Conflict-Directed Graph Coverage

Daniel Schwartz-Narbonne[1], Martin Schäf[2], Dejan Jovanović[2], Philipp Rümmer[3], and Thomas Wies[1]

[1] New York University
[2] SRI International
[3] Uppsala University

Abstract. Many formal method tools for increasing software reliability apply Satisfiability Modulo Theories (SMT) solvers to enumerate feasible paths in a program subject to certain coverage criteria. Examples include inconsistent code detection tools and concolic test case generators. These tools have in common that they typically treat the SMT solver as a black box, relying on its ability to efficiently search through large search spaces. However, in practice the performance of SMT solvers often degrades significantly if the search involves reasoning about complex control-flow. In this paper, we open the black box and devise a new algorithm for this problem domain that we call conflict-directed graph coverage. Our algorithm relies on two core components of an SMT solver, namely conflict-directed learning and deduction by propagation, and applies domain-specific modifications for reasoning about control-flow graphs. We implemented conflict-directed coverage and used it for detecting code inconsistencies in several large Java open-source projects with over one million lines of code in total. The new algorithm yields significant performance gains on average compared to previous algorithms and reduces the running times on hard search instances from hours to seconds.

1 Introduction

Inconsistent code represents a class of program abnormalities whose detection has attracted considerable attention over the past years [4, 15, 18, 20, 28, 29]. A statement in a program is considered inconsistent if it never partakes in any properly terminating execution of the program. That is, the statement is either unreachable or any execution passing through it must inevitably lead to an unrecoverable error.

Detecting inconsistent statements is important for several reasons. First, inconsistent code is closely correlated with the existence of real bugs and is difficult to detect using testing. For example, Wang et al. [29] have used code inconsistency detection to identify optimization-unstable code in C programs[4] and discovered previously unknown optimization-related bugs in the Linux kernel.

[4] The C standard allows compilers to eliminate code with undefined behavior. Such code can be formalized as being inconsistent.

The inconsistency detection tool Joogie developed by some of the authors [4] has revealed several new bugs in open-source Java programs, including Apache Tomcat and Ant [25]. Second, code inconsistencies can be detected statically and locally by checking individual procedures in isolation, without requiring precise procedure contracts. This enables the use of theorem provers to obtain a fully automated analysis that scales to entire programs. Well, not entirely . . . A small number of indomitable inconsistencies still holds out against efficient detection.

In this paper, we present a new algorithm for detecting inconsistencies efficiently, realizing considerable performance improvements over existing algorithms in practice.

Most existing algorithms implement variations of the following basic idea: for a given procedure, one first computes an abstraction to obtain an acyclic control-flow graph (see, e.g., [18]). This graph is then encoded into a first-order logic formula whose models can be mapped back to the feasible executions of the procedure. Then, an SMT solver is repeatedly queried to obtain such feasible executions. For each obtained execution, the statements on the corresponding control-flow path are marked as consistent and a condition is added to prevent the prover from finding other executions of the same path. The algorithm terminates once the formula becomes unsatisfiable and all program statements that have not been covered by a feasible path are reported as inconsistent.

All the variations of this basic algorithm have in common that they treat the SMT solver as a black box. They completely rely on the solver's ability to (1) efficiently reason about the propositional encoding of the control-flow graph; and (2) learn theory conflicts from the constraints on infeasible paths to avoid enumerating them one at a time. However, the interplay between propositional reasoning and theory reasoning in a general purpose SMT solver is complex. In particular, for search-intensive problems, it is unavoidable that the solver will end up doing theory reasoning across many different paths at once, even if the application only requires reasoning about individual paths. We have observed that this can lead to severe performance degradation during inconsistent code detection, in particular, if inconsistent statements participate in many paths of a control-flow graph.

Our new algorithm builds upon the basic search loop of an SMT solver. That is, we use (1) a DPLL-style SAT procedure to search for individual paths; and we rely (2) on conflict-driven clause learning (CDCL) [27] to detect inconsistent statements efficiently by generalizing theory conflicts. However, we propose several important application-specific modifications to the standard CDCL and DPLL procedures. First, we completely separate the search for paths in the control-flow graph from checking feasibility of these paths. That is, theory reasoning and conflict learning are always restricted to single complete paths. Second, we exploit that the propositional formula given to the DPLL procedure encodes a control-flow graph. Namely, we devise specialized propagation rules that accelerate the search for individual paths. Together, these modifications ensure that none of the solver components gets confused by the complex propositional structure of the control-flow graph encoding.

```
 1 boolean equals(MyClass other) {
 2   if (this == other) return true;
 3   if (other == null) return false;
 4   if (getClass() != other.getClass()) return false;
 5   if (bases == null) {
 6     if (other.bases != null) return false;
 7     //inconsistent with line 9
 8   }
 9   if (bases.hashCode() != other.bases.hashCode())
10     return false;
11   return true;
12 }
```

Fig. 1. A method with inconsistent code taken from the Bouncy Castle library. Any path through line 7 (i.e., the implicit else block of line 6) requires that `bases` and `other.bases` are `null`, which is inconsistent with the fact that these objects are dereferenced in line 9.

