Induction in CEGAR for Detecting Counterexamples

Chao Wang, Aarti Gupta, and Franjo Ivančić
NEC Labs America, Princeton, NJ 08540, U.S.A.
{chaowang,agupta,ivancic}@nec-labs.com

***Abstract*— Induction has been studied in model checking for proving the validity of safety properties, i.e., showing the *absence* of counterexamples. To our knowledge, induction has not been used to refute safety properties. Existing algorithms including bounded model checking, predicate abstraction, and interpolation are not efficient in detecting long counterexamples. In this paper, we propose the use of induction inside the counterexample guided abstraction and refinement (CEGAR) loop to prove the *existence* of counterexamples. We target bugs whose counterexamples are long and yet can be captured by regular patterns. We identify the pattern algorithmically by analyzing the sequence of spurious counterexamples generated in the CEGAR loop, and perform the induction proof automatically. The new method has little additional overhead to CEGAR and this overhead is insensitive to the actual length of the concrete counterexample.**

I. Introduction

Induction techniques have been used in model checking to prove safety properties in a state transition system. A property is called *safety* if it can be refuted by examining a finite computation path of the model. Properties of the form $\mathsf{AG}\,\psi$ (i.e., ψ is an invariant) are an important special case since a general safety property can be reduced to an invariant by a compilation process [1]. Conceptually, one can prove invariant properties by showing that ψ holds in the initial states, and ψ is maintained by the transition relation. SAT based induction methods [2], [3], for instance, rely on the observation that a failing property has a simple path from an initial state to a bad state. An invariant holds if all paths of length k or shorter satisfy ψ, and there is no simple path of length $k+1$ or longer from an initial state. Induction has a clear advantage over other proof methods since it has to consider only paths of a shorter length (up to k in k-induction) whereas bounded model checking (BMC [4]), for instance, needs to check all paths up to a completeness threshold.

However, induction has not been used to refute safety properties, that is, to show that a concrete counterexample exists. Existing methods for finding bugs, including BMC, counterexample guided abstraction refinement (CEGAR [5], [6], [7]), and interpolation [8], are not efficient in the presence of long concrete counterexamples. For example, BMC has been widely regarded as effective in detecting shallow bugs in large models; however, state-of-the-art BMC algorithms handle only up to a few hundred unrollings on typical industrial-scale models. When there are deep bugs, CEGAR and interpolation based methods do not work well either, since they too rely on finding a state-by-state match between the abstract and the concrete counterexamples.

We propose an induction based refutation method for detecting long counterexamples, whose computational overhead is independent of the actual counterexample length. Our main observation is that deep bugs can often be captured by a regular pattern in the counterexample of a family of systems obtained by introducing a parameter to the system under analysis. Instead of looking for a state-by-state match of the abstract counterexample to a particular concrete counterexample, we prove by induction that there always exists a counterexample of that general pattern.

```
(1)  unsigned i , n ;           (1)  bool P = * ;
(2)  ...                        (2)  ...
(3)  n = 10000 ;                (3)  P = * ;
(4)  ....                       (4)  ....
(5)  i = 0 ;                    (5)  P = T ;
(6)  while ( i<=n ) {           (6)  while ( * ) {
(7)    assert ( i<n ) ;         (7)    assert ( P ) ;
(8)    i++ ;                    (8)    P = P ? * : F ;
     }                               }
```

Fig. 1. The original C program and the first Boolean abstraction

Consider the program in Fig. 1, which has a simple and yet representative bug in line 7 (e.g., an array bound violation). Detecting this bug is challenging for all aforementioned methods. For illustration purposes, a standard predicate abstraction procedure would add the following predicates, `P:(i<10000)`, `P1:(i<9999)`, `P2:(i<9998)`, etc. The procedure needs n refinement iterations in order to produce a concrete counterexample. One may argue that CEGAR is not well suited for finding this type of bugs. However, if the loop condition were `(i < n)` and the assertion never failed, then CEGAR is efficient in getting the proof.

Our new method provides complementary strength (good for refutation) to the popular CEGAR style abstraction algorithms (good for proof). In Fig. 1, regardless of the initial value of n, there is an assertion failure in Line 7. Furthermore, the sequence of concrete counterexamples, in terms of the line numbers leading to the failure, is

$$\texttt{(1)(2)(3)(4)(5)} \quad \left\{\texttt{(6)(7)(8)}\right\}^n \quad \texttt{(6)(7)}$$

If we can prove that a counterexample of this regular pattern exists for all $n \geq 1$, then it follows that there exists a counterexample for `n = 10000`.

Although CEGAR is not efficient in detecting long counterexamples, it can be useful in identifying the regular pattern of the counterexample. We present an algorithm to identify the regular pattern of the counterexample, by analyzing the failed counterexample concretization attempts inside the CEGAR loop. The basic idea behind this algorithm is that the regular pattern of a concrete counterexample is often shown in the series of spurious counterexamples encountered in the CEGAR loop. In Fig. 1, for instance, the set of spurious abstract counterexamples produced by the CEGAR procedure have the same regular pattern—they differ from the concrete counterexample only in the number of copies of the recurring segment `(6)(7)(8)`.

The existence of a parameterized counterexample can then be proved by induction: (1) in the base step, we show that a concrete counterexample of the given regular pattern exists for $n = 1$; (2) as an induction hypothesis, we assume that for $n = k$, a concrete counterexample of the given pattern exists. (3) in the induction step, we extend the counterexample for $n = k$ to build a new counterexample for $n = k + 1$, and show that the new counterexample exists. One of our main contributions is proposing a *goal containment* check that is sufficient to establish the induction proof for $n = k + 1$ based on the induction hypothesis for $n = k$. In the goal containment check, we align the common prefixes of the two consecutive counterexamples for k and $k + 1$ and compare their suffixes.

We have implemented the proposed method in a standard CEGAR procedure and conducted experiments on some software examples from the public domain. The results show that when augmented with the new induction based method, CEGAR is able to find some very long counterexamples in nontrivial examples, most of which would have eluded existing model checking methods.

