loop body $s$ with arbitrary values, obscuring which iteration is occurring. Unrolling the body once, and inserting $\texttt{havoc}(\mathit{targets}(s))$, we have:

$$\texttt{skip} \;\Box\; (\texttt{havoc}(\mathit{targets}(s))\ ;\ s) \sqsupseteq s^*$$

We can also unroll more than once:

$$\left(\begin{array}{ll} & \texttt{skip} \\ \Box & \texttt{havoc}(\mathit{targets}(s)); s \\ \Box & \texttt{havoc}(\mathit{targets}(s)); s; \\ & \texttt{havoc}(\mathit{targets}(s)); s \end{array}\right) \quad \sqsupseteq s^*$$

but this does not buy us anything. The weakest precondition of the loop unrolled twice is the same as the weakest precondition when unrolled only once, due to the havoc statements.

One disadvantage of havoc statements is that they may obscure too much information, leading to false negatives. Ideally we would use a loop invariant, $I$, to describe what condition is true on every loop iteration:

$$\begin{array}{ll} & \texttt{assume}\ I \\ \Box & \\ & \texttt{havoc}(\mathit{targets}(s)); \\ & \texttt{assume}\ I; \\ & s; \\ & \texttt{assume}\ I \end{array}$$

If the loop invariant $I$ is too weak (*e.g.*, $\texttt{true}$), then we may miss instances of fatal code, but we will get no false positives. If the loop invariant is too strong (*e.g.*, $\texttt{false}$), then one of the assertions may fail, which can lead to incorrect reports of fatal code.

## 3.5 Procedures

In the presentation so far the weakest precondition function uses the provided specification of the target function when analyzing a function call. In the absence of complete procedure specifications, this approach could seem overly imprecise.

To address this imprecision we could inline target functions and perform wedge calculation across procedure boundaries. Unfortunately, this can result in large numbers of false positives. The problem arises because most procedures represent abstractions, intended to be used in a variety of conditions. In the context of any specific function call, only some subset of these conditions are likely to hold, and therefore some portions of the target function body will be dead code. If the function body were inlined, each branch in the target function that could not be taken in one specific context where it is called would be flagged as inconsistent.

The problem of inlining procedures arises when using macros in C, as well. Many macros are treated as small, inline procedures, and they often contain conditionals that will be forced to take a single path in the specific context where they occur. The results in Sec. 6 show that macros are the most significant cause of uninteresting inconsistencies when analyzing C code.

## 4. BOUNDARY CONDITIONS

The $wedges$ function presented in Sec. 3.1 creates one wedge for each alternative branch in the program. Each wedge represents the subset of the program's state space that is possible if the chosen branch is taken. We can then ask whether or not it is possible for that wedge to terminate successfully.

However, other subsets of the program state space can be interesting, as well. One particularly common type of error occurs when comparisons are "off by one" and therefore succeed on values that are either just below or just above the correct boundary. It turns out that our existing wedge calculation technique applies with few modifications to this situation.

```
void overflow(int n) {
    int a[n]; int i;
    for(i = 0; i ≤ n; i++) a[i] = 0;
}
```

**Figure 5: Simple Array Clearing Program**

Consider the program in Fig. 5 which clears the contents of an array (as well as one element outside of the array). One of the wedges of this program will include the following fragment, corresponding to the execution of the loop body while the loop condition is true:

```
assume(i ≤ n);
a[i] = 0;
```

For deterministic programming languages, we can associate a condition with every wedge. In the above case, the condition for the wedge described above is $i \leq n$.

If we suspect that errors are common on the boundary conditions of inequalities, we may want to consider the wedge that represents a smaller, more targeted subset of the state space. Consider a wedge identical to the one above, but with the condition replaced with i == n. If we check the terminability of this new wedge (assuming that array accesses are guarded with bounds checks), we will find that it will always fail, accessing an array element one past the end of the array.

Formally, we modify the definition of the wedge calculation function by extending the rule for choice statements in the case where the immediate sub-statements begin with assumptions. Fig. 6 shows the modified case for $wedges(C, s)$ function, along with a pair of auxiliary functions.