We have implemented our algorithm in the tool GraVy [26] and applied it to several open-source Java projects consisting of more than thirty thousand methods and over one million lines of code in total. Our evaluation shows that, even though difficult search-intensive problem instances are relatively rare, our new algorithm leads to considerable performance improvements on average. For individual difficult instances, we have observed performance improvements of up to two orders of magnitude, reducing the running times of the analysis from hours to seconds.

2 Overview

We explain our Conflict-Directed Coverage algorithm along the code snippet in Figure 1. The snippet shows an occurrence of inconsistent code that we found and fixed in the cryptography library Bouncy Castle [1]. In this example, line 7 can only be reached by executions where `other.bases` is `null`. Further, any such execution must also reach line 9 which dereferences `other.bases`. Thus, there can be no normally terminating execution going through line 7 since any such execution will throw a null pointer exception in line 9. Therefore, line 7 is inconsistent.

To find this inconsistency, existing algorithms translate the program into a formula in first-order logic whose models encode the feasible complete executions of the program. The formula is sent to an SMT solver. Once a model is found, the formula is extended to block all other models that can be mapped to the same control-flow path. This way, each time the theorem prover is queried, it has to find a new feasible complete path. Once this formula becomes unsatisfiable, we know that all statements that occur on feasible complete paths have been covered and the remaining statements are inconsistent.

```
try {
 ...
 while((obj = ois.x()) != null) {
  if (obj instanceof SampleEvent)
        {
   try {
    ...
   } catch (...Exception err) {
    if (...){
       throw new ...;
    }
   }
  } else {...}
 }
} catch (EOFException err) {...
} catch (IOException err) { ...
} catch (...Exception err) { ...
} finally {
  try {...}
  catch (...Exception e) {...}
  if(!temporaryFile.delete())
       {...}
}
```

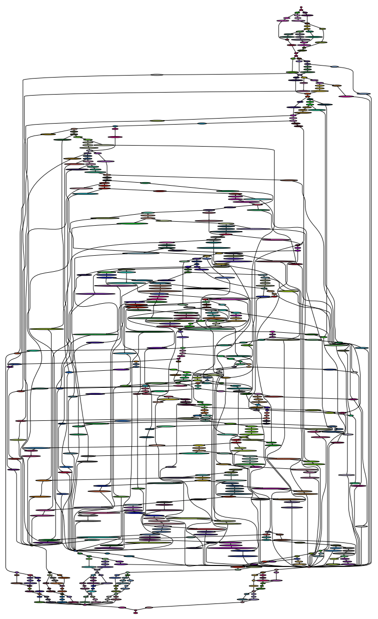

Fig. 2. The CFG of the method `testEnded` from `DiskStoreSampleSender` in the Apache `jMeter` project. The listing on the left shows an excerpt of the Java code. The method consists of only 55 lines but a combination of switch-cases, loops, and complex exception handling results in complex control-flow. The right-hand side shows the CFG. Each of the 659 nodes represents a block of straight-line bytecode instructions.

This approach is efficient as long as it is easy to find new models. In our example, we would quickly cover everything other than line 7. In order to show that line 7 is inconsistent, however, the theorem prover still has to show that all complete paths through that line are infeasible. In this example, there are only two such paths, so the prover will not struggle to solve this quickly. In practice, however, the number of paths can be prohibitively large.

Figure 2 shows an excerpt of a method from the Apache `jMeter` project and its control-flow graph. This example shows that even a relatively small Java method (here 55 lines) can result in very complex control-flow graphs. Our previous algorithm is very fast at finding feasible paths through the first 600 of 659 nodes in the graph. Finding paths for the remaining nodes, however, takes over 90 minutes, because the solver starts enumerating all paths through these nodes to show the absence of feasible paths.