The rest of this paper is organized as follows. After reviewing the related work and introducing the notation, we present the algorithm for identifying a counterexample pattern in Section IV. We present our symbolic method for establishing an induction proof in Section V. We demonstrate the effectiveness of our method through experiments in Section VII, and then conclude this paper in Section VIII.

II. Related Work

Detecting a long counterexample has been a well-known problem in formal verification of both hardware and software systems. In [9], Ho *et al.* attempt to solve the problem by simulating up to a deep state and then searching around that state exhaustively. This technique can be considered as a semi-formal method, in that it combines directed random simulation and model checking. A similar approach was adopted by DART [10] in the context of program verification. Semi-formal methods may miss bugs since they selectively, as opposed to exhaustively, explore the state space.

In [11], Nanshi and Somenzi use guided pseudo-random simulation in the search of a concrete counterexample, where the guidance is provided by synchronous onion rings (SORs [12]), a data structure that implicitly captures all shortest abstract counterexamples. In [13], Bjesse and Kukula present a repeat extender algorithm for counterexample generation within an abstraction refinement loop. They use the abstract counterexample as *backshell lighthouses* without limiting the BMC search to concrete counterexamples of the same length. In [14], Kroening and Weissenbacher propose a similar method targeting counterexamples with loops. After heuristically identifying the loop, they put the assignment statements of the loop body into a closed form representation, and use a SAT solver to calculate a conservative bound on the number of loop iterations. Then, they use BMC to iterate through the loop exactly that number of times and derive a concrete counterexample.

All these CEGAR based methods insist on finding a concrete path from an initial state leading to the bad state, which makes them less scalable when the model is complex and the counterexample is long.

In [15], Seghir and Podelski use *transition abstraction* to shortcut the "transfinite" sequence of refinement steps. They abstract not just states but also the state changes induced by the structured language constructs including `for` and `while` statements. In [16], Ball *et al.* analyze termination of loops without refinement and without well-founded sets and ranking functions. Their method is based on the symbolic reasoning of abstract counterexamples; instead of using induction, they try to identify *must transitions* in the abstract state transition graph using conditions on the structure of the graph.

III. Preliminaries

A. From CDFGs to Models

We represent the model under verification as a tuple $M = \langle S, T, I, S_E \rangle$, where S is a set of states, $T \subseteq S \times S$ is the transition relation, $I \subseteq S$ is the set of initial states, and $S_E \subseteq S$ is the set of error states. Given a set $X = \{x_1, \ldots, x_n\}$ of state variables, each state $s \in S$ is a valuation of the variables in X. A concrete path is a sequence of states $s_1 \ldots s_l$ such that $(s_i, s_{i+1}) \in T$ for all $1 \leq i < l$.

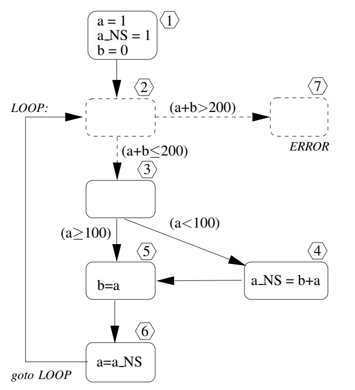
We use a control and data flow graph (CDFG) as the intermediate representation, where CDFGs may be derived from either hardware designs or software programs [17].

Definition 1 *A control and data flow graph (CDFG) is a 5-tuple $G = \langle \mathcal{B}, \mathcal{E}, V, \Delta, \theta \rangle$ such that*

- *$\mathcal{B} = \{b_1, \ldots, b_L\}$ is a finite set of basic blocks, where b_1 is the entry block.*
- *$\mathcal{E} \subseteq \mathcal{B} \times \mathcal{B}$ is a set of edges representing transitions between basic blocks.*
- *V is a finite set of variables that consists of actual design/program variables and the auxiliary variables added for modeling the hardware/software semantics.*
- *$\Delta : \mathcal{B} \to 2^{\mathcal{S}_{asgn}}$ is a labeling function that labels each basic block with a set of parallel assignments. $\mathcal{S}_{asgn}$ is the set of possible assignments.*
- *$\theta : \mathcal{E} \to \mathcal{S}_{cond}$ is a labeling function that labels each edge with a conditional expression. $\mathcal{S}_{cond}$ is the set of possible conditional expressions.*

Figure 2 shows a sample C program and its CDFG. Each rectangle is a basic block. Block 1 is the entry block and block 7 is the error block. Basic blocks are connected with each other by edges, which are labeled by conditional expressions. For example, the transition from block 3 to block 4 is guarded by `(a<100)`. Edges that are not labeled by any condition are assumed to have a true label.

The CDFG is regarded as an explicit representation of the concrete model. To represent the CDFG symbolically as a verification model $M = \langle S, T, I, S_E \rangle$, we add a special Program Counter (PC) variable x_{pc} whose domain is the set



```
void main ()
{
  a = 1;
  a_NS = 1;
  b = 0;

  LOOP:
    if (a + b > 200)
      ERROR

    if (a < 100)
      a_NS = b + a;
    b = a;

    a = a_NS;
  goto LOOP;
}
```

Fig. 2. The control and data flow graph

of basic block indexes $\mathcal{B} = \{b_1, \ldots, b_n\}$. Furthermore, we assume in the model M that $X = \{x_{pc}\} \cup V$ is the set of state variables (V consists of the original variables from the model). If the entry block is $b_1 \in \mathcal{B}$ and the error block is $b_{err} \in \mathcal{B}$, then we have $I = (x_{pc} = b_1)$ and $S_E = (x_{pc} = b_{err})$.

