Infeasible Code Detection

Cristiano Bertolini[1], Martin Schäf[1] and Pascal Schweitzer[2]

[1] United Nations University, IIST, Macau
[2] Australian National University

Abstract. A piece of code in a computer program is infeasible if it cannot be part of any normally-terminating execution of the program. We develop an algorithm for the automatic detection of all infeasible code in a program. We first translate the task of determining all infeasible code into the problem of finding all statements that can be covered by a feasible path. We prove that in order to identify all coverable statements, it is sufficient to find all coverable statements within a certain minimal subset. For this, our algorithm repeatedly queries an oracle, asking for the infeasibility of specific sets of control-flow paths.
We present a sound implementation of the proposed algorithm on top of the Boogie program verifier utilizing a theorem prover to provide the oracle required by the algorithm. We show experimentally a drastic decrease in the number of theorem prover queries compared to existing approaches, resulting in an overall speedup of the entire computation.

1 Introduction

Static analysis allows us to detect undesired behavior of a program before it is executed or even compiled. A particular application of static analysis is to identify code fragments that show *only* undesired behavior. Tools implementing this approach detect code which is never part of an execution that terminates normally. We refer to this type of code as *infeasible code*. The terminology infeasible code is justified as follows: an execution is infeasible if it does not terminate normally (note that we do not model error states and thus, any terminating execution terminates normally). A path is infeasible if all its executions are infeasible. And code is infeasible if it only occurs on infeasible paths. Compared to unreachable code, where no feasible path *ends* in the considered code, infeasible code is a more general concept as it only requires that no feasible path *contains* the considered code.

Subsets of infeasible code are, for example, found by the static analyzers in modern development environments such as Eclipse. These tools detect simple, yet common errors such as guaranteed null pointer dereference, use of uninitialized variables, or unreachable program fragments. In practice, finding such type of errors is one of the most frequent applications of static analysis. Thus, improving the detection of infeasible code is a worthwhile problem.

An intriguing property of infeasible code is that it can be detected without prior knowledge of the desired program behavior or it's environment (e.g., the

possible input values). If a piece of code is infeasible, it will stay infeasible even if the context is subsequently restricted by other means such as adding guarding statements or specifying admissible input values. Thus, infeasible code can be detected while typing the program, and infeasible code can only be eliminated by changing the code fragment itself and not by changing other code.

Recently, new static analysis approaches have emerged that use formal methods to prove the presence of infeasible code [11,14]. They prove a particular program statement to be infeasible code by encoding all paths passing through the statement in a logic formula. The formula is satisfiable if there exists a normally terminating execution of the program following one of these paths. The benefit of proving the presence of infeasible code with this approach is that it does not produce false warnings [12].

To detect *all* infeasible code in a program, this method is repeatedly applied to different program statements. Each application invokes a theorem prover query and is thus computationally expensive. It is possible to query the infeasibility of several statements simultaneously, in order to reduce the number of queries. Minimizing the number of queries can help us to devise more efficient ways to process the entire code, however, using more complex queries may be computationally more expensive. Thus the following question arises:

What is an efficient strategy to detect all infeasible statements in a given program?

In this paper, we show that the problem of identifying all infeasible code can be expressed as a set cover problem on a minimal subset of program statements. We then show that the problem of detecting infeasible code is equivalent to proving the non-existence of a feasible path cover for this subset of statements in the control-flow graph of the program. We further show that a feasible path cover of this set covers all feasible statements in the program. This in particular shows how the feasibility or infeasibility of every statements can be determined from the feasibility information on the subset of statements. We present a query optimal greedy algorithm to compute a feasible path cover and a sound implementation. The implementation uses a theorem prover (we use Z3 [6]) as an oracle to check the existence of a feasible control-flow path in a particular set. We show experimentally that the proposed method is more efficient in terms of oracle calls and computation time than existing approaches.

Related Work. Numerous approaches exist that, among other things, show infeasible code to be faulty. E.g., when a test case executes infeasible code it must reveal an error. Many of these approaches suffer from false warnings or require a strong user interaction (e.g., [8,19]). Moreover, the approaches do not prove code to be infeasible. We restrict the discussion to static error detection approaches that detect statements from which a good state is unreachable. Findbugs [13] detects contradicting control-flow using pattern matching. It detects statements similar to infeasible code, however, pattern matching is neither sound nor complete.

Encoding the feasibility of control-flow paths as a logic formula goes back to the idea of predicate transformers [7]. An algorithm that uses formal methods to prove that a statement cannot be *reached* by a feasible execution is presented in [14]. Unreachable code is a special type of infeasible code. An algorithm to detect if a particular control location is never passed by a feasible execution is developed in [11]. They query a theorem prover whether there exists some feasible path containing a particular location. Their approach detects doomed locations while our approach detects infeasible statements. By inserting auxiliary locations, the two approaches can be reduced to each other. However, one of the central insights of this paper is that, in order to find all statements occurring on feasible paths, it is in general *not* sufficient to check only control locations.