To avoid this worst-case enumeration of all paths, we propose a new algorithm that learns conflict clauses from each infeasible path. These clauses then prune entire subgraphs to show that no further models exist. We explain this approach using the simple example from Figure 1. Let us assume that we have already covered all feasible paths in the method. What is left are two paths, one passing the lines 7 and 10, and the other passing the lines 7 and 11. Let us further assume that the SMT solver first checks feasibility of the path through the lines 7 and 10. The query for this check looks roughly as follows:

$$\begin{aligned}
&\ldots\\
&\texttt{bases} = \texttt{null} \wedge\\
&\texttt{other} \neq \texttt{null} \wedge\\
&\texttt{other} \neq \texttt{null} \wedge\\
&\texttt{other.bases} = \texttt{null} \wedge\\
&\texttt{bases} \neq \texttt{null} \wedge\\
&\texttt{other.bases} \neq \texttt{null} \wedge\\
&\texttt{bases.hashCode()} = \texttt{other.bases.hashCode()} \wedge\\
&\texttt{ret} = \texttt{false}
\end{aligned}$$

This formula is unsatisfiable because of the contradicting clauses `other.bases` = `null` generated from line 7 and `other.bases` ≠ `null` generated from line 10. Our algorithm extracts this conflict by computing a minimal unsatisfiable core of the formula.[5]

A theorem prover would learn this conflict as well, however, it would be unable to infer the clause `other.bases` ≠ `null` in the conflict also guards the other path through line 7 directly. When reasoning about the control-flow, which is typically implemented through Boolean variables intertwined with the actual transition relations (e.g., [6]), the solver would not consider the information that it should not attempt any further path containing the learned conflict. In the worst-case, this causes the theorem prover to enumerate many paths that are known to be infeasible already. This problem has, e.g., been described in [10,13].

To avoid enumerating all paths, our algorithm checks if all paths through line 7 must contain the clauses from this unsatisfiable core. To that end, it turns the problem of finding the next path into a SAT problem. The structure of the control-flow graph is encoded into a SAT formula which is constrained by the fact that the node we want to cover must be used, that source and sink of the graph must be used, and that no model is allowed that contains any of the learned conflicts. Further, we give the solver additional propagation rules presented in [3] that simplify reasoning about graphs. E.g., if the solver picks a transition through one side of a diamond, we enforce that no transition on the other side of the diamond can be chosen.

Using the learned conflicts and the knowledge about the structure of the control-flow graph, we are able to establish the inconsistency of line 7 immediately without analyzing any further path and our algorithm terminates, reporting line 7 to be inconsistent. We discuss our algorithm in more detail in Section 4.

Note that for a small example such as the one shown in Fig 1, our algorithm will typically be less efficient than a simple algorithm that enumerates

[5] The following argument is unchanged if the solver instead found the unsat core `bases` = `null` and `bases` ≠ `null`.

all infeasible paths explicitly because computing unsatisfiable cores is relatively expensive. Our algorithm targets big problems where existing algorithms fail because of the enormous number of paths. In particular, for the example shown in Fig. 2, our new algorithm manages to identify infeasible subgraphs efficiently using the computed conflict clauses, ruling out many complete infeasible paths simultaneously. The method `testEnded` can therefore be analyzed in less than 1 minute compared to the 90 minutes with our previous algorithm. This example shows that our new approach leads to a significant performance improvement if inconsistencies can be detected locally. If inconsistencies can not be generalized to larger subgraphs, we will not see any performance gain as our algorithm will fall back to enumerating individual infeasible paths. However, our evaluation in Section 5 shows that, in practice, inconsistencies in real code often generalize and, hence, our new algorithm works much better.

3 Preliminaries

We represent programs as control-flow automata where program statements are encoded directly in terms of their transition relations, expressed as formulas in first-order logic. We assume standard syntax of such formulas and we assume that their semantics is given by some appropriate first-order theory that we leave unspecified. We only assume that all values used for the interpretation of formulas are drawn from a fixed set V. As usual, for a formula ϕ with free variables X and a valuation $\sigma : X \to V$, we write $\sigma \models \phi$ to indicate that σ satisfies ϕ.

We fix a set of variables X, which we call *program variables*. In order to be able to reason over the states in a program execution, we define a sequence of variable sets $\{X^{\langle i \rangle}\}_{i \in \mathbb{N}}$. We use the variables in $X^{\langle i \rangle}$ to describe a program state that has been reached after executing i statements in the program. Note that for a loop free program, $X^{\langle i \rangle}$ is effectively equivalent to SSA form [7,12]. Formally, define $X^{\langle 0 \rangle} = X$ and for all $i > 0$, let $X^{\langle i \rangle}$ be a set of variables that is of equal cardinality than X and pairwise disjoint from all other $X^{\langle j \rangle}$, $j \neq i$. Let $\cdot^{\langle 1 \rangle}$ be a bijective function that maps the variables $X^{\langle i \rangle}$ to the variables $X^{\langle i+1 \rangle}$, for all $i \in \mathbb{N}$. For a variable x we denote by $x^{\langle i \rangle}$ the result of applying this function i times to x. We extend this *shift* function to formulas as expected.