B. Parameterized Counterexamples

An abstract model $\widehat{M}$ can be derived from an over-approximated pair $(\widehat{T}, \widehat{I})$ such that $T \subseteq \widehat{T}$ and $I \subseteq \widehat{I}$. Given a CDFG representation, a natural way of creating an abstract model $\widehat{M}$ is to over-approximate the assignment statements. For instance, if we ignore the assignment statements (by assuming that arbitrary values can be assigned to the left-hand side variables), the remaining control flow graph can be regarded as an abstract model. In this abstract model $\widehat{M}$, an abstract state is represented by the set $\widehat{s}_i = (x_{pc} = b_i)$, where $b_i \in \mathcal{B}$. Since all the guards in $\widehat{M}$ can be either true or false, $\widehat{M}$ has all the behaviors of M and possibly more. An abstract path is a sequence of abstract states $\pi = \widehat{s}_i, \ldots, \widehat{s}_j$.

Definition 2 *A parameterized abstract counterexample ξ is an abstract path of the form*

$$\xi = \widehat{s}_1 \ldots \widehat{s}_{i-1} \ \{ \ \widehat{s}_i \ldots \widehat{s}_j\}^n \ \widehat{s}_{j+1} \ldots \widehat{s}_l \ \ ,$$

such that $\widehat{s}_1 \subseteq \widehat{I}$ and $\widehat{s}_l \subseteq \widehat{S}_E$. The recurring segment is $\pi_r = \widehat{s}_i \ldots \widehat{s}_j$, the finite prefix is $\pi_p = \widehat{s}_1 \ldots \widehat{s}_{i-1}$, and the finite suffix is $\pi_s = \widehat{s}_{j+1} \ldots \widehat{s}_l$. The integer n is the number of copies of π_r.

Any of the three segments in ξ can be empty. For example, if ξ is used to match the concrete counterexample in a state-by-state way, we assume that π_r may be an empty segment.

Given a set Q of states, the post-condition (pre-condition) of Q with respect to T consists of all successors (predecessors) of Q. Formally,

$$\begin{aligned} pre\ (Q) = &\quad \{s \ \mid \exists s' \in Q : (s, s') \in T\} \ , \\ post(Q) = &\quad \{s' \mid \exists s \ \in Q : (s, s') \in T\} \ . \end{aligned}$$

The definition can be extended transitively with respect to a control path $\pi = \widehat{s}_i, \ldots, \widehat{s}_j$ as follows,

$$\begin{aligned} pre^*\ (\pi, Q) = \{s \mid &\ \text{there is a concrete run inside } \pi \\ &\ \text{from } s \text{ to } s' \text{ such that } s' \in (Q \cap \widehat{s}_j)\} \ , \\ post^*(\pi, Q) = \{s' \mid &\ \text{there is a concrete run inside } \pi \\ &\ \text{from } s \text{ to } s' \text{ such that } s \in (Q \cap \widehat{s}_i)\} \ . \end{aligned}$$

Given a control path π, the transitive version of pre-condition can be computed as follows: let $Z = \widehat{s}_j \cap Q$, and then repeatedly compute $Z = \widehat{s}_i \cap pre(Z)$ for all $i \leq j$.

Function $f(V)$ is called *disjunctively decomposable* with respect to the partition $V = V_a \cup V_b$ if

$$f(V) = (\exists V_a . f(V_a, V_b)) \wedge (\exists V_b . f(V_a, V_b)) \ \ ,$$

C. The CEGAR Procedure

Counterexample guided abstraction refinement is an iterative procedure consisting of three phases: abstraction, model checking, and refinement. Algorithm 1 shows the pseudo code of a generic CEGAR procedure. Typically, one starts with a coarse initial abstraction $\widehat{M}$ and applies model checking. If the property holds in $\widehat{M}$, it also holds in M and the property is proved. If the property fails and the model checker returns an abstract counterexample (ACE), a concretization procedure is used to check whether a concrete counterexample (CCE) exists. If a concrete counterexample exists, the property is refuted. Otherwise, the *spurious* counterexample is used during refinement to identify the needed information currently missing in the abstraction.

Algorithm 1 CEGAR(M, ψ)

```
1: $\widehat{M}$ = INITIAL_ABSTRACTION($M, \psi$);
2: while (1) do
3:     $ACE$ = MODEL_CHECKING($\widehat{M}, \psi$);
4:     if ($ACE$ is empty) then
5:         return TRUE;
6:     end if
7:     $CCE$ = CONCRETIZECEX($M, ACE$);
8:     if ($CCE$ not empty) then
9:         return (FALSE, $CCE$);
10:    end if
11:    $\widehat{M}$ = REFINEMENT($M, ACE$);
12: end while
```

One inefficiency of CEGAR in detecting long counterexamples is due to its concretization algorithm. Algorithm 2 is a standard concretization procedure which takes the abstract counterexample $\pi = \widehat{s}_1, \ldots, \widehat{s}_l$ as input. It starts from the error states $\widehat{s}_l$ and repeatedly computes pre-conditions. If the pre-condition q_1 is non-empty, a concrete counterexample has been found. If the precondition becomes empty before i decreases to 1, concretization fails and the abstract counterexample is marked as *spurious*. After the counterexample is declared as spurious, the refinement step follows to improve the abstract model by analyzing the spurious counterexample. The CEGAR loop continues until either the property is proved (no abstract counterexample), or a concrete counterexample is found, or the procedure runs out of computing resources.

It may take a large number of refinement iterations for the CEGAR procedure to produce an abstract counterexample whose length matches the length of the concrete counterexample—before that, all the abstract counterexamples are declared as spurious. If the concrete counterexample has a parameterized pattern, using Algorithm 2 to concretize it is inefficient. Recall that $\xi = \pi_p\{\pi_r\}^n\pi_s$. When ξ contains a large number of copies of π_r, the pre- and post-condition computations over ξ can be expensive.

Algorithm 2 ConcretizeCEX(M, π)

```
1: $i = l$;
2: $q_i = \widehat{s}_i$;
3: while ($i \neq 1$) do
4:     $q_{i-1} = pre(q_i) \cap \widehat{s}_{i-1}$;
5:     if ($q_{i-1} \neq \emptyset$) then
6:         $i = i - 1$;
7:     else
8:         return no concrete counterexample;
9:     end if
10: end while
11: return a concrete counterexample;
```

IV. Identifying Counterexample Patterns

To augment the standard CEGAR procedure, we add an induction proof attempt right after concretization fails, but before refinement. Our procedure tries to identify a regular pattern from the set of abstract counterexamples by analyzing the failed concretization attempts. The counterexample pattern becomes a hypothesis, which subsequently will be validated by an induction step. If we can prove that for all induction parameter values (including the one in the concrete counterexample), an instance of the parameterized counterexample exists, then the property is refuted. If this added induction proof attempt does not succeed, we fall back upon the standard CEGAR loop and continue with refinement.