In [12] the algorithm from [11] is extended to an algorithm that detects all such locations. They present a strategy to minimize the number of theorem prover queries to detect infeasible code on a loop-free abstraction of the program. In this paper, we present a more general type of query. This type of query allows us to check infeasibility of several statements simultaneously. Moreover, we prove that it is possible to determine all infeasible statements by only checking a minimal subset of all statements. Without the use of auxiliary locations, our proof *cannot* be translated to suit the approach using locations. Any direct proof for program locations seems to require further properties of the program. We show our approach, which also works for programs with loops, further reduces the number of queries and prove that our approach is query optimal.

Algorithms to compute feasible path covers or sets of infeasible control-flow paths have been proposed in software testing (e.g. [4]). These approaches either require an executable program, or over-approximate the feasible path cover.

Organization of the paper. In Section 2 we give some examples of infeasible code. In Section 3 we formalize the notion of infeasible code and show that it's detection can be expressed as a path cover problem. In Section 4 we then present an algorithm to check the existence of a feasible path cover of a control-flow graph that uses an oracle to check if a given set of paths is infeasible. In Section 5 we show how this oracle can be realized using weakest liberal preconditions. In Section 6 we present a prototype implementation of our algorithm and in Section 7 we experimentally compare this implementation with existing implementations in terms of theorem prover calls and computation time.

2 Examples of Infeasible Code

Figure 1 provides 4 example programs that demonstrate the usefulness of infeasible code detection. In `ex01`, line 3 is infeasible because any execution passing that line will loop forever. This example shows that infeasible code also refers to code that does neither reach a normal terminating state nor violates an assertion. However, infeasible code detection only detects non-termination if any execution entering the loop must run forever.

The program `ex02` has infeasible code in line 3. It shows that code can be infeasible because its execution always causes an error at some later stage.

```
1 void ex01(int x) {
2   while (x>0) {
3     x=(x/2)+1;
4   }
5 }
```

```
1 void ex02(C a, int x) {
2   if (a==null)
3     x=-1;
4   a.toString();
5 }
```

```
1 void ex03() {
2   int a, b;
3   a=1;
4   if (a>0) b=1;
5   a=b;
6 }
```

```
1 void ex04(C a) {
2   int y=0;
3   if (a!=null)
4     y=1;
5   if (y==0)
6     a.toString();
7 }
```

Fig. 1. The example programs show different possible causes for code to become infeasible.

Indeed, on any execution passing line 3, the reference `a` is guaranteed to be `null` and thus, the program terminates abnormally in line 4. Note that the only infeasible statement in this program is line 3. All other statements are part of feasible executions.

In practice, we use automatically generated assertions to guard pointer dereferences and other properties. In general, the approach presented in this paper can be used with arbitrary safety properties. E.g., we can use infeasible code detection for definite-assignment analysis and encode the property that every variable must be written once before it is read by using helper variables and assertions. To this end, we can show that in `ex03`, the variable `b` is initialized on any feasible path. In contrast, the Java compiler rejects this program claiming `b` might be not initialized if `a` is not positive at line 4. We can encode other properties, such as array-bound checking, or locking behavior in the same way.

A more complex example of infeasible code is given in `ex04`. Any path containing line 6 is either infeasible because the conditional evaluates to false, or `null` is dereferenced. For code to be infeasible, it is not necessary that all paths are infeasible for the same reason or diverge at the same control location.

3 Infeasible Code, Effectual Sets, and Path Covers

A program is defined by a control-flow graph $\mathcal{P} = (S, \delta, \Sigma)$. A control-flow graph is a connected directed graph where S is the set of control locations and Σ is the set of instructions in the program. A program statement $st = (s, inst, s') \in \delta$ is an instruction $inst \in \Sigma$ at a control-location s whose execution ends in a control-location s'. The transition relation $\delta \subseteq S \times \Sigma \times S$ represents the set of statements in $\mathcal{P}$. W.l.o.g., we assume that the program has a unique source and a unique sink node. A path from a node s_1 to a node s_{k+1} in $\mathcal{P}$ is a sequence of statements $\pi = st_1 \dots st_k = (s_1, inst_1, s_2) \dots (s_k, inst_k, s_{k+1})$, s.t. $st_1, \dots, st_k \in \delta$.

Note that, as customary in the context of control flow graphs, a path may use vertices repeatedly. A *complete* path is a path that starts in the source node and ends in the sink node of the graph. Throughout this paper, unless stated otherwise, the word path always refers to a complete path.

Infeasibility. We assume that the semantics of a statement st is given by a weakest liberal precondition operator $P = wlp(st, Q)$, s.t. any execution of st starting in a state satisfying P results in a state satisfying Q or does not terminate normally [7]. We say an execution does not terminate normally if it blocks, either because a conditional statement is not satisfied or it crashes because an (possibly implicitly) assertion is violated, or it runs forever. We extend the weakest liberal precondition from statements to paths in the obvious way.

Definition 1. *Given a program $\mathcal{P} = (S, \delta, \Sigma)$, a path π in $\mathcal{P}$ is infeasible if $wlp(\pi, false) = true$.*

Here, $true$ denotes the set of all possible states and $false$ the empty set of states. Note that we do not take into account the reason for paths being infeasible. We are only interested in the fact that their executions do not terminate normally.

In general, not every control-flow path in a genuine program is feasible. E.g., control-flow paths may be infeasible because of complex conditional branching. Only if a statement is not part of any feasible execution we call it infeasible code.