As we will see in Sec. 6.7, this approach is particularly good at detecting buffer overflows, but results in false positives in some circumstances. Consider, for instance, the following code that performs different operations for zero, positive and negative numbers.

```
if (x == 0) { ... } else if (x > 0) { ... } else { ... }
```

This program fragment has two choice statements, one with a branch for x == 0 and a branch for x ≠0, and one with a branch for x > 0 and a branch for x ≤0. Note, however, that the latter two branches operate in a context where the x ≠0 branch has been taken, so x ≠0 is necessarily true. Therefore, when we take the x ≤0 wedge and specialize it to x == 0, we get a contradiction that does not represent fatal code in the original program.

$$\begin{array}{rcl} wedges(C, s_1 \ \square \ s_2) & = & wedges(C, s_1) \cup wedges(C, s_2) \\ & \cup & bwedge(C, s_1) \cup bwedge(C, s_2) \end{array}$$

$$bwedge : \mathit{Context} \times \mathit{Statement} \rightarrow 2^{\mathit{Statement}}$$

$$bwedge(C, s) = \begin{cases} \{C[\texttt{assume } P'; s']\} & \text{if } s = \texttt{assume } P; s' \\ & \text{and } bcond(P) = P' \\ \emptyset & \text{otherwise} \end{cases}$$

$$\begin{array}{rcl} bcond & : & \mathit{Predicate} \rightarrow \mathit{Predicate} \\ bcond(e_1 < e_2) & = & (e_1 = e_2 - 1) \\ bcond(e_1 > e_2) & = & (e_1 = e_2 + 1) \\ bcond(e_1 \leq e_2) & = & (e_1 = e_2) \\ bcond(e_1 \geq e_2) & = & (e_1 = e_2) \end{array}$$

**Figure 6: Boundary Condition Wedge Calculation**

## 5. IMPLEMENTATION

To validate the utility of universal reachability analysis, we developed a prototype implementation called Curate that can check for inconsistencies in C programs.

*Architecture* Curate begins by transforming C into an unstructured form of $GC$. Once the original C source has been translated into the intermediate language, we eliminate loops and insert havoc instructions. We then translate the loop-free function bodies into passive form [8]. Finally, we calculate the wedges of each function body according to a variant of the wedge calculation algorithm from Sec. 3.1 modified to work on unstructured programs, calculate the weakest precondition of each wedge and use Yices [12] to check the validity of this precondition.

*Memory Model* We treat the entire heap of the program as a single array, mapping unbounded integers to unbounded integers. Local variables become variables in the syntax of the intermediate representation, unless their address is taken, in which case we introduce a variable representing the address of the original variable, and use the address variable to index into the heap array. We have not noticed false positives caused by this approximation in our experimental results, but it may lead to false negatives.

*Loops* Loops introduce back edges in the control flow graph. We remove back edges and insert a havoc instruction at each loop head. This havoc instruction overwrites any variable written in the loop body. We calculate the loop body by determining the set of instructions that are reachable from the target of the back edge and that can reach the source of the back edge. This approach correctly obscures which iteration is currently under analysis.

*Procedures* The current Curate implementation performs intraprocedural analysis. When encountering call, the target procedure is assumed to have precondition `true`, and to potentially modify any element of the heap.

## 6. EXPERIMENTAL RESULTS

We compared Curate with the Saturn null pointer analysis [4] and the Clang Analyzer [15]. The results show that Curate runs in less time than the comparable Saturn null pointer analysis, while usually finding a larger number of inconsistencies, although Saturn does discover one class of inconsistency that Curate does not. The Clang Analyzer runs much more quickly than either Curate or Saturn, but discovers primarily shallow bugs, most of which will not lead to run-time errors. The warnings given by Clang only overlap with either Saturn or Curate in a handful of cases.

## 6.1 Benchmarks

We chose several programs on which to evaluate universal reachability. We included our chosen benchmarks from Dillig *et al.* [4] our collection: MPlayer 1.0pre8, OpenSSL 0.9.8b, Samba 3.0.23b, OpenSSH 4.3p2, and Sendmail 8.13.8. We skipped the Pine and Linux benchmarks of Dillig *et al.* because they presented parsing or type-checking difficulties to one or more of the tools included in our comparison.