Now, a program P is formally defined as a tuple (L, T, ℓ_0, ℓ_f) where

- L is a finite set of control locations,
- T is a finite set of statements (ℓ, φ, ℓ') with $\ell, \ell' \in L$ and φ a formula over the variables $X \cup X'$,
- $\ell_0 \in L$ is the initial location, and
- $\ell_f \in L$ is the final location.

For a statement $\tau = (\ell, \varphi, \ell')$ we define $\mathsf{start}(\tau) = \ell$, $\mathsf{end}(\tau) = \ell'$, and $\mathsf{tr}(\tau) = \varphi$. We call $\mathsf{tr}(\tau)$ the *transition formula* of τ.

A state $s = (\ell, \sigma)$ of a program P consists of a program location ℓ and a valuation of the program variables $\sigma : X \to V$. We denote the location of a state

s by $\mathsf{loc}(s)$ and its valuation by $\mathsf{val}(s)$. For a valuation $\sigma : X \to V$ and $i \in \mathbb{N}$, we define $\sigma^{\langle i \rangle} : X^{\langle i \rangle} \to V$ such that for all $x \in X$, $\sigma^{\langle i \rangle}(x^{\langle i \rangle}) = \sigma(x)$. Similarly, we define $s^{\langle i \rangle} = (\ell, \sigma^{\langle i \rangle})$ for a state $s = (\ell, \sigma)$.

A path π of program P is a finite sequence $\tau_0, \ldots, \tau_n$ of P's statements such that $\mathsf{start}(\tau_0) = \ell_0$, $\mathsf{end}(\tau_n) = \ell_f$, and for all $i \in [0, n)$, $\mathsf{end}(\tau_i) = \mathsf{start}(\tau_{i+1})$. We extend transition formulas from statements to paths by defining

$$\mathsf{tr}(\pi) = \mathsf{tr}(\tau_0)^{\langle 0 \rangle} \wedge \cdots \wedge \mathsf{tr}(\tau_n)^{\langle n \rangle} \ .$$

We call $\mathsf{tr}(\pi)$ the *path formula* of π. A path formula encodes the executions of the path. That is, projecting a model σ_π of π onto the variables $X^{\langle i \rangle}$ yields $\mathsf{val}(s^{\langle i \rangle})$, the valuation of the i-th state s in the execution.

We call a path π *feasible* if its path formula is satisfiable. A statement $\tau \in T$ is *inconsistent* if it does not occur on any feasible path of P.

Finally, we define a preorder $\preceq$ on statements that captures the notion of domination in control flow graphs: for $\tau_1, \tau_2 \in T$ we have $\tau_1 \preceq \tau_2$ iff for every path π of P, if τ_2 occurs on π, then so does τ_1. We write $\tau_1 \sim \tau_2$ if $\tau_1 \preceq \tau_2$ and $\tau_2 \preceq \tau_1$. Finally, for $\tau \in T$, we denote by $[\tau]_\sim$ the equivalence class of τ in the quotient set that is induced by the equivalence relation $\sim$. The following lemma states that inconsistency propagates along the domination relation, which gives rise to certain optimizations during inconsistency detection.

Lemma 1. *Let τ_1 and τ_2 be statements of a program. If $\tau_1 \preceq \tau_2$ and τ_1 is inconsistent, then so is τ_2.*

4 Algorithm

Our method for detection of infeasible code operates over loop-free graphs, and we assume that the input programs are loop-free. We refer the reader to existing work (e.g. [18] and [28]) that describes the algorithms for sound loop abstraction in the context of inconsistent code detection. Program statements are encoded in SSA form [7,12].

4.1 Main Procedure

The main procedure Cover is described in Algorithm 1. Cover takes a loop-free control-flow graph as input and returns the set of statements C that can occur on feasible paths.

The procedure Cover starts by collecting the set of all statements that need to be discharged (i.e., proven inconsistent, or covered by a feasible path) in the set S, and setting the set C to empty.