Definition 2. *Given a program $\mathcal{P} = (S, \delta, \Sigma)$, a statement $st \in \delta$ is infeasible code, if there is no feasible path π in $\mathcal{P}$ that contains st.*

There are two reasons why a statement can be infeasible code. One is that it is not part of any (terminating) execution, the other is that it is only part of executions that terminate in an error state.

Effectual sets. For the detection of infeasible code, it is not necessary to consider all edges (statements) in a control-flow graph. It suffices to focus on a minimal subset of edges of which each control-flow path contains at least one. As shown in [3, 12], this set can be identified using a partial order over control-flow edges. The minimal subset can be used to decide whether there is infeasible code. However, for our purpose of determining *all* infeasible code, there are examples of control flow graphs where these sets are not sufficient (see [2] for an example).

Definition 3. *Given a program $\mathcal{P}$ and two statements $st, st' \in \delta$, we write $st \preceq st'$ if every complete or infinite path π in $\mathcal{P}$ that contains st also contains st'.*

We remark that in acyclic graphs the defined relation coincides with the one used in [12]. The relation $\preceq$ is reflexive and transitive, therefore the relation $\simeq = \preceq \cap \preceq^{-1}$ is an equivalence relation. We denote by $[st]$ the equivalence class of a statement st under $\simeq$. We say that $[st] \preceq [st']$ if and only if $st \preceq st'$.

For a set $\delta' \subseteq \delta$ we define $\mathrm{cov}(\delta')$ to be the maximal number of elements of δ' contained in a (not-necessarily feasible) path.

We call a set $\delta' \subseteq \delta$ *effectual* if it is a maximal set of statements that are all minimal w.r.t. $\preceq$, such that for any two distinct statements $\mathit{st}, \mathit{st}' \in \delta'$ we have $\mathit{st} \notin [\mathit{st}']$. We will usually denote effectual sets by $\mathcal{F}(\delta)$.

Path covers. A *path cover* of the program $\mathcal{P} = (S, \delta, \Sigma)$ is a set of paths such that each statement in δ is contained in at least one of the paths. A path cover is feasible if it contains only feasible paths. Path covers and effectual sets are the key element to our method of determining infeasible code.

Theorem 1. *Let $\mathcal{P} = (S, \delta, \Sigma)$ be a program and $\mathcal{F}(\delta) \subseteq \delta$ an effectual set. Program $\mathcal{P}$ has no infeasible code, if and only if there is a feasible path cover of $\mathcal{F}(\delta)$.*

Proof. "$\Rightarrow$": If $\mathcal{P}$ has no infeasible code, every statement $\mathit{st} \in \delta$ is part of some feasible path π_{st}. The set $\bigcup_{\mathit{st} \in \mathcal{F}(\delta)} \pi_{\mathit{st}}$ is a feasible path cover of $\mathcal{F}(\delta)$.

"$\Leftarrow$": suppose there is a feasible path cover of $\mathcal{F}(\delta)$. Let $\mathit{st} \in \delta$ be a statement. Since $\mathcal{F}(\delta)$ is maximal, there is a statement $\mathit{st}' \in \mathcal{F}(\delta)$ such that $\mathit{st}' \preceq \mathit{st}$. Since there is a feasible path cover of $\mathcal{F}(\delta)$, there is a feasible path that contains st'. Since $\mathit{st}' \preceq \mathit{st}$ this path also contains st. Thus every statement is contained in some feasible path and $\mathcal{P}$ has no infeasible code. □

The theorem shows that the problem of detecting infeasible code can be understood as a path cover problem on an effectual set. By definition, the code of two equivalent statements $\mathit{st} \simeq \mathit{st}'$ is either for both infeasible or for neither. Thus, from knowing for each statement in an effectual set whether it is infeasible code, we can easily identify all minimal statements that are infeasible code. In the following we show that in reducible control flow graphs we can even identify all infeasible code. A control flow graph is reducible if removing all its back edges yields an acyclic graph. Recall that any path that contains a back edge has a loop that contains the back edge. We first show this for acyclic control flow graphs.

Lemma 1. *Let $\mathcal{P} = (S, \delta, \Sigma)$ be an acyclic program. A statement $\mathit{st} \in \delta$ is infeasible code, if and only if every statement st' which is minimal with respect to $\preceq$ and for which $\mathit{st}' \preceq \mathit{st}$ holds is infeasible.*

Proof. If there is a feasible statement st' with $\mathit{st}' \preceq \mathit{st}$, there exists a feasible path which contains st' and therefore also contains st. Thus, st is feasible.