A. The Recurring Segment

To characterize ξ, we need to identify the head and tail states of π_r from a given abstract path $\pi = \widehat{s}_1 \ldots \widehat{s}_l$. We accomplish this by modifying the standard concretization procedure. We rely on the fact that if the concrete counterexample is an instance of a parameterized abstract counterexample $\pi_p\{\pi_r\}^n\pi_s$, then the CEGAR procedure is likely to generate a series of spurious counterexamples of the following form:

$$\begin{array}{lll} \text{iteration 1:} & \pi & = \pi_p \;\; \pi_r \;\; \pi_s, \\ \text{iteration 2:} & \pi & = \pi_p \;\; \pi_r \;\; \pi_r \;\; \pi_s, \\ \text{iteration 3:} & \pi & = \pi_p \;\; \pi_r \;\; \pi_r \;\; \pi_r \;\; \pi_s, \\ \ldots & & \end{array}$$

the sequence continues until the number of copies of π_r matches the value in the concrete counterexample.

Recall that for a spurious abstract path $\pi = \widehat{s}_1 \ldots \widehat{s}_l$, the concretization procedure in Algorithm 2 will find a *failing index* i such that $1 \leq i < l$ and $pre(q_{i+1}) \cap \widehat{s}_i = \emptyset$. In Fig. 3, for instance, the failing index is i since q_i is empty. In Algorithm 2, once a failing index i is found, π is declared as spurious and the concretization stops.

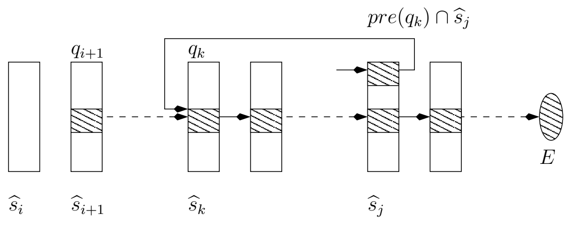


Fig. 3. Backleap to identify the counterexample pattern

We modify Algorithm 2 to allow the search for a potentially longer concrete counterexample by using a "backleap" strategy. The new concretization procedure is given in Algorithm 3, in which the additional steps (with respect to the standard algorithm) are listed in lines 8-19. C_{thres} is a predetermined threshold denoting the maximum number of backleaps allowed in a concretization attempt.

The idea is to start from the failing index i and search backward for a transition $(\widehat{s}_j, \widehat{s}_k)$ with $i \leq k \leq j \leq l$ such that in the concrete model $\widehat{s}_j$ is reachable from $\widehat{s}_k$ in one step. If k and j exist, the concretization retreats from q_{i+1} back to q_k and makes a successful backleap from q_k to $\widehat{s}_j$; after that we continue the concretization process from $pre(q_k) \cap \widehat{s}_j$. At the same time, we record $\widehat{s}_j$ as a candidate tail state of π_r and $\widehat{s}_k$ as a candidate head state of π_r. In this modified concretization procedure, we can make backleaps more than once (bounded by the constant C_{thres})—therefore, it is possible to find a concrete counterexample that is longer than the abstract counterexample to be concretized.

Algorithm 3 ConcretizeCEX_backleap(M, π)

```
1: $i = l$; $q_i = \widehat{s}_i$;
2: $\eta_{bLeap} = 0$;
3: while ($i \neq 1$) do
4:     $q_{i-1} = pre(q_i) \cap \widehat{s}_{i-1}$;
5:     if ($q_{i-1} \neq \emptyset$) then
6:         $i = i - 1$;
7:     else
8:         find $k$ and $j$ such that $i \leq k \leq j \leq l$ and $pre(q_k) \cap \widehat{s}_j \neq \emptyset$.
9:         if ($k$ and $j$ do not exits, or $\eta_{bLeap} > C_{thres}$) then
10:            if ( Prove_CEX_by_induction( ) ) then
11:                return a concrete counterexample; //proved
12:            else
13:                return no concrete counterexample;
14:            end if
15:        end if
16:        $q_j = pre(q_k) \cap \widehat{s}_j$;
17:        $i = j$;
18:        $\eta_{bLeap} + +$;
19:        add segment $\pi^{k,j}$ to a list of candidates of $\pi_r$;
20:    end if
21: end while
22: return a concrete counterexample;
```

If after making C_{thres} backleaps, the new concretization procedure fails to find a concrete counterexample but some $\pi_r = \widehat{s}_k \ldots \widehat{s}_j$ have been recognized, we enter the induction proof mode (lines 9-15). In Algorithm 3, induction proof is implemented in the function Prove_CEX_by_induction.

B. The Induction Parameter

Given a recurring segment $\pi_r = \widehat{s}_k \ldots \widehat{s}_j$, we need to identify the *potential* induction parameter associated with π_r before calling Prove_CEX_by_induction. We will show in the next section that checking whether the induction proof holds is cheap computationally (compared to model checking), and only a correct induction parameter (paired with a true recurring segment) allows the induction proof to hold. The naive approach is to blindly try all the program variables one by one as the induction parameter, and under that assumption check whether the induction proof holds. The naive approach can be costly, but it does not affect the correctness of the overall CEGAR procedure.

In practice, however, we need to reduce the overhead of identifying induction variable. We first compute a list of promising candidate variables, that is, the program variables in V that are likely to be induction parameters. Then we try the candidate variables one by one, to see whether treating each of them as the induction parameter makes the induction proof go through.