Next we include a set of widely-used programs chosen independently. Some of these are known to have particular bugs. These benchmarks include BC 1.06, SpiderMonkey 1.70, NCompress 4.2.4, PCC 0.9.9, Squid 2.3STABLE1, and LibTIFF 3.7.0.

Finally, we include a set of small programs published by the SAMATE project [14] with the purpose of benchmarking automated defect-detection tools. This collection consists of more than 1500 simple C files that contain instances of common security bugs.

## 6.2 Experimental Setup

Before running the benchmark programs through each of the analysis tools, we set up an experimental framework to ensure that we could reasonably compare the results between benchmarks and between tools.

*System Configuration* We performed all experimental evaluation on a workstation running Mac OS X 10.6.5 with 8GB of RAM and a 4-core 2.5GHz Intel i5 processor.

*Pre-processing* We pass each program through the C preprocessor (CPP) once, and use the preprocessed version as the input to each of the analysis programs.

*Exit Functions* One critical consideration for both Saturn's null pointer analysis and universal reachability analysis is the detection of exit functions. The implementation of the `assert` function in C often consists of a conditional that aborts the program if the assertion evaluates to `false`, and continues execution otherwise. Knowing that the function call terminates the program is essential to avoiding false positives for both analyses.

Saturn includes an analysis to detect exit functions, and the result of this analysis, if available, is automatically used by the null pointer analysis. Therefore, we ran Saturn's exit function analysis on all of the benchmarks in advance, and used its results as input to Curate, as well.

*Time Limits* We ran both Saturn and Curate with a timeout value of 120 seconds.

*Source Code Changes* Because each tool uses a different front end to parse and type-check the C source code, we needed to make some minor modifications to the source code, after preprocessing, so that all tools could successfully load it. The modifications we made included adding function prototypes, removing complex static global initializers, limiting each benchmark to a single main function, and other minor changes to allow all the code to be processed by all of the tools.

## 6.3 Warning Categories

We classified each warning produced by any of the three tools into one of the following categories. The first collection of categories represents likely bugs, or at least awkward code. The second collection represents various types of dead code that can be detected by universal reachability analysis. The third represents false positives, largely caused by engineering tradeoffs in each of the tools. For each category, we give an abbreviation used in the tables of experimental results. For a few cases, we were not able to determine the cause of the inconsistency warning (typically due to the complexity of the associated source code). We place these few warnings in the "Unknown" category.

For the Samba benchmark, we give only the total number of warnings produced by each tool, as the combined number of warnings produced by the three tools was prohibitively high to allow for manual categorization.

### 6.3.1 True Errors

We refer to warnings that indicate bugs or awkward programming practices as *true* errors, of the following types.

*Bug-Causing Inconsistency (Inc.)* The most interesting inconsistency warnings indicate cases where some path through the function under analysis could lead to either an assertion failure or an incorrect result. This category includes any indication that an assertion failure is certain within a normal wedge (as opposed to a boundary condition wedge, as described next), even if the rest of the program never calls the function under analysis with arguments that would lead to a crash. It also includes cases where an assertion failure may not occur, but where the code will clearly operate incorrectly. When the Clang Analyzer discovers a potential null pointer dereference that coincides with one discovered by either Saturn or Curate, we include it in this category.

*Boundary Condition Inconsistency (Bound.)* If Curate discovers a potential assertion failure using boundary condition specialization, described in Sec. 4, it falls into this category.

*Programmer Confusion (Confus.)* Some cases of dead code detected by Curate or Saturn will not lead to assertion failures, but clearly indicate confusion or misunderstanding on the part of the author of the function under analysis. This category includes cases where dead code exists because a feature has not been completely implemented, and therefore parts of the implementation are temporarily unreachable.

*Other Tool Warning (Other)* Saturn's null pointer analysis includes a number of heuristics that fall outside of the scope of universal reachability analysis. In addition, it performs an interprocedural analysis, while Curate is intra-procedural. The Clang Analyzer also produces warnings for a number of problems that are outside of the scope of our comparison.

### 6.3.2 Dead Code

Some cases of inconsistency indicate dead code that is intentional or unavoidable, such as that which arises in code intended to run on multiple platforms.