In each iteration of the main loop, the procedure then tries to cover a statement τ from S, or prove that it is inconsistent. The statement τ is picked using the procedure **FindMaxElement**. This procedure takes the set of not-yet-covered statements S, the control-flow graph cfg, and returns the statement $\tau \in S$ that is maximal with respect to the pre-order $\preceq$ defined in Section 3. The

Algorithm 1: Cover

```
Input: The control-flow graph cfg.
Output: The set C of statements that occur on feasible complete paths in cfg.
begin
    $S \leftarrow$ GetStatements(cfg) ;
    $C \leftarrow \emptyset$ ;
    repeat
        $\tau \leftarrow$ FindMaxElement(cfg, $S$);
        $\pi \leftarrow$ FeasiblePath(cfg, $\tau$) ;
        if $\pi = \bot$ then
            $S \leftarrow S \setminus [\tau]_\sim$ ;
        else
            $S \leftarrow S \setminus \pi$ ;
            $C \leftarrow C \cup \pi$ ;
        end if
    until $S = \emptyset$;
    return $C$;
end
```

intuition behind this choice is two-fold. First, we aim at covering a particular node, and we focus the underlying solver on the sub-graph of paths that contain it. Second, by picking a τ to be $\preceq$-maximal, we make sure that this sub-graph is indeed reduced since it will exclude the nodes that are $\preceq$-sibling to τ (i.e. branches adjacent to τ).

The statement τ returned by **FindMaxElement** is then delegated to the FeasiblePath procedure that checks whether there exists a feasible path through this particular statement. If such a path π exists, all statements in π are then removed from S, since they are also covered by π. Otherwise, if no such path exists, the statement τ is discharged as inconsistent along with all the nodes in its equivalence class $[\tau]_\sim$ (by Lemma 1).

This procedure repeats this loop until the set of statements S becomes empty, implying that it has discharged all statements. Since in each loop iteration at least one statement is removed from S, the procedure is guaranteed to terminate.

4.2 Finding Feasible Paths

Algorithm 2 describes the FeasiblePath procedure that discharges inconsistency of a particular statement τ. The procedure takes as input the loop-free control-flow graph cfg, and the statement τ in cfg, and either returns $\bot$, if τ is inconsistent in cfg, or a feasible full path π that contains τ.

Algorithm 2 implements a typical CDCL-style search loop where path candidates are generated, while learning from unsuccessful attempts, until either a full solution is found (a feasible path), or it can be shown that no such path can exist. The set of conflicts encountered during the search is kept in the set conflicts that is initialized to the empty set. Each conflict is a set of statements

Algorithm 2: FeasiblePath

```
Input: The control flow graph cfg and the statement $\tau$ to be discharged.
Output: A feasible complete path $\pi$ that covers $\tau$, or $\perp$ if $\tau$ is inconsistent.
begin
    conflicts $\leftarrow \emptyset$ ;
    while true do
        $\pi \leftarrow$ FindPath(cfg, $\tau$, conflicts) ;
        if $\pi = \perp$ then
            return $\perp$
        if CheckSat($\pi$) then
            return $\pi$ ;
        else
            core $\leftarrow$ UnsatCore($\pi$) ;
            conflicts $\leftarrow$ conflicts $\cup$ {core} ;
        end if
    end while
end
```

such that any full path that includes all statements of the conflicts can not be feasible.

In each iteration of the loop, the procedure first picks *a candidate* path π from the cfg using the procedure **FindPath**. The procedure **FindPath** takes as input the control-flow graph, the statement τ, and the set of learned conflicts, and returns a complete path through τ that does not contain any of the already known conflicts. This path need not be feasible, i.e. the only requirement on **FindPath** is that the path must be complete and not pass through any of the conflicts discovered so far. This can be implemented using a simple call to a SAT-solver, and does not require any SMT reasoning, making it very efficient. Our implementation uses a dedicated path solver for control-flow graphs [3] available in the Princess theorem prover.

If **FindPath** returns that no path through τ exists, we have a proof that τ is inconsistent, and we return $\perp$. Otherwise, we take the candidate path π and check whether it is feasible by calling the Princess theorem prover [24] to check if the formula $\mathtt{tr}(\pi)$ is satisfiable. If so, we know that τ is not inconsistent and the algorithm returns the path π. If $\mathtt{tr}(\pi)$ is unsatisfiable, the procedure computes a minimal unsatisfiable core of $\mathtt{tr}(\pi)$. [6] Since any path that includes all statements from the core will also be infeasible, the core is then added to the conflict set.

The **FeasiblePath** procedure terminates because each iteration of the loop either leaves the loop directly, or learns one new conflict. Each conflict eliminates a set of paths from the control-flow graph (in the worst-case only π). That is,

[6] Our current implementation finds the minimal core by taking the full path and then removing each statement that can be removed while keeping unsatisfiability. This is feasible only because path satisfiablity is very efficient to check.

since the set of paths in the control-flow graph is finite, the algorithm terminates eventually.