We use the following criteria to filter out non-induction variables. Let $\pi_r = \widehat{s}_k, \ldots, \widehat{s}_j$ and $g(V)$ be the conditional expression guarding the transition from $\widehat{s}_j$ to $\widehat{s}_k$ (the back edge). For a program variable n to be the induction parameter, n needs to be in the transitive support of the expression $g(V)$. Furthermore, all the guard expressions inside π_r must be monotonic with respect to variable n—this guarantees that as long as a transition (of π_r) is valid for $n = k$, it is valid also for the $n = k + 1$ counterexample instance.

C. Counterexample Instances

In order to prove that ξ exists for all $n \geq 1$, we need to control the value of n in the model to produce a set of parameterized counterexample instances. Conceptually this is accomplished by finding inside π_p the last assignment statement to n, and replacing it with a statement assigning a symbolic value k to n.

In Fig. 1, for instance, the prefix segment is π_p =`(1)(2)(3)(4)(5)` and the last assignment statement to n is in line 3. A simple static analysis can locate the statement `n=1000` at line 3 and rewrite it into `n=k`, where k remains a parameterized symbolic value. When setting k to 1, we can check whether ξ exists for the induction basis `n=1`.

In practice, when ξ and n are given, we have implemented a procedure to locate the last assignment statement to n inside π_p, followed by an automatic rewriting of CDFG representation of the concrete model[1].

V. Proving The Existence of Counterexamples

In this section we explain the underlying algorithm for the function PROVE_CEX_BY_INDUCTION. Recall that before entering the induction proof mode, we have already identified a potential parameterized abstract counterexample $\xi = \pi_p\{\ \pi_r\ \}^n\ \pi_s$.

A. Induction Proof

The induction hypothesis is that there exists a concrete path inside the abstract counterexample $\pi_p\{\pi_r\}^k\pi_s$. We divide the counterexample instance into $\pi_p\{\pi_r\}^k$ and π_s, and define the intermediate pre-condition and post-conditions as follows:

$$\begin{aligned} G &= post^*(\pi_p, I) \\ F &= post^*(\pi_p\ \{\pi_r\}^k, I) \\ B &= pre^*\ (\pi_s, S_E) \end{aligned}$$

To rephrase the induction hypothesis,

$$F\ \cap B\ \neq \emptyset\ .$$

[1]If n does not appear in π_p, e.g., when loop condition is `(i<CONST)` and we rewrite it as `(i<n)`, we assume that $n = k$ holds in the initial state.

That is, there exists $s \in (F \cap B)$ such that s is reachable from I through $\pi_p\{\pi_r\}^k$ and can reach S_E through π_s.

We want to prove that there also exists a concrete path inside the abstract counterexample $\pi_p\{\pi_r\}^{k+1}\pi_s$. Similar to the previous case, we divide the counterexample instance into two parts: $\pi_p\{\pi_r\}^k$ and $\pi_r\pi_s$. The corresponding pre-conditions and post-conditions are defined as follows:

$$\begin{aligned} G' &= post^*(\pi_p, I) \\ F' &= post^*(\pi_p\ \{\pi_r\}^k, I) \\ B' &= pre^*\ (\pi_r\ \pi_s, S_E) \end{aligned}$$

The existence of a concrete counterexample is rephrased as follows,

$$F' \cap B' \neq \emptyset\ .$$

The induction step says that, if there exists a concrete state $s \in (F \cap B)$ when $n = k$, then there exists a concrete state $s' \in (F' \cap B')$ when $n = k+1$. With a little abuse of notation (F as a set and $F(V)$ as a formula), we express the induction step formally as follows,

$$\exists V\ .\ F(V) \wedge B(V)\ \rightarrow\ \exists V\ .\ F'(V) \wedge B'(V)\ .$$

Here V is the set of program variables. However, we shall never compute $F'(V)$ and $F(V)$ explicitly since they are over parameterized segment (expensive to compute). Instead, we rely on the analysis of $G(V)$ and $G'(V)$ to derive sufficient conditions under which the induction holds.

B. Induction Condition

We partition the set V of variables into $V = V_a \cup V_b$. V_b contains the induction parameter n and variables which are assigned in π_p to values that depends on n, and V_a contains the remaining variables. To get an induction proof, consider the following requirement on V_a and V_b (**C0**):

- in π_r, variables in V_b do not appear in any assignment (neither in left-hand side nor in right-hand side);
- guards $g(V_a, V_b)$ in π_r are monotonic with respect to V_b.

In such cases, pre- and post-conditions over π_r can be computed by updating functions for V_a and V_b separately (a Cartesian product). This characteristic has been captured by the notion of a disjunctively decomposable function described in Section III.

Given G and G', we define

$$\begin{aligned} G_a &= \exists V_b\ .\ G(V_a, V_b) \\ G_b &= \exists V_a\ .\ G(V_a, V_b) \\ G'_a &= \exists V_b\ .\ G'(V_a, V_b) \\ G'_b &= \exists V_a\ .\ G'(V_a, V_b) \end{aligned}$$

The induction holds if the following conditions are satisfied:

C1: G and G' differ only in the valuations of V_b:

$$\begin{aligned} G &= G_a(V_a) \wedge G_b(V_b), \\ G' &= G'_a(V_a) \wedge G'_b(V_b), \\ G_a &= G'_a \end{aligned}$$

C2: B and B' satisfy the following *goal containment* check:

$$\exists V_b\ .\ G_b \wedge B\ \rightarrow\ \exists V_b\ .\ G'_b \wedge B'\ .$$

All conditions can be checked algorithmically in the CDFG model by a combination of static analysis (for checking the partition of V) and pre-condition and post-condition computations over finite counterexample segments.

C. Proof of Correctness

The correctness of the induction proof is established by Theorem 3. The proof is illustrated pictorially in Fig. 4.