*Trivial Impossible Cases (Trivial)* Some code, including infinite loops and nested conditionals without a final `else` clause, include branches with necessarily unsatisfiable conditions.

*Configuration Dead Code (Config.)* Some dead code occurs as a result of comparisons between values that are constant at compile time but that may vary across system configurations.

*Macro Instantiation (Macro)* Macros, like functions, typically represent abstractions and are written to apply in a wide variety of situations using conditional statements. In any particular context, some wedges of the expansion of a macro may contain dead code, and therefore lead to inconsistency warnings.

### 6.3.3 False Positives

Finally, some warnings reported by our tool are false positives in that they do not indicate fatal code. These occur due to heuristics, engineering compromises, or bugs in the implementation.

*Boundary Condition False Positive (Bound.)* As described in Sec. 4, the boundary condition extension to our analysis can result in false positives when a specialized wedge condition implies a condition already covered by an earlier branch.

*Unsupported (Uns.)* Some warnings occur as the result of explicitly unsupported language features. In our case, this includes multi-dimensional arrays, floating point arithmetic, nonlinear arithmetic, bit-level operations, overflow, and conversion between signed and unsigned values. These warnings would be avoidable with a more complete implementation. We also include in this category some cases where programmers intentionally stretch the boundaries of the semantics of C, but in ways that cannot cause runtime failures. Examples include taking the address of a field of a dereferenced null pointer (*e.g.*, &(((`struct` s ∗)NULL)→fld)) to calculate the offset of a field, and taking the address of an array element at an out-of-bounds index (*e.g.*, &(a[`sizeof`(a)])) but never dereferencing the result.

## 6.4 Small Examples

We first ran all three tools on a collection of small examples, each intended primarily as an illustration of a program inconsistency. In Tbl. 1 we show the number and category of warnings produced by each of the three tools.

As these examples were written to illustrate using universal reachability errors, Curate generates warnings about all of them (along with one case of trivial dead code in the binary search example). Saturn also warned about the inconsistency in the doubly-linked list example, but the Clang Analyzer was unable to detect any of the errors.

**Table 1: Warnings from Small Examples**

| Benchmark | Type | Curate | Saturn | Clang |
|---|---|---|---|---|
| `bsearch` | Inc. | 1 | 0 | 0 |
| | Trivial | 1 | 0 | 0 |
| `dblfree` | Inc. | 1 | 0 | 0 |
| `list` | Inc. | 1 | 1 | 0 |
| `oom` | Inc. | 1 | 0 | 0 |
| `overflow` | Bound. | 1 | 0 | 0 |
| `unstructured` | Inc. | 1 | 0 | 0 |
| `useafterfree` | Inc. | 1 | 0 | 0 |
| Total | All | 8 | 1 | 0 |

## 6.5 Saturn Benchmarks

Tbl. 2 shows the results of running each tool on a subset of the benchmarks from Saturn's null pointer analysis. In this table, we omit rows in which all entries are zero. For each benchmark, we list the number of preprocessed lines of code (PLOC) it contains. We write a hyphen in entries that are not supported by a tool.

Some of the entries for Saturn and Curate are of the form $S+U$, where $S$ is the number of warnings shared between the two tools, and $U$ is the number of unique warnings generated by the tool. The entries for Clang in the "Inc. Error" category are of the form $S + C + B$, where $S$ is the number shared with Saturn, $C$ is the number shared with Curate, and $B$ is the number shared with both.