If the unsatisfiable cores that the procedure computes always contain the full paths, the procedure can diverge into complete path enumeration. However, in practice, our algorithm detects inconsistent code without enumerating all paths, as shown in our evaluation on some challenging real-world problems.

4.3 Details of the FindPath Function

Our algorithm works for any implementation of **FindPath** that computes, given the control-flow graph, the statement τ to be covered, and the set of conflicts, a single complete path through τ. Computing such paths reduces to the well-known problem of finding *hitting sets,* and is in general NP-hard [21]; it is therefore meaningful to employ a theorem prover or SAT solver for computing such paths.

Our implementation of **FindPath** uses the control-flow theory developed in [3], which provides a tailor-made propagator for reasoning about control-flow graphs. This propagator can be loaded as a plug-in into a theorem prover, and ensures that only models are computed that correspond to single, complete paths in the graph, reducing the number of explicit Boolean clauses needed to encode those constraints.

5 Evaluation

We evaluated our algorithm by comparing it against the algorithm from our Joogie tool [4]. The algorithm in Joogie uses a greedy path cover strategy to find all feasible paths. To that end, it used a special theory in the SMT solver that allows it to reason about the control-flow more efficiently [3]. Our implementation uses the same theory when searching for new paths. For comparability, both algorithms use Princess as the underlying SMT solver. The code of all algorithms and all benchmark problems are available online [26].

Setup. We compare the two algorithms on a set of open-source programs. Both algorithms are applied to each procedure in these programs. That is, the algorithms do not perform inter-procedural analysis. The task is to find all inconsistent statements in each individual procedure. Since both algorithms use the same front-end and the same loop abstraction, the set of inconsistent statements is the same in both cases.

Each algorithm is given ten minutes per procedure before it is stopped by a timeout. Procedures that timeout are counted as ten minutes computation time. The goal of the experiment is to show that our algorithm times out less often and is able to analyze the benchmark programs faster. We do not discuss the inconsistent code that was found as this would exceed the scope of this paper. For the interested reader, we provide some examples of inconsistent code that we found and fixed in our wiki.[7]

[7] https://github.com/martinschaef/gravy/wiki/Bugs-found-so-far.

			Joogie Algorithm		New Algorithm	
Benchmark	# procedures	KLOC	time	timeouts	time	timeouts
Ant	10,563	271	23,150s	34	13,136s	12
Bouncy Castle	10,692	461	8,002s	7	8,499s	6
Cassandra	1,110	237	1,073s	1	969s	1
jMeter	3,712	114	7,302s	11	2,833s	2
Log4j	2,836	65	2,950s	4	1,960s	1
Maven (Core)	2,307	43	2,793s	4	1,192s	1

Table 1. Results of running the Conflict-Directed Inconsistency Detection algorithm against the optimized algorithm from [3]. The algorithms are applied to each procedure in isolation. The timeout per procedure is 10 minutes. The time shown is the overall running time from start to end including logging, etc. All experiments are carried out on a 2.8 GHz i7 with 16GB memory. The JVM is started with `-Xmx4g -Xms4g -Xss4m`.

Our benchmark suite consists of a variety of open-source Java programs (see Table 1). For each benchmark program, we used the source code available from GitHub at the day of submission for our experiments. Benchmark programs are translated into Boogie [23] which serves as the input language to our algorithm. The translation is sound for large parts of the Java language but abstracts certain language features such as multithreading and reflection. Looping control-flow and procedure calls are eliminated using the sound abstraction described in [18]. We do not discuss translation and abstraction in further detail. However, we emphasize that both algorithms in our experiments use the exact same code for the abstraction to ensure that the results are comparable. Details on the translation can be found on the website mentioned above.

Results. Table 1 shows the results of our experiment. The first three columns show the name of each benchmark, the total number of analyzed procedures, and the number of lines of code in thousands (excluding comments). Column four and five show the computation time and the number of timeouts for the Joogie algorithm, and the last two columns show the same for our Conflict-Directed Inconsistency Detection Algorithm.