To show the other direction, we define a statement st to be *bad* if every minimal element st' with $\mathit{st}' \preceq \mathit{st}$ is infeasible but st itself is feasible. We need to show that there are no bad statements. For the sake of contradiction, suppose st is a bad statement that is minimal among all bad statements. Let st_1 be some minimal element with $\mathit{st}_1 \preceq \mathit{st}$. By our assumption st_1 is infeasible. Let π_1 be an infeasible path that contains st_1 and therefore necessarily also contains st, see Figure 2. W.l.o.g. we assume that when traversing π_1 from the source to the sink we first encounter st and then encounter st_1 (otherwise we invert the directions of all edges). Since st is feasible, there is a feasible path π_2 that contains st. Since π_2

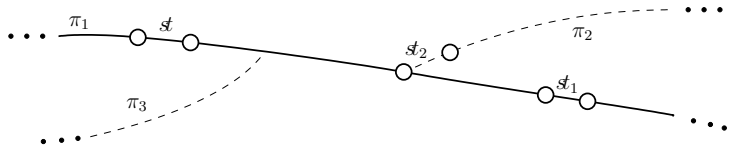

Fig. 2. The elements used in the proof of Lemma 1.

does not contain $\mathfrak{st}_1$, after passing through $\mathfrak{st}$ it must leave the path π_1 before reaching $\mathfrak{st}_1$. Let $\mathfrak{st}_2$ be the first edge on π_2 encountered after passing $\mathfrak{st}$ that is not on π_1. We now show that $\mathfrak{st}_2 \preceq \mathfrak{st}$. Suppose this is not the case, then there is a path π_3 that contains $\mathfrak{st}_2$ but not $\mathfrak{st}$. We construct a path that contains $\mathfrak{st}_1$ but not $\mathfrak{st}$ giving a contradiction: This path is obtained by starting along the path π_3 up to the starting vertex of the edge $\mathfrak{st}_2$ and then following π_1 until the end. This path cannot exist, therefore we conclude $\mathfrak{st}_2 \preceq \mathfrak{st}$. We have $\mathfrak{st} \not\preceq \mathfrak{st}_2$, since there is a path, namely π_1, that contains $\mathfrak{st}$ but not $\mathfrak{st}_2$. Finally note that $\mathfrak{st}_2$ is bad since it is feasible and has the property that all statements $\mathfrak{st}'$ with $\mathfrak{st}' \preceq \mathfrak{st}_2$ in particular fulfill $\mathfrak{st}' \preceq \mathfrak{st}$ and are thus infeasible. This is a contradiction to our minimal choice of a bad statement $\mathfrak{st}$ and shows the theorem. □

For the curious reader, we remark that there is no equivalent theorem for the case of path-vertex covers (see [2] for an example). We now extend the lemma from acyclic graphs to reducible graphs.

Theorem 2. *Let $\mathcal{P} = (S, \delta, \Sigma)$ be a program with a reducible control flow graph.*

1. *A statement $\mathfrak{st} \in \delta$ is infeasible code, if and only if every statement $\mathfrak{st}'$ which is minimal with respect to $\preceq$ and for which $\mathfrak{st}' \preceq \mathfrak{st}$ holds is infeasible.*
2. *If a set of feasible paths covers all feasible statements within an effectual set $\mathcal{F}(\delta)$, then the set of paths covers all feasible statements.*

Proof. (Part 1). Given a reducible control flow graph $\mathcal{P}$ let $\mathcal{P}'$ be the graph obtained by redirecting all endpoints of back edges into the sink. Since $\mathcal{P}$ is reducible, $\mathcal{P}'$ is acyclic. Abusing terminology, for a back edge $\mathfrak{st}$ in $\mathcal{P}$, we refer to the redirected back edge in $\mathcal{P}'$ also as $\mathfrak{st}$.

Claim: For statements $\mathfrak{st}, \mathfrak{st}'$ we have $\mathfrak{st}' \preceq \mathfrak{st}$ in $\mathcal{P}$ if and only if $\mathfrak{st}' \preceq \mathfrak{st}$ in $\mathcal{P}'$. To see the claim it suffices to observe that for every complete or infinite path π in one of the programs $\mathcal{P}$ or $\mathcal{P}'$ that uses a set of statements δ' there is a complete or infinite path π' in the other program that uses a (not necessarily strict) subset $\delta'' \subseteq \delta'$ of the statements. For a path π in $\mathcal{P}$ this is easy to see. We now show this for a path π in $\mathcal{P}'$. Let δ' be the set of statements on the path π. First note that π is finite since $\mathcal{P}'$ is acyclic. If the last edge of π does not correspond to a back edge in $\mathcal{P}$ then π is also a complete path in $\mathcal{P}$. Otherwise, if the last edge of π is a back edge in $\mathcal{P}$ then this edge closes a loop. By repeating this loop indefinitely, we obtain a path whose set of statements is the same as that of π. Either way, we obtain a path in $\mathcal{P}$ with the desired properties, showing the claim.

In $\mathcal{P}'$ we define a complete path to be feasible if its projection to $\mathcal{P}$ is a subgraph of some (complete) feasible path of $\mathcal{P}$. With this definition, a statement st is feasible in $\mathcal{P}$ if and only if it is feasible in $\mathcal{P}'$. Moreover, having now an acyclic control flow graph, Theorem 2 applies. Since feasibility of a statement and the "$\preceq$" relation are equivalent in $\mathcal{P}$ and $\mathcal{P}'$, Part 1 of the theorem follows. (Part 2). To show Part 2 of the theorem let st be feasible code. Then, by the first part of the theorem, there is a minimal element $\mathit{st}' \preceq \mathit{st}$ that is feasible code. Any path cover that covers all feasible statements in an effectual set $\mathcal{F}(\delta)$ covers st'. A path that contains st' also contains st and the theorem follows. □

The proof of the theorem also provides us with a method to compute the relation "$\preceq$": Indeed, to compute the relation $\preceq$ of a program $\mathcal{P}$ we first construct the acyclic program $\mathcal{P}'$ obtained by redirecting back edges into the sink. Since the relation $\preceq$ described in [12] then coincides with the relation defined in this paper, we can then apply the method described in [12] for the computation of $\preceq$.