**Table 2: Warnings from Saturn Benchmarks**

| | Type | Curate | Saturn | Clang |
|---|---|---|---|---|
| `MPlayer`, 2,552,948 PPLOC | | | | |
| True errors | Inc. | 6+27 | 6+13 | 2+3+0 |
| | Confus. | 1+39 | 1+8 | - |
| | Other | 0 | 58 | 1149 |
| Dead code | Trivial | 4 | 0 | - |
| | Config. | 1 | 0 | - |
| | Macro | 7 | 2 | - |
| False positives | Uns. | 14 | 1 | - |
| Unknown | Unk. | 3 | 0 | - |
| `openssh`, 907,011 PPLOC | | | | |
| True errors | Inc. | 1 | 0 | 0 |
| | Confus. | 1 | 0 | - |
| | Other | 0 | 4 | 72 |
| Dead code | Trivial | 1 | 0 | - |
| | Config. | 1 | 0 | - |
| | Macro | 0 | 1 | - |
| `openssl`, 3,084,124 PPLOC | | | | |
| True errors | Inc. | 4+5 | 4+2 | 2+3+2 |
| | Confus. | 36 | 6 | - |
| | Other | 0 | 138 | 517 |
| Dead code | Trivial | 6 | 0 | - |
| | Config. | 2 | 0 | - |
| | Macro | 159 | 0 | - |
| False positives | Uns. | 3 | 0 | - |
| Unknown | Unk. | 4 | 1 | 0 |
| `samba`, 10,765,507 PPLOC | | | | |
| Unknown | Unk. | 834 | 219 | 320 |
| `sendmail`, 498,941 PPLOC | | | | |
| True errors | Inc. | 0 | 0 | 0 |
| | Confus. | 4 | 0 | - |
| | Other | 0 | 23 | 151 |
| Dead code | Trivial | 1 | 0 | - |
| | Config. | 2 | 0 | - |
| | Macro | 3 | 0 | - |
| False positives | Uns. | 8 | 1 | - |
| Unknown | Unk. | 8 | 0 | - |

On this set of benchmarks, Curate detects some of the same inconsistencies as Saturn, as expected, along with many additional inconsistencies. The additional inconsistencies all fall outside of the domain of Saturn's analysis, either because they involve non-pointer variables, or because they rely on function preconditions or logical theories not included in Saturn's null pointer analysis.

Saturn also detects a few errors that Curate misses. Some of these are because Saturn's null pointer analysis includes a number of heuristics that fall outside of the domain of pure inconsistencies. These errors are listed in the "Other" category. However, Saturn also detects a few true inconsistencies that Curate misses: 13 for MPlayer and 2 for OpenSSL.

The inconsistencies detected by Saturn and not Curate occur for two reasons. The first is that the translation Curate uses for the C short-circuit operators (&&, ||), is semantically correct, but obscures inconsistencies from universal reachability analysis. We could have modified the translation of short-circuit operators to match that of the `if` statement, but we decided to maintain the existing approach because it illustrates an important disadvantage of universal reachability analysis: semantically-correct transformations that change the branching structure of a program can affect what inconsistencies are detected.

The second reason is that Saturn's analysis detects a class of inconsistencies that are not instances of fatal code but still often indicate real defects. We describe this class of inconsistencies further in Sec. 7.1.

Along with a slightly higher number of true defects, Curate also reports a higher number of arguably spurious warnings. Some are false positives resulting from engineering compromises, while others simply indicate dead code. Excluding false positives arising from macro expansion, however, the rate of spurious or uninteresting warnings is low, and could be made lower in a production-quality implementation.

## 6.6 Other Open Source Package Benchmarks

Tbl. 3 shows the results of the tools on our additional selection of open source benchmarks. In this collection of benchmark programs, we observe similar patterns to those in the previous set. The Clang Analyzer generates more warnings than either of the inconsistency detectors, except in the case of SpiderMonkey (`js`). Curate detects two significant inconsistencies missed by Saturn's null analysis (on SpiderMonkey), and Saturn detects two significant inconsistencies missed by Curate (on BC and PCC). Excluding macros, the number of spurious warnings is relatively low overall.

## 6.7 SAMATE Benchmarks and Boundary Condition Specialization

Finally, Tbl. 4 shows the warnings produced for the collection of small benchmarks from NIST's SAMATE team. Curate and Saturn both discover a number of true inconsistencies, but Curate discovers more. Curate is able to discover a number of cases of null pointer and freeing already-freed pointers. The null dereferences are also detected by both Saturn and the Clang Analyzer.

The Clang Analyzer also generates a large number of warnings outside of the realm of inconsistency analysis. Most of these indicate values written and then never read.