Theorem 3 *If $F \cap B \neq \emptyset$ and conditions **C0,C1,C2** are satisfied, then $F' \cap B' \neq \emptyset$.*

Proof: Since G and G' are disjunctively decomposable, V_b variables do not change their values in π_r, and V_a variables are updated independently from V_b in π_r, we know that F and F' are also disjunctively decomposable; that is,

$$\begin{aligned} F &= F_a(V_a) \wedge F_b(V_b) &&= F_a(V_a) \wedge G_b(V_b), \\ F' &= F'_a(V_a) \wedge F'_b(V_b) &&= F'_a(V_a) \wedge G'_b(V_b), \end{aligned}$$

and $F_a = F'_a$ (because of $G_a = G'_a$). From condition **C2**,

$$\begin{aligned} \exists V_b.G_b \wedge B &\subseteq \exists V_b.G'_b \wedge B' \ , \\ \exists V_b.F_a(V_a) \wedge G_b \wedge B &\subseteq \exists V_b.F'_a(V_a) \wedge G'_b \wedge B' \ , \\ \exists V_b.F \wedge B &\subseteq \exists V_b.F' \wedge B' \ . \\ \exists V.F(V) \wedge B(V) &\subseteq \exists V.F'(V) \wedge B'(V) \ . \end{aligned}$$

This means that if $s \in F \cap B$ exists, then $s' \in F' \cap B'$ exists. Note that s and s' may differ only in their valuations of variables in V_b. ■

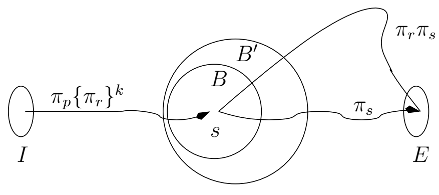


Fig. 4. The normal induction condition.

VI. Goal Containment Checking

A. The Working Example

In Fig. 1, when the set of predicates is empty (the initial abstraction), the abstract counterexample produced by the CEGAR procedure is $\pi =$(1)...(5)(6)(7). This abstract counterexample cannot be concretized by the standard concretization procedure or our backleap algorithm. Furthermore, our concretization algorithm cannot detect any counterexample pattern; at line 8 in Algorithm 3 there does not exist a valid index pair (j, k) for backleap.

After the first refinement iteration (which removes the spurious counterexample), the induction proof attempt becomes possible. Algorithm 3 will return the following counterexample pattern: $\xi = \pi_p\{\pi_r\}^n\pi_s$ such that $\pi_p =$(1)...(5), $\pi_r =$(6)(7)(8), and $\pi_s =$(6)(7).

The set V of program variables is partitioned into $V_a = \{\mathtt{i}\}$ and $V_b = \{\mathtt{n}\}$. Inside π_r, all the LHS variables are included in V_a, and the only variable in V_b does not change. Furthermore, inside π_r the guard $g : (\mathtt{i} \leq \mathtt{n})$ is monotonic with respect to n.

For the first two conditions,

$$\begin{aligned} G &= (\mathtt{i}=\mathtt{0}) \wedge (\mathtt{n}=\mathtt{k}) \ , \\ G' &= (\mathtt{i}=\mathtt{0}) \wedge (\mathtt{n}=\mathtt{k}+\mathtt{1}) \ . \end{aligned}$$

Let $G_a = G'_a = (\mathtt{i}=\mathtt{0})$, $G_b = (\mathtt{n}=\mathtt{k})$, and $G'_b = (\mathtt{n}=\mathtt{k}+\mathtt{1})$; both G and G' are disjunctively decomposable.

For the last condition,

$$\begin{aligned} B &= pre^*(\pi_s, i \geq n) &&= (\mathtt{i}=\mathtt{n}) \\ B' &= pre^*(\pi_r\pi_s, i \geq n) &&= (\mathtt{i}+\mathtt{1}=\mathtt{n}) \end{aligned}$$

Therefore,

$$\begin{aligned} \exists V_b \ . \ G_b \wedge B &= (\mathtt{i}=\mathtt{k}) \\ \exists V_b \ . \ G'_b \wedge B' &= (\mathtt{i}+\mathtt{1}=\mathtt{k}+\mathtt{1}) \end{aligned}$$

This proves the goal containment,

$$\exists V_b \ . \ G_b \wedge B \ \rightarrow \ \exists V_b \ . \ G'_b \wedge B' \ .$$

B. Goal Containment

Sets B' and B are regarded as the *goals* of postcondition computations over the common prefix $\pi_p\{\pi_r\}^k$. The goal containment check requires a decision procedure that supports quantified formulas. In our implementation, we use a mixed model checking procedure [18] which incorporates both bit-level and word-level symbolic representations. If all program variables are assumed to be in finite domains, one can also choose to use standard BDD-based fixpoint algorithms

Since we always consider a single program path (a finite prefix or suffix), the pre- and post-condition computations can be made efficient in practice. Although the CDFG may have many branching statements, when computing $post^*()$ and $pre^*()$ over π_p and π_s, we do not need to consider more than one branch at each pre or $post$ step. For instance, given $\pi = s_i, \ldots, s_j$ and a propositional formula ϕ, the weakest liberal pre-condition [19] of ϕ with respect to π, denoted by $wlp(\pi, \phi)$, is computed as follows,

- For a statement `s: v = e`, $wlp(s, \phi) = \phi(e/v)$;
- For a statement `s: assume(c)`, $wlp(s, \phi) = \phi \wedge c$;
- For a sequence of statements `s1; s2`, $wlp(s1 : s2, \phi) = wlp(s1, wlp(s2, \phi))$.

For a single CDFG path, there are only two types of statements: assignment statements and branching statements (`assume(c)` comes from `if(c)`). Complex C statements involving pointers, arrays, structures, function calls, etc. can be rewritten into simple statements involving scalar variables only during a preprocessing phase of the CDFG representation [20]. Therefore, the pre- and post-condition results over a single CDFG path do not blow up. In practice, the time spent on computing G, G', B, B' is often negligible when compared to other phases of the CEGAR procedure.

C. Strengthening Induction

The conditions can be strengthened by imposing a restricted area within which goal containment should hold. This step is optional, but may increase the chance of getting a proof. Specifically, we identify a state subspace $F_\infty \subseteq S$ such that goal containment within F_∞ can establish the proof. We define F_∞ as the union of F for all $k \geq 1$,

$$F_\infty = \bigcup_{k=1}^{\infty} post(\ \pi_p\{\pi_r\}^k, I\ )\ .$$

By definition, $F \wedge F_\infty = F$ and $F' \wedge F_\infty = F'$.