To make use of the connection between infeasible code, effectual sets, and path covers, we first design an efficient path cover algorithm.

4 A Path Cover Algorithm

In this section we describe an algorithm that finds a feasible path cover for a given reducible control flow graph. The feasible paths are not given explicitly. We rather assume an oracle answers queries from which we can infer which edges of the graph are coverable by some feasible path. Intuitively, we need to optimize our query strategy towards a method that quickly dismisses large portions of the graph as coverable, allowing us to focus on edges that are uncoverable.

Abstractly we have the following model: We are given a control-flow graph $\mathcal{P}$ and repeatedly query for the non-existence of a feasible path with certain properties. We assume an oracle is available that provides us with an answer that either proves that no feasible path with the desired properties exists, or with a counterexample in form of a feasible path that possesses the required properties.

In more detail, the oracle answers the following type of query: For a specified set of nodes δ', and specified positive integers $\ell, k \in \mathbb{N}$ with $\ell \leq k$, does *no* feasible path exist that contains at least ℓ, and at most k elements of δ'? We assume this Constrained Path Infeasibility query is answered by a call to the function $CPI(\delta', \ell, k)$. In Sections 5 and 6 we explain why we use specifically queries of this type and how to realize an oracle that answers them.

Being able to query for a path that contains at least a certain number of edges from a specified subset allows us to adapt the greedy algorithm to our scenario. The standard greedy algorithm for the set cover problem repeatedly chooses a set that covers a maximal number of previously uncovered elements. A classic result by Johnson [15] shows this algorithm to be an $O(\log(n))$ approximation in terms of the number of sets used, and this is best possible, unless $P = NP$ [18].

Description of the algorithm. The path cover algorithm PCA (Algorithm 1) takes as input a reducible control flow graph and outputs a set of feasible paths that

Algorithm 1 Path Cover Algorithm PCA

Input: $\mathcal{P} = (S, \delta, \Sigma)$: A reducible control flow graph.
Output: A set of feasible paths that cover all feasible code of $\mathcal{P}$.

```
    compute an effectual set F(δ)
    k ← cov(F(δ))
    δ' ← F(δ)
    while δ' ≠ {} do
 5:   k ← min{k, cov(δ')}
      query CPI(δ', ⌈k/2⌉, k)
      if the query returned a path π then
        let E(π) be the statements on the path π
        δ' ← δ' \ E(π)
10:   else
        if k = 1 then
          return all paths that were reported by queries
        else
          k ← ⌊k/2⌋
15:     end if
      end if
    end while
    return all paths that were reported by queries
```

cover all feasible code. The algorithm starts by computing an effectual set $\mathcal{F}(\delta)$. It maintains a subset δ' of $\mathcal{F}(\delta)$ which at any point in time contains all elements of $\mathcal{F}(\delta)$ that cannot be covered by a feasible path. It repeatedly queries the oracle, and if returned a path, removes the statements on that path from δ'. The algorithm also maintains an integer k which is an upper bound on the number of statements in δ' that may lie simultaneously on a feasible path.

Theorem 3. *Let $\mathcal{P} = (S, \delta, \Sigma)$ be a program with a reducible control flow graph that has a unique sink, a unique source, and an unknown set of feasible paths. Let $\mathcal{F}(\delta)$ be an effectual set. Algorithm 1 returns a set of feasible paths that covers all feasible code of $\mathcal{P}$. If K is the size of the smallest set of feasible paths that covers all coverable elements in $\mathcal{F}(\delta)$, then Algorithm 1 performs at most $O(K \cdot \log(\mathrm{cov}(\mathcal{F}(\delta))))$ queries.*

A proof of Theorem 3 is given in the extended version of this paper [2]. As mentioned previously, the set cover problem cannot be approximated with an approximation ratio of $o(log(n))$ unless $P = NP$ [18]. In our algorithm we made use of the parameter $\mathrm{cov}(\mathcal{F}(\delta))$ to get a finer analysis of the number of queries.

In general every set cover problem can be modeled as a path cover problem on a graph with unique sink and unique source: Indeed, by taking the transitive closure of a directed path, any subset of the edges of the original path can be chosen to be simultaneously on a feasible path. The inapproximability result thus applies to our path cover problem as well, and in this sense our algorithm is optimal with respect to the number of queries. However, control-flow

graphs are not arbitrary graphs, and it might be possible to improve beyond the inapproximability ratio by using properties inherent to control-flow graphs.

5 Checking Constrained Path Infeasibility

In this section we explain how to construct the oracle $CPI(\delta', \ell, k)$ that checks for a program $\mathcal{P} = (S, \delta, \Sigma)$ whether there exists no feasible control-flow path π that contains at least ℓ and at most k statements in $\delta' \subseteq \delta$. The call CPI returns such a feasible path π if it exists, otherwise it returns the empty path. We first devise a technique that allows us to modify any program so that this type of query can be answered. In particular with our modification, each query translates into a formula whose validity is equivalent to the non-existence of such a path. We use the concept of *reachability variables* introduced in [11] to monitor which statements are involved in an execution of a program $\mathcal{P}$.