Beyond the results of normal inconsistency detection, the SAMATE collection particularly highlights the benefits of boundary condition specialization. We show the results of this specialization for all benchmarks in 5. The majority of the SAMATE test cases include buffer overflows of varying complexity. Almost all of them are detectable through intra-procedural analysis, and involve local arrays of known size. This is the condition in which boundary condition specialization excels, and we see that Curate can detect the majority of the buffer overflow defects. Although the buffers in these test cases are all of constant, known sizes, universal reachability analysis is able to detect overflows in more complex cases as long as the relationship between program variables and buffer sizes is known.

Overall, these results suggest that boundary condition specialization, and potentially other variants of universal reachability analysis, can detect interesting bugs, even though it leads to large numbers of false positives in some cases. Indeed, the largest collections of spurious warnings generated by Curate other than those caused by macro expansion result from boundary condition specialization. Boundary condition specialization discovered 2 real errors in the Saturn benchmarks and 96 false positives, and 7 real errors in the other open source benchmarks along with 15 false positives.

**Table 3: Warnings from Other Open Source Packages**

| | Type | Curate | Saturn | Clang |
|---|---|---|---|---|
| `bc`, 42,607 PPLOC | | | | |
| True errors | Other | 0 | 1 | 24 |
| False positives | Uns. | 2 | 0 | - |
| `js`, 309,428 PPLOC | | | | |
| True errors | Inc. | 2 | 0 | 0 |
| | Confus. | 16 | 1 | - |
| | Other | 0 | 96 | 99 |
| Dead code | Config. | 2 | 0 | - |
| | Macro | 11+28 | 11+5 | - |
| False positives | Uns. | 5 | 1 | - |
| Unknown | Unk. | 0 | 3 | - |
| `ncompress`, 2,915 PPLOC | | | | |
| | No warnings | | | |
| `pcc`, 72,948 PPLOC | | | | |
| True errors | Inc. | 0 | 1 | 0+1+0 |
| | Other | 0 | 22 | 35 |
| Dead code | Trivial | 1 | 0 | - |
| | Config. | 2 | 0 | - |
| | Macro | 1 | 0 | - |
| False positives | Uns. | 1 | 2 | - |
| `squid`, 807,286 PPLOC | | | | |
| True errors | Inc. | 1+0 | 1+0 | 1+1+1 |
| | Confus. | 4 | 0 | - |
| | Other | 0 | 20 | 243 |
| Dead code | Trivial | 2 | 0 | - |
| | Macro | 1+21 | +1 | - |
| `tiff`, 172,842 PPLOC | | | | |
| True errors | Inc. | 0 | 0 | 0 |
| | Confus. | 3 | 0 | - |
| | Other | 0 | 3 | 56 |
| Dead code | Trivial | 1 | 0 | - |
| | Macro | 3 | 1 | - |

**Table 4: Warnings from SAMATE Benchmarks**

| | Type | Curate | Saturn | Clang |
|---|---|---|---|---|
| True errors | Inc. | 6+16 | 6+0 | 6 |
| | Confus. | 1 | 0 | 0 |
| | Other | 0 | 20 | 377 |
| False positives | Config. | 2 | 0 | 0 |

## 6.8 Performance

Along with defect detection rates, we also measured the time taken to perform various aspects of universal reachability analysis. First, we measured the total time taken, per benchmark, for each of the three tools. We also instrumented Curate to record the time taken to analyze each function.

The elapsed times for all three analysis tools appear in Fig. 6. The times for Saturn and Curate include only analysis time, and are calculated using timing routines built in to the tools. Parsing time (for Curate) and database construction time (for Saturn) are not included. Similarly, the time to do exit function analysis, the results of which are shared between the two tools, is not included.

**Table 5: Warnings from Boundary Condition Specialization**

| Benchmark | True Errors | False Positives |
|---|---|---|
| MPlayer | 2 | 62 |
| openssh | 0 | 5 |
| openssl | 0 | 25 |
| samba | - | - |
| sendmail | 0 | 4 |
| bc | 6 | 0 |
| js | 1 | 4 |
| ncompress | 0 | 0 |
| pcc | 0 | 4 |
| squid | 0 | 5 |
| tiff | 0 | 2 |
| SAMATE | 836 | 0 |