For all benchmarks, we can see a reduction in the number of timeouts. For all benchmarks other than Bouncy Castle, the new algorithm was also faster then the old algorithm; for Bouncy Castle, it took slightly longer but succeeded on more procedures. For Ant, our new algorithm is almost twice as fast and only has one third of the timeouts. The large difference in computation time can mostly be attributed to the number of timeouts. Without considering the procedures that timeout, both algorithms take roughly the same amount of time. Most of the procedures for which the old algorithm timeouts are of a similar nature to the one in Figure 2 of Section 2: a large control-flow graph, where a small subgraph is inconsistent. Our algorithm can establish the inconsistency after learning clauses from a few infeasible paths (in our experiments, the algorithm never needed to learn more than 10 conflicts). The old algorithm, on the other

Benchmark	# Joogie Better	# New Algorithm Better	Both timeout
Ant	1	23	11
Bouncy Castle	0	1	6
Cassandra	0	0	1
jMeter	0	9	2
Log4j	1	4	0
Maven (Core)	0	3	1

Table 2. Distribution of timeouts per benchmark. Column 2 is the number of procedures where the old algorithm succeeded within 10 minutes, but the new algorithm timed out. The third column shows the number of procedures in that benchmark where the new algorithm succeeded within 10 minutes, but the old algorithm timed out. The final column shows the number of examples where both algorithms timed out.

	Joogie Algorithm				New Algorithm			
Benchmark	$\leq$ 1s	$>$ 1s	$>$ 100s	$>$ 300s	$\leq$ 1s	$>$ 1s	$>$ 100s	$>$ 300s
Ant	97.73%	2.26%	0.35%	0.33%	97.88%	2.11%	0.23%	0.16%
Bouncy Castle	98.87%	1.11%	0.18%	0.09%	98.88%	1.12%	0.16%	0.08%
Cassandra	98.46%	1.53%	0.27%	0.09%	98.46%	1.53%	0.18%	0.09%
jMeter	98.16%	1.83%	0.32%	0.32%	98.27%	1.72%	0.18%	0.13%
Log4j	98.80%	1.19%	0.17%	0.14%	98.80%	1.19%	0.17%	0.14%
Maven (Core)	97.83%	2.16%	0.17%	0.17%	98.17%	1.82%	0.08%	0.04%

Table 3. Distribution of computation time per benchmark. For the respective algorithm, we show the percentage of procedures that can be analyzed in less than a second, more than one second, more than 100 seconds, and more than 300 seconds.

hand, starts enumerating all paths which is impractical for problems of that size and therefore leads to a timeout after ten minutes.

Table 2 shows the relative timeouts between the two algorithms. The first column shows the name of the benchmark. The second column shows the number of procedures in that benchmark where the old algorithm succeeded within 10 minutes, but the new algorithm timed out. The third column shows the number of procedures in that benchmark where the new algorithm succeeded within 10 minutes, but the old algorithm timed out. The final column shows the number of examples where both algorithms timed out. For all benchmarks, the new algorithm had significantly fewer timeouts than the old algorithm.

Table 3 shows the distribution of computation time per benchmark in more detail. For both algorithms, the table shows the fraction of procedures that can be analyzed in less than a second, over a second, over 100 seconds, and over 300 seconds (with a timeout of 600 seconds). The table shows that our new algorithm has the most impact on procedures in the range above 100 or 300 seconds. That is, while we see improvements across all columns, our algorithm is particularly strong on procedures where the old algorithm struggles.

The performance difference between the two algorithms would become more visible if we would not timeout the algorithms after 10 minutes. For example, for the procedure from Figure 2 of Section 2 the old algorithm took over 90 minutes where the new algorithm took less than one minute. However, running the experiments without timeout on such a large corpus of benchmarks seemed impractical.

Threats to validity. Although the benchmark applications is a potential threat to validity, we tried to select a diverse set of benchmark programs ranging from desktop applications (Log4j and Ant), over web applications (Cassandra), to cryptographic code (Bouncy Castle). Further, since both algorithms share a common front-end, we are confident that neither the selection of benchmarks nor the infrastructure is biased towards one of the algorithms.

Another potential threat to validity is the choice of the SMT solver. We chose Princess because it is the only solver that provides the CFG-theory [3], which we found to be vital for finding paths efficiently. During the development of our algorithm we tested whether the timeouts that we encountered in the old algorithm are specific to Princess. To that end, we exported several of our hard queries into the SMT-LIB format and checked them with Z3 [14], which also timed out.

6 Related Work

The concept of inconsistent code has been presented in several works (e.g., [4, 15, 28]), sometimes under different names such as infeasible code [9], doomed program points [18], or deviant behavior [16]. The approaches in [4,9,18,28] and a similar approach in [20] are based on deductive verification. In [4] and on our website [25] we demonstrate that these approaches can identify relevant coding mistakes even in mature code.

The algorithm presented in this paper can extend any of these approaches. The approach in [15] is based on a type-checking approach but unfortunately, no tool is available for comparison. Most compilers have built in data-flow and type-checking tools as well to detect inconsistent code. For example, most compilers do not allow the use of uninitialized variables, which can be seen as a form of inconsistency. Due to the simpler reasoning, these tools only detect a subset of what can be detected with deductive verification-based approaches but are also significantly faster.