So far, our approach is independent of a particular programming language. In the following, we require that our programming language is expressive enough to support variables with numeric types and assignment statements.

Let $\mathcal{P} = (S, \delta, \Sigma)$ be a program and $\delta' \subseteq \delta$. For each statement $st \in \delta'$, we create an auxiliary *reachability variable* r_{st} in $\mathcal{P}$ which is initially zero. We replace a statement $st \in \delta'$ by the sequence $r_{st} := 1; st$. That is, every time st is executed r_{st} is assigned to one. Thus, after the execution of a path π in $\mathcal{P}$ the sum $\sum_{st \in \delta'} r_{st}$ is the total number of statements in δ' that occur on π.

Having introduced the reachability variables, the existence of a feasible path with at least ℓ and at most k statements from δ' in a program $\mathcal{P}$ can be checked using the weakest liberal precondition wlp of $\mathcal{P}$ and the postcondition $\neg(\ell \leq \sum_{st \in \delta'} r_{st} \leq k)$. This leads to the following theorem:

Theorem 4. *Let $\mathcal{P} = (S, \delta, \Sigma)$ be a program, $\delta' \subseteq \delta$ a set and ℓ, k integers with $1 \leq \ell \leq k \leq |\delta'|$. The program $\mathcal{P}$ has no feasible path π, s.t. π contains at least ℓ and at most k statements from δ' if and only if the formula*

$$CPI(\delta', \ell, k) := wlp(\mathcal{P}, \neg(\ell \leq (\sum_{st \in \delta'} r_{st}) \leq k))$$

is universally valid in the program augmented with reachability variables.

A proof of Theorem 4 is given in the extended version of this paper [2]. In principle, we could design the query function CPI to work for arbitrarily complicated properties that are based on the reachability variables. In particular for any first order formula over the reachability variables we can obtain a theorem analogous to the one just presented. However, for actual implementations of the oracle, the complexity of the queries may alter the query response time, as we show in our experiments in Section 7. Our choice of query type is motivated by the fact that linear inequalities in practice can be handled well by theorem provers, and that this type of query suffices to construct an algorithm that is optimal with respect to the number of queries.

Up to this point neither the algorithm PCA nor the realization of CPI perform any abstraction. That is:

Lemma 2. *Given a sound, complete implementation of CPI, algorithm PCA is a sound and complete method to detect all infeasible code in a program.*

The proof of Lemma 2 follows directly from Theorem 3 and 4. However, an implementation of CPI needs to compute the weakest liberal precondition of a program, which is an undecidable problem in general. We will thus not be able to design an implementation of CPI which makes algorithm PCA simultaneously sound and complete. In the following, we describe a sound implementation of PCA which uses a sound *wlp* computation presented in [11].

6 Implementation

We now describe our implementation of a sound tool that detects infeasible code. Our implementation takes a Boogie program [16] as input, augments it with reachability variables, then applies the algorithm described in Section 4 and returns a subset of the infeasible statements of the program. We implement the constrained path infeasibility queries CPI using the sound over-approximation of the weakest liberal precondition presented in [11]. Soundness, in this case, means that for any path that has a feasible execution in the original program, there is a corresponding path with a feasible execution in the abstract program. If our translation were not sound, we might report infeasible paths that have feasible executions in the original program. We stress that this notion of soundness is dual to the notion of soundness used in verification.

Computing a formula representation of an over-approximation of the weakest (liberal) precondition of a program is a common technique [9, 10, 12, 14, 17]. It involves two steps: 1.) compute a loop-free abstraction of the input program, 2.) compute a formula representation of the weakest (liberal) precondition of the loop-free program.

Compute a loop-free abstraction. Given a Boogie program, we use the abstract loop unrolling presented in [12]. Loops are unrolled three times. The first unrolling represents the first iteration of the loop. The third unrolling represents the last iteration of the loop. Any other iteration is represented by the second, abstract unrolling. To retain soundness of the abstraction, non-deterministic assignments to all variables modified by the loop are added before and after the abstract unrolling. This abstraction is sound as it preserves the set of feasible executions of the original program. A proof of soundness is given in [12].

Compute weakest liberal precondition. For the loop-free program we perform a single-assignment transformation (e.g., [5]). We introduce an auxiliary variable for each assignment statement such that each variable is only written once. The resulting program is passive in a way that it does not change its state.

For a program $\mathcal{P}$, we denote the result of introducing reachability variables, eliminating loops, and altering the code to single assignment form by $trans(\mathcal{P})$. For $trans(\mathcal{P})$, we can compute a formula representing the weakest liberal precondition straightforwardly (see, e.g., [1, 11, 14, 17]).

In our implementation we can now use an automated theorem prover to check the satisfiability of the negation of the formula CPI from Theorem 4:

$$VC(\delta', \ell, k) := \quad \ell \leq \sum_{st \in \delta'} (r'_{st}) \leq k \wedge wlp\left(trans(\mathcal{P}), false\right),$$

where r'_{st} refers to the last incarnation introduced by the single assignment transformation. If the theorem prover is able to prove VC unsatisfiable for given δ', ℓ, k we know that there exists no feasible path in the over-approximation of the program that contains at least ℓ and at most k statements in δ', and from the soundness of *trans* it follows that there is no feasible path in the original program either. Together with the algorithm PCA, this gives us a sound tool to detect infeasible code. We now compare this tool with existing techniques.