**Table 6: Overall Analysis Time**

| | Analysis Time (H:M:S)/Timeouts | | | | |
|---|---|---|---|---|---|
| Benchmark | Curate | | Saturn | | Clang |
| MPlayer | 10:46:55 | 258 | 8:49:38 | 170 | 22:34 |
| openssh | 14:41 | 4 | 45:23 | 14 | 1:31 |
| openssl | 1:16:01 | 15 | 10:46:39 | 191 | 3:44 |
| samba | 1:44:51 | 12 | 33:59:38 | 738 | 10:23 |
| sendmail | 1:12:59 | 26 | 3:59:20 | 88 | 5:22 |
| bc | 3:56 | 1 | 23:11 | 12 | 0:21 |
| js | 1:13:46 | 25 | 5:00:36 | 137 | 2:59 |
| ncompress | 2:29 | 0 | 0:39 | 0 | 0:03 |
| pcc | 18:45 | 7 | 46:35 | 20 | 0:57 |
| squid | 10:34 | 1 | 1:57:25 | 50 | 1:43 |
| tiff | 13:14 | 4 | 28:48 | 7 | 1:02 |
| SAMATE | 4:57 | 0 | 24:29 | 0 | 0:56 |
| Total | 17:23:08 | 353 | 67:22:21 | 1427 | 51:35 |

The time taken by the Clang Analyzer, however, is elapsed time as reported by the Unix `time` command.

These numbers align closely with the engineering choices of each tool. Saturn performs inter-procedural analysis and uses a SAT solver to check for inconsistencies. Curate performs an intraprocedural analysis, and uses an SMT solver, which can solve certain queries more efficiently. Finally, the Clang Analyzer performs a well-engineered but coarse symbolic execution that is very effective at finding simple bugs.

Universal reachability analysis of a statement requires analysis of a number of wedges, each of which is similar in size and structure to the original statement.

We measured the total time taken to perform universal reachability analysis on each function body. The measured time includes the total time to analyze all of the wedges. The distribution of analysis times, in seconds, appears in Fig. 7. In this histogram, the first bar represents 98% of function bodies, all of which took less than one second to analyze. We show them compared to the function $2000x^{-1.8}$ for reference.

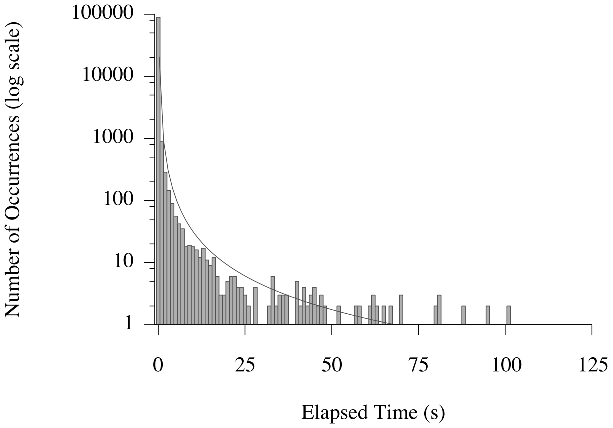

**Figure 7: Distribution of Function Analysis Times**

# 7. RELATED WORK

## 7.1 Inconsistency Analysis

The work presented in this paper is inspired by previous work on semantic inconsistencies. Much of the current work in this area builds on the 2001 paper by Engler *et al.* [5], which discusses contradictions between the beliefs implied by particular constructs in source code. Most of the inconsistency-detection tools derived from this research use symbolic execution as a core analysis.

One of the key works building on Engler's research, by Dillig, *et al.* [4] operates on the same overall principle, namely searching for cases where code indicates contradictory programmer beliefs.

The class of bugs detected by our analysis overlaps significantly with the work of Dillig *et al.*. The inconsistencies discovered by their analysis can be divided into sequential and non-sequential inconsistencies. A sequential inconsistency is one in which two program points that can execute in sequence along a single path make conflicting assumptions. For example, consider the following fragment of C code:

```
if(x) y = *x;
 ...
z = *x;
```

The first assignment suggests that x is allowed to be null. However, the second assignment will only succeed if x is non-null. A non-sequential inconsistency, on the other hand, is one in which two program points that may not both execute along the same path make conflicting assumptions. For example:

```
if (c1) { if (x) y = *x; }
if (c2) { y = *x; }
```

Here, the fact that the dereference is guarded by a null check in one case but not the other suggests an error. However, consider the wedge in which the `else` branch of the null check executes. It is still possible for this wedge to terminate, because it is possible that c2 is false, and therefore the second dereference never occurs.