Assume that $F \wedge B \neq \emptyset$. We now prove that if $\exists V_b.\ (G_b \wedge B \wedge F_\infty)\ \rightarrow\ \exists V_b.\ (G'_b \wedge B' \wedge F_\infty)$, then

$$F' \wedge B' \neq \emptyset\ .$$

Since goal containment holds inside F_∞,

$$\begin{array}{rcl} \exists V_b.\ F_a \wedge (G_b \wedge B \wedge F_\infty) & \subseteq & \exists V_b.\ F'_a \wedge (G'_b \wedge B' \wedge F_\infty) \\ \exists V_b.\ (F \wedge B \wedge F_\infty) & \subseteq & \exists V_b.\ (F' \wedge B' \wedge F_\infty) \\ \exists V_b.\ (F \wedge B) & \subseteq & \exists V_b.\ (F' \wedge B') \end{array}$$

This is further illustrated in Fig. 5. We call it a *strengthening* because goal containment with F_∞ increases the chance of getting a proof.

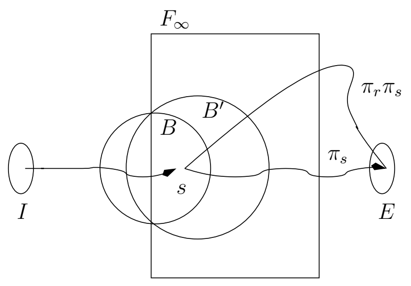


Fig. 5. The strengthened induction condition.

Our use of F_∞ is similar to the method in [21] on strengthening BMC induction proof with over-approximated reachable states. To ensure termination when computing F_∞, we compute exact post-conditions over $\pi_p\{\pi_r\}^k$ up to a finite set of values of k and then switch to *widening* [22]. In the working example, for instance, $F_\infty = i \leq n$.

D. Composition of Parameterized Traces

Extending the induction method to handle more complex counterexample patterns is possible. For counterexamples involving concatenation and embedding of recurring segments, we have identified sufficient conditions (special cases) in which goal containment can be check efficiently. Due to page limit, we omit the discussion on these complex patterns; an extended version of this paper is available upon request.

VII. EXPERIMENTS

We have implemented the new method in the F-Soft verification platform [23] and integrated with a CEGAR procedure [24]. F-Soft is a tool for analyzing safety properties in C programs by checking whether certain labeled statements are reachable. It has a preprocessing phase in which complex C statements (such as pointers, arrays, function calls, etc.) are rewritten into simple statements involving scalar variables only. For the simplified C program, it builds a CDFG representation, which is taken as input by subsequent analysis procedures, including CEGAR and our induction based method.

Our test cases are several software benchmarks in the public domain. For each test case, we run both standard CEGAR and the augmented version (with our induction method) of the same CEGAR procedure. All the experiments were conducted on a workstation with 3 GHz Pentium 4 processor and 2GB of RAM running Red Hat Linux 7.2.

A. The GNU bc Example

Our first test case comes from the GNU *bc* package (`bc-1.06`), which implements a Unix command line calculator with arbitrary precision. There is a known array bounds violation bug in line 176 of the file `storage.c`. The bug is illustrated in the last line of Fig. 6, where the guard `(indx<v_count)` should be changed to `(indx<a_count)`. This bug is inherently difficult to find with testing [25], since the corrupted heap does not always cause an immediate crash—it often causes a crash when another sometimes unrelated `malloc()` is called.

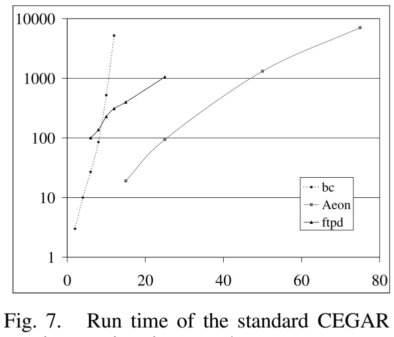
```
a_count = 256;
...
old_count = a_count;
a_count = a_count + STORE_INCR;
...
arrays = bc_malloc(a_count*sizeof(bc_var_array*));
names  = bc_malloc(a_count*sizeof(char*));
...
for (indx=1; indx<old_count; indx++)
 arrays[indx] = old_ary[indx];

for (; indx < v_count; indx++)
  arrays[indx] = NULL;              //failure
```

Fig. 6. Code in `more_arrays()` of the GNU bc example

This bug cannot be detected using standard CEGAR when `a_count=256`. However, if `a_count` is set to some small value, standard CEGAR may find a concrete counterexample. In our experiments, `a_count` is set to various values starting with 1, 2, 3, ... The result is given in Fig. 7. With a small initial value, standard CEGAR found the concrete counterexample, but it demonstrates poor runtime performance and is clearly not scalable.

The CEGAR procedure augmented with our induction method was able to identify and prove the existence of



bc	
a_count	time (s)
1	2.1
2	3.6
3	5.9
4	10
5	14
6	27
7	49
8	85
9	135
10	522
11	885
12	5256

Fig. 7. Run time of the standard CEGAR procedure: x-axis: values of n; y-axis: run time in seconds.

a parameterized counterexample within 10 seconds. This is slightly slower than standard CEGAR for `a_count=3`. However, the proof is valid (i.e., a concrete counterexample instance exists) for all `a_count`$= 1, 2, ..., k$.

B. The Aeon Example

Our second test case comes from the Linux mail transfer agent `Aeon 0.02a`. There is a buffer overflow inside the function `getConfig`, whenever it calls `strcpy` to duplicate a string returned by the function `getenv` to a buffer with size `MAX_LEN`. This bug is representative for many buffer overflows leading to possible security breaches. This example was also studied by Kroening and Weissenbacher in [14].