The approach of Engler et al in [16] uses syntactic pattern matching and is therefore not directly comparable. Other syntactic tools, such as Findbugs [19] also detect a certain set of inconsistencies but have the usual limitations in terms of precision.

Inconsistent code detection has several other applications beyond finding coding mistakes. For example, [20] uses a variation of our algorithm to check reachability of annotated code. Our algorithm can be used in the same way and, beyond that, can identify code that must violate other specification statements.

That is, our approach can be extended to debug functional specification in the spirit of [17], [11], and [8].

Identifying all statements in a program that occur on feasible paths also has applications for the generation of verification counterexamples. Usually, deductive verification returns only one counterexample if the proof of the desired property fails. This can be time consuming as the verification has to be re-run very often. In [2], we have presented an approach how a coverage algorithm like the one presented in this paper can be used to identify all assertions in a procedure that may fail. This is closely related to other techniques that find multiple counterexamples for one verification attempt, such as [5], and also related to the problem extracting error traces from theorem prover models as presented in [22].

7 Conclusion

We have proposed a new algorithm for finding feasible paths in a program that satisfy certain coverage criteria. The algorithm makes application-specific modifications to the core components of an SMT solver in order to accelerate the search in complex control-flow graphs. We have used our algorithm to find code inconsistencies in large Java programs and showed that we gain significant performance improvements compared to previous algorithms. We believe that the usefulness of our algorithm extends beyond inconsistent code detection to applications with different coverage notions such as those used in concolic testing.

Acknowledgments. This work was in part supported by the NASA contract NNX14AI05A and the NSF grant CCS-1350574. The content is solely the responsibility of the authors and does not necessarily represent the official views of NASA or NSF.

References

1. The legion of the bouncy castle. `https://www.bouncycastle.org/`.
2. S. Arlt, C. Rubio-González, P. Rümmer, M. Schäf, and N. Shankar. The gradual verifier. In *NASA Formal Methods*, pages 313–327. Springer, 2014.
3. S. Arlt, P. Rümmer, and M. Schäf. A theory for control-flow graph exploration. In *ATVA*, pages 506–515, 2013.
4. S. Arlt and M. Schäf. Joogie: Infeasible code detection for java. In *CAV*, 2012.
5. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *SIGPLAN Not.*, pages 97–105, 2003.
6. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. *SIGSOFT Softw. Eng. Notes*, pages 82–87, 2005.
7. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 82–87. ACM, 2005.
8. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in actl formulas. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 279–290, London, UK, UK, 1997. Springer-Verlag.

9. C. Bertolini, M. Schäf, and P. Schweitzer. Infeasible code detection. In *VSTTE*, pages 310–325, 2012.
10. N. Bjørner, B. Dutertre, and L. de Moura. Accelerating lemma learning using joins–DPLL (join). In *Int. Conf. Logic for Programming, Artif. Intell. and Reasoning, LPAR*, 2008.
11. H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage Metrics for Temporal Logic Model Checking. In *TACAS*, pages 528–542, 2001.
12. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, pages 451–490, 1991.
13. L. de Moura and N. Bjørner. Relevancy propagation. Technical Report MSR-TR-2007-140, Microsoft Research, 2007.
14. L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
15. I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *PLDI*, 2007.
16. D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.
17. M. Gheorghiu and A. Gurfinkel. Vaquot: A tool for vacuity detection. Technical report, In Proceedings of Tool Track, FM'06, 2005.
18. J. Hoenicke, K. R. Leino, A. Podelski, M. Schäf, and T. Wies. Doomed program points. *Formal Methods in System Design*, 2010.
19. D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.
20. M. Janota, R. Grigore, and M. Moskal. Reachability analysis for annotated code. In *SAVCBS*, 2007.
21. R. M. Karp. Reducibility among combinatorial problems. In *Symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.
22. K. R. M. Leino, T. D. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 55(1-3):209–226, 2005.
23. K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: design and logical encoding. In *TACAS*, 2010.
24. P. Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, 2008.
25. M. Schäf. Bixie: Find contradictions in java code. http://www.csl.sri.com/bixie-ws/, 2014.
26. M. Schäf. Gravy website. https://github.com/martinschaef/gravy, 2014.
27. J. P. M. Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.
28. A. Tomb and C. Flanagan. Detecting inconsistencies via universal reachability analysis. In *ISSTA*, pages 287–297, 2012.
29. X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *SOSP*, pages 260–275. ACM, 2013.