Any sequential inconsistency discovered by the analysis of Dillig *et al.* will also be discovered by our analysis. Non-sequential inconsistencies will be discovered by their analysis but not ours.

Note that sequential inconsistencies will either lead to assertion failures or involve dead code. Parallel inconsistencies may be neither. They suggest the possibility of conflicting programmer beliefs, but in our experience led to more false positives than true bugs, though some did indicate significant bugs.

## 7.2 Static and Dynamic Contract Checking

Many programming languages and external analysis tools have adopted the use of one of a variety of contract systems.

Systems such as ESC/Java [7] and Spec# [1] adopt the technique of adding contracts to implementation code, but try to statically verify that the contracts hold, using an automated theorem prover, rather than depending on dynamic checks. Tools of this sort depend on extensive contract annotations and generate many false positives on unannotated programs.

The Eiffel language [13] also emphasizes function contracts, in which specifications written in Eiffel code accompany the implementation. This technique allows for very expressive specifications, but executes specification code dynamically, and therefore detects errors late in the development process.

## 7.3 Flexible Type Systems

Hybrid type checking [6] also uses an undecidable dependent type system, and has a similar philosophy to universal reachability analysis in that it rejects a program only if it can prove the program violates its specification. If the type checker cannot prove type soundness, but cannot disprove it either, it resorts to dynamic checking.

Gradual type systems take a similar approach, focusing on the latter benefit of allowing any degree of annotation along a spectrum, from completely dynamic to completely static [16]. In gradual type systems, type checking has typically been decidable, but some components may be untyped (or marked implicitly with the type Dynamic). Interfaces between code that has statically-known types and code that uses dynamic types require runtime casting operations that may fail.

# 8. SUMMARY

Automated tools for detecting inconsistencies in software can reveal serious bugs. We argue that axiomatic semantics can form the basis of an elegant and extensible technique for inconsistency detection that generalizes previous approaches. Our approach retains the benefits of previous inconsistency-detection techniques, and brings advantages of its own.

We have attempted to generalize and formalize existing approaches to inconsistency analysis by building on the foundation of axiomatic semantics, resulting in universal reachability analysis. This approach to inconsistency detection has a number of desirable properties that go beyond those of inconsistency analysis:

- The definition of the analysis is strongly connected to the semantics of the underlying language. Therefore, we can prove that the analysis correctly captures the details of the program being analyzed.
- The same implementation can be used for both inconsistency detection and deductive verification. Under-specified code can be checked for obvious mistakes, and fully-specified code can be verified, simply by varying the parameters to the analysis.

# 9. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grants CNS-0905650 and CCF-1116883.

# 10. REFERENCES

[1] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2004.

[2] Flaviu Cristian. Exception handling. In *Dependability of Resilient Computers*, pages 68–97, 1989.

[3] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., October 1976.

[4] Isil Dillig, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 435–445, New York, NY, USA, 2007. ACM.

[5] Dawson Engler, David Y. Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Operating Systems Review*, 35(5):57–72, October 2001.

[6] Cormac Flanagan. Hybrid type checking. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 245–256, January 2006.

[7] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 234–245, June 2002.

[8] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 193–205, January 2001.

[9] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–31. American Mathematical Society, Providence, R.I., 1967.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[11] RTCA Inc. DO-178B, software considerations in airborne systems and equipment certification.

[12] SRI International. Yices SMT solver. `http://yices.csl.sri.com/`.

[13] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[14] National Institute of Standards and Technology. SAMATE: Software Assurance Metrics And Tool Evaluation. `http://samate.nist.gov/`.

[15] The LLVM Project. Clang static analyzer. `http://clang-analyzer.llvm.org/`.

[16] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

[17] Aaron Tomb. *Program Inconsistency Detection: Universal Reachability Analysis and Conditional Slicing*. Ph.D. dissertation, University of California, Santa Cruz, 2011.