We applied standard CEGAR as well as the augmented version to this example. When `MAX_LEN=512`, our implementation of standard CEGAR failed to detect the bug within 4 hours (BLAST [26] and SLAM [7] also timed out, as reported in [14]). Our induction based method was able to identify and prove the existence of a parameterized counterexample within 6 seconds. In comparison, the loop detection method in [14] detected the bug within 254.5 seconds; the runtime of their method will keep increasing as `MAX_LEN` becomes larger (its runtime was 25.0 seconds when `MAX_LEN=25`). In contrast, our proof is valid (i.e., a concrete counterexample instance exists) for all `MAX_LEN`$= 1, 2, ..., k$.

C. The ftpd Example

Our third test case comes from the `wu-ftpd-2.6.2` package. There is a buffer overrun inside `ftprestart.c` when the function `newfile` is called. The induction parameter is `numfiles`, which is 1024 in the concrete counterexample. This example was also studied in [27]. Standard CEGAR failed to detect the bug (although it can find a bug when we set `numfiles` to smaller values, as is shown in Fig. 7), whereas our induction augmented CEGAR procedure found the bug in 22 seconds.

VIII. CONCLUSIONS

We have presented an induction based method for proving the existence of long counterexamples. The method avoids a state-by-state match of the abstract counterexample during the search for concrete counterexamples. It provides complementary strengths to the popular CEGAR methods. For future work, we want to extend the induction method to handle more complex counterexample patterns.

REFERENCES

[1] O. Kupferman and M. Y. Vardi, "Model checking of safety properties," *Formal Methods in Systems Design*, vol. 19, no. 3, pp. 291–314, 2001.

[2] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Formal Methods in Computer Aided Design*. Springer, 2000, pp. 108–125, LNCS 1954.

[3] L. de Moura, H. Rueß, and M. Sorea, "Bounded model checking and induction: From refutation to verification," in *Computer Aided Verification (CAV'03)*. Springer, 2003, pp. 1–13, LNCS 2725.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, Mar. 1999, pp. 193–207, LNCS 1579.

[5] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes*. Princeton, NJ: Princeton University Press, 1994.

[6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification (CAV'00)*. Springer, 2000, pp. 154–169, LNCS 1855.

[7] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Programming Language Design and Implementation (PLDI'01)*, June 2001, pp. 203–213.

[8] K. L. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification (CAV'03)*. Springer, July 2003, pp. 1–13, LNCS 2725.

[9] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart simulation using collaborative formal and simulation engines," in *International Conference on Computer-Aided Design*, San Jose, CA, Nov. 2000, pp. 120–126.

[10] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing." in *Programming Language Design and Implementation (PLDI'05)*, June 2005, pp. 213–223.

[11] K. Nanshi and F. Somenzi, "Guiding simulation with increasingly refined abstract traces." in *Proceedings of ACM/IEEE Design Automation Conference*, 2006, pp. 737–742.

[12] C. Wang, B. Li, H. Jin, G. D. Hachtel, and F. Somenzi, "Improving Ariadne's bundle by following multiple threads in abstraction refinement," in *International Conference on Computer-Aided Design*, Nov. 2003, pp. 408–415.

[13] P. Bjesse and J. Kukula, "Using counter example guided abstraction refinement to find complex bugs," in *Design, Automation and Test in Europe (DATE'04)*, Mar. 2004, pp. 10 156–10 161.

[14] D. Kroening and G. Weissenbacher, "Counterexamples with loops for predicate abstraction," in *Computer Aided Verification (CAV'06)*. Springer, 2006, pp. 152–165, LNCS 4144.

[15] M. N. Seghir and A. Podelski, "ACSAR: Software model checking with transfinite refinement," in *International SPIN Workshop on Model Checking Software*. Springer, 2007, LNCS 4595.

[16] T. Ball, O. Kupferman, and M. Sagiv, "Leaping loops in the presence of abstraction," in *Computer Aided Verification (CAV'07)*. Springer, 2007, pp. 491–503, LNCS 4590.

[17] C. Wang, H. Kim, and A. Gupta, "Hybrid CEGAR: Combining variable hiding and predicate abstraction," in *International Conference on Computer Aided Design (ICCAD'07)*, 2007, to appear.

[18] Z. Yang, C. Wang, F. Ivančić, and A. Gupta, "Mixed symbolic representations for model checking software programs," in *Formal Methods and Models for Codesign (MEMOCODE'06)*, July 2006, pp. 17–24.

[19] E. Dijkstra, *A Discipline of Programming*. NJ: Prentice Hall, 1976.

[20] F. Ivančić, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C. Wang, and Z. Yang, "Model checking C program using F-Soft," in *International Conference on Computer Design*, Oct. 2005, pp. 297–308.

[21] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar, "Abstraction and BDDs complement SAT-based BMC in DiVer," in *Computer Aided Verification (CAV'03)*. Springer, 2003, pp. 206–209, LNCS 2725.

[22] C. Wang, Z. Yang, A. Gupta, and F. Ivančić, "Using counterexamples for improving the precision of reachability computation with polyhedra," in *Computer Aided Verification (CAV'07)*. Springer, 2007, pp. 352–265, LNCS 4590.

[23] F. Ivančić, Z. Yang, I. Shlyakhter, M. Ganai, A. Gupta, and P. Ashar, "F-SOFT: Software verification platform," in *Computer-Aided Verification*. Springer, 2005, pp. 301–306, LNCS 3576.

[24] H. Jain, F. Ivančić, A. Gupta, I. Shlyakhter, and C. Wang, "Using statically computed invariants inside the predicate abstraction and refinement loop," in *Computer Aided Verification (CAV'06)*. Springer, 2006, pp. 137–151, LNCS 4144.

[25] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous identification of multiple bugs." in *International Conference on Machine Learning*, 2006, pp. 1105–1112.

[26] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Principles of programming languages (POPL'02)*, 2002, pp. 58–70.

[27] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek, "Buffer overrun detection using linear programming and static analysis." in *ACM Conference on Computer and Communications Security*, Oct. 2003, pp. 345–354.