7 Evaluation

In this section we compare 3 algorithms to detect infeasible code in terms of theorem prover calls and computation time. We compare *PCA* presented in this paper, *Doomed* [11], which checks a minimal set of statements on loop-free programs, and *DoomedCE* [12], which performs the same checks as Doomed but utilizes the counterexamples emitted by the theorem prover to avoid redundant queries. Note, that DoomedCE can be considered a special case of PCA, where $\ell = 1$ and k is set to the number of statements.

We do not need to consider detection rate since all algorithms use the same sound weakest liberal precondition computation (which is part of the Boogie program verifier) and thus report the same infeasible code. For a comparison of detection rate with, e.g., Findbugs we refer to [12].

Experimental Material. To evaluate the performance of our algorithm, we use a set of 100 randomly generated Boogie procedures. We use generated programs because this allows us to vary the number of diamond shapes in the control-flow graph freely and no translation from a high-level language to Boogie that preserves the set of feasible executions is required. Existing translations from high-level languages into unstructured languages are not suitable for our algorithms since they over-approximate the set of infeasible executions to retain soundness w.r.t. partial correctness proofs. The threat to validity which arises from generated programs is discussed at the end of this section.

Each generated procedure has between 150 and 1500 lines of code and modifies up to 50 unbounded integer. The body of a procedure contains a sequence of 5 to 10 conditional choices (diamond shapes) each with up to 5 nested conditional choices or loops. Besides the nested conditionals and loops, each block has between 3 and 5 statements. The statements are either assignments of expressions to variables, or assertions. All experiments are run several times on a standard laptop computer with ample memory.

Comparison of the algorithms. All algorithms identify 23997 out of 116925 statements to be infeasible code. PCA uses 1383 theorem prover calls and a total time

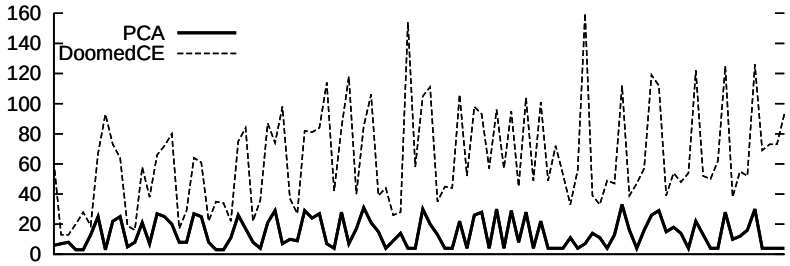

Fig. 3. Comparison of the number of queries. The x-axis ranges over the tested procedures, sorted by increasing length, the y-axis indicates the total number of theorem prover calls.

of 324 sec to identify all infeasible code; Doomed uses 8942 theorem prover calls and 1309 seconds; DoomedCE uses 6365 queries theorem prover calls and 948 seconds. By its definition Doomed uses one query for each element in the effectual set. The algorithm DoomedCE covers the effectual set by querying 71% of the elements, and PCA is able to cover the set by querying 15% of the elements. In terms of computation time, DoomedCE needs 72% of the computation time of Doomed, and PCA needs 24% of the time used by Doomed.

Figure 3 compares the number of queries of PCA and DoomedCE in more detail. As expected, we can see that the number of queries for PCA is drastically lower than the number of queries for DoomedCE.

Figure 4 compares the computation time per procedure for algorithms PCA and DoomedCE. We can see that, even though PCA uses more expensive oracle queries than DoomedCE, there is a significant speedup due to the reduced number of queries.

Threats to Validity. The randomly generated programs is the main internal threat to validity. However, they allow us to control the shape of the control-flow graph and, in particular, the number of diamond shapes which is important to show the benefit of the path cover algorithm. The generated programs are of a very simple nature. They do not use complex types or a heap. For all algorithms, each query reasons about the set of paths in the program, these queries will become proportionally more expensive if reasoning about a single path becomes more expensive. Thus, we expect that our observations will still hold for real programs. However, in our future work, we have to evaluate if control-flow graphs of this complexity occur in practice.

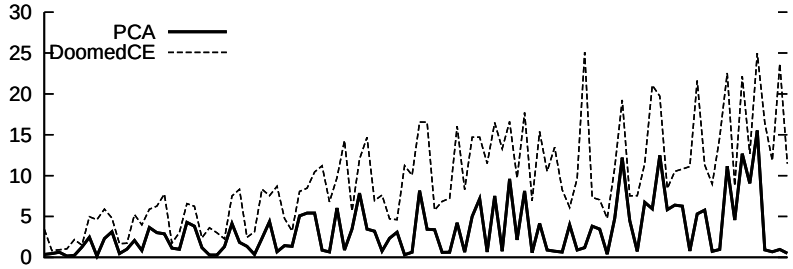

Fig. 4. Comparison of the computation time per program for PCA and DoomedCE. The x-axis ranges over the tested procedures, sorted by increasing length, the y-axis displays computation time in seconds.

Another internal threat to validity of our experiments arise form the used theorem prover. In our experiments we use only Z3. Other theorem provers may have a different efficiency for our kind of query (i.e., the linear inequalities). However, to avoid using a slow integer theory solver, we could alternatively encode the sum of reachability variables as a boolean formula.

External threats to validity arise from the needed translation from some high-level language to Boogie.

Discussion of the results. The drawback of DoomedCE compared to PCA is its inability to influence the counterexamples produced by the theorem prover. A counterexample may cover many statements that have already been covered by previously found counterexamples. Yet, forcing the theorem prover to provide more useful counterexamples comes at the price of longer query times. How high this price is, depends on the program structure. For example, if a program consists of sequential but independent parts, there are feasible paths that provide useful information in all parts simultaneously.

The effectiveness of searching useful counterexamples is also influenced by the properties we check. It is to be expected that there are properties for which the queries presented in this paper are significantly more expensive than, e.g., the queries used in DoomedCE. For these cases, algorithm PCA may require noticeably more computation time than DoomedCE. However, our experiments indicate that, at least for null pointer dereference and definite assignment analysis, the presented approach yields a significant performance improvement.

As mentioned earlier, algorithm DoomedCE can be considered as a special variant of PCA. Both algorithms are part of a family of algorithms obtained by varying the variables ℓ and k used to call CPI. This indicates that, even though

the presented approach is query optimal, there is still room for optimization to achieve optimal computation times.

8 Conclusion

We have shown that the detection of infeasible code can be seen as a set cover problem and, more importantly, that covering *all* feasible statements in an effectual set determines all feasible statements in a program.

We presented an algorithm that detects all infeasible code in a program which uses an optimal number of queries. Using our implementation, for various applications of infeasible code detection, we have experimentally shown a significant decrease in the computation time when compared to existing methods.

For our future work we see two promising directions. We will incorporate the path cover algorithm directly in a theorem prover. In fact, the presented algorithm can be seen as a strategy to force a theorem prover to search for a particular counterexample. Thus, implementing this directly in a theorem prover may lead to performance improvements by allowing reuse of information more efficiently.

Another direction of future work is the development of different strategies to realize CPI based on different approximations of the weakest liberal precondition. A sound implementation of CPI that is only required to preserve all feasible executions of a program (in contrast to verification contexts, where all infeasible executions must be preserved) can use a coarser approximation and may result in a significant performance improvement while maintaining a reasonable detection rate. For some properties, a very coarse abstraction of *wlp* may still be sufficient to identify most infeasible code.

The main usefulness of the presented approach is that it detects some common types of errors. It works without user interaction and, in particular, without any false warnings. Perhaps the most intriguing aspect is that it has the potential to become fast enough to be applied while typing.

Acknowledgements. This work is supported by the project ARV funded by Macau Science and Technology Development Fund, by the National Research Fund, Luxembourg, and the Marie Curie Actions of the European Commission.

References

1. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '05, pages 82–87, New York, NY, USA, 2005. ACM.
2. C. Bertolini, M. Schäf, and P. Schweitzer. Infeasible code detection. Technical Report 455, United Nations University, IIST, November 2011.
3. A. Bertolino. Unconstrained edges and their application to branch analysis and testing of programs. *Journal of Systems and Software*, 20:125–133, February 1993.

4. A. Bertolino and M. Marré. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Trans. Softw. Eng.*, 20:885–899, 1994.
5. 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. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991.
6. 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.
7. E. W. Dijkstra. *A discipline of programming / Edsger W. Dijkstra*. Prentice-Hall, Englewood Cliffs, N.J., 1976.
8. P. Emanuelsson and U. Nilsson. A comparative study of industrial static analysis tools. *Electron. Notes Theor. Comput. Sci.*, 217:5–21, July 2008.
9. J.-C. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Heidelberg, 2007. Springer.
10. R. Grigore, J. Charles, F. Fairmichael, and J. Kiniry. Strongest postcondition of unstructured programs. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, FTfJP '09, pages 6:1–6:7, New York, NY, USA, 2009. ACM.
11. J. Hoenicke, K. R. Leino, A. Podelski, M. Schäf, and T. Wies. It's doomed; we can prove it. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 338–353, Berlin, Heidelberg, 2009. Springer.
12. J. Hoenicke, K. R. Leino, A. Podelski, M. Schäf, and T. Wies. Doomed program points. *Form. Methods Syst. Des.*, 37:171–199, December 2010.
13. D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 132–136, New York, NY, USA, 2004. ACM.
14. M. Janota, R. Grigore, and M. Moskal. Reachability analysis for annotated code. In *Proceedings of the 2007 conference on Specification and verification of component-based systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SAVCBS '07, pages 23–30, New York, NY, USA, 2007. ACM.
15. D. S. Johnson. Approximation algorithms for combinatorial problems. volume 9, pages 256–278, Orlando, FL, USA, December 1974. Academic Press, Inc.
16. K. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327, Berlin, Heidelberg, 2010. Springer.
17. K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93:281–288, March 2005.
18. R. Raz and S. Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 475–484, New York, NY, USA, 1997. ACM.
19. N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.