View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Apollo

Danger Invariants*

Cristina David[1], Pascal Kesseli[1], Daniel Kroening[1], Matt Lewis[1,2]

[1] University of Oxford, [2] Improbable

Abstract. Static analysers search for overapproximating proofs of safety commonly known as *safety invariants*. Conversely, static bug finders (e.g. Bounded Model Checking) give evidence for the failure of an assertion in the form of a *counterexample trace*. As opposed to safety invariants, the size of a counterexample is dependent on the *depth of the bug*, i.e., the length of the execution trace prior to the error state, which also determines the computational effort required to find them. We propose a way of expressing danger proofs that is independent of the depth of bugs. Essentially, such danger proofs constitute a compact representation of a counterexample trace, which we call a *danger invariant*. Danger invariants summarise sets of traces that are guaranteed to be able to reach an error state. Our conjecture is that such danger proofs will enable the design of bug finding analyses for which the computational effort is independent of the depth of bugs, and thus find deep bugs more efficiently. As an exemplar of an analysis that uses danger invariants, we design a bug finding technique based on a synthesis engine. We implemented this technique and compute danger invariants for intricate programs taken from SV-COMP 2016.

1 Introduction

Safety analysers search for proofs of safety commonly known as *safety invariants* by overapproximating the set of program states reached during all program executions. Fundamentally, they summarise traces into abstract states, thus trading the ability to distinguish traces for computational tractability [1].

Conversely, static bug finders that use techniques such as Bounded Model Checking (BMC) search for proofs that safety can be violated. Dually to safety proofs, we will call these *danger proofs*. Traditionally, a danger proof is represented by a concrete counterexample trace leading to an error state [2].

For illustration, we examine the safe and unsafe programs in Fig. 1. The program in Fig. 1a is safe as witnessed by the safety invariant $Inv(x) = x{\neq}y$, which holds in the initial state (where $x{=}0$ and $y{=}1$), is inductive with respect to the body of the loop ($x{\neq}y \Rightarrow (x{+}1){\neq}(y{+}1)$) and, on exit from the loop, makes the assertion hold. Now, if we replace the guard by `x<1000000`, the program remains safe as witnessed by the same safety invariant.

On the other hand, the program in Fig. 1b is unsafe as, depending on a nondeterministic choice (denoted by "*"), y may not be incremented in each

* This research was supported by ERC project 280053 (CPROVER).

```
x = 0; y = 1;
// while (x<1000000)
while (x<10){
     x++;
     y++;
}

assert ( x != y );
```

(a)

```
x = 0; y = 1;
// while (x<1000000)
while (x<10){
     x++;
     if(*) y++;
}

assert ( x != y );
```

(b)

Fig. 1: Safe and unsafe example programs

iteration. A possible danger proof for this example is given by the concrete counterexample trace: (x=0, y=1), (x=1, y=1), (x=2, y=2), (x=3, y=3), (x=4, y=4), (x=5, y=5), (x=6, y=6), (x=7, y=7), (x=8, y=8), (x=9, y=9), (x=10, y=10).

Similarly to what we did for the program in Fig. 1a, let the guard in Fig. 1b now be replaced by `x<1000000`. However, as opposed to the program in Fig. 1a, now we cannot use the same danger proof we computed for the original program (instead a possible danger proof for the modified program is (x=0, y=1), (x=1, y=1), (x=2, y=2), (x=3, y=3), $\cdots$ (x=1000000, y=1000000)). The cause for this is that, as opposed to safety invariants, the size of a counterexample trace is dependent on the *depth of the bug*, i.e., the length of the execution trace prior to the error state. The bug in the original program in Fig. 1b manifests in execution traces of length 10, whereas for the modified program we need execution traces of length 1000000 to expose the bug. We will refer to bugs that only manifest in long execution traces as *deep bugs.*

The size of the counterexample also impacts the computational effort required to find them. For instance, bounded model checkers compute counterexample traces by progressively unwinding the transition relation. Consequently, the computational effort required to discover an assertion violation typically grows exponentially with the depth of the bug. Notably, the scalability problem is not limited to procedures that implement BMC. Approaches based on a combination of over- and underapproximations such as predicate abstraction [3] and lazy abstraction with interpolants (LAwI) [4] are not optimised for finding deep bugs either. The reason for this is that they can only detect counterexamples with deep loops after the repeated refutation of increasingly longer spurious counterexamples. The analyser first considers a potential error trace with one loop iteration, only to discover that this trace is infeasible. Consequently, the analyser increases the search depth, usually by considering one further loop iteration. This repeated search suffers from the same exponential blow-up as BMC.

In this paper we propose a way of expressing danger proofs that is independent of the depth of the bug. Essentially, such a danger proof constitutes a compact representation of a counterexample trace, which we call a *danger invariant.* Similarly to safety invariants, danger invariants are based on summarisation. Our conjecture is that such danger proofs will enable the design of bug finding analyses for which the computational effort is also independent of the depth of bugs, and thus have the potential to find deep bugs more efficiently.

As an exemplar of an analysis that uses the newly introduced notion of danger invariants, we design a bug finding technique based on a synthesis engine.

Contributions:

– We introduce the notion of danger invariant, which, similarly to safety invariants, uses summarisation to compactly represent counterexamples. We discuss danger invariants both in the context of total and partial correctness.

– We present a procedure for inferring such danger invariants based on program synthesis. Our program synthesiser is specifically tailored for danger invariants, being able to efficiently synthesise multiple programs.

– We implemented our analysis and applied it to intricate programs taken from the Competition on Software Verification SV-COMP 2016 [5]. The focus of our experimental evaluation are danger invariants for code with deep bugs. Our experimental results show that our technique outperforms other tools when the bugs require many iterations of a loop in order to manifest. This suggests that it has strengths complementary to those of other techniques and could be used in combination with them (e.g., a compositional analysis based on may/must analysis and danger invariants).

2 Illustration

```
int i, j, k;

for (k = 0; k < 100; k++) {
  if (*) j++;
}
for (i = 0; i < 1000000; i++) {
  if (*) j++;
}
assert(i != j);
```

(a)

```
x = 0; y = 1;
while (x < 10) {
  y++;
}
assert(x < 10);
```

(b)

Fig. 2: Illustrative examples

To illustrate some of the pitfalls involved in proving that a program has a bug, we direct the reader's attention to Fig. 2a. This program is unsafe (the assertion can be violated), but this fact is hard to prove for traditional bug finders (based on random testing, BMC or concolic execution). We found that SMACK 1.5.1 [6] and CBMC 5.5 [7] timed out on this example, Seahorn 2.6 [8] returned "unknown" and CPAChecker 1.4 [9] (incorrectly) says "safe". This program is difficult for bug finders to analyse for the following reasons:

- The program is nondeterministic and the vast majority of the paths through the program do not trigger the bug.
- Many of the initial values of the program variables do not lead to the bug.
- The assertion violation does not occur until a very large number of loop iterations have executed.

Despite these features and the difficulty that automated tools have with this program, it is quite easy to convince a human that the program is unsafe using an argument something like the following:

1. In the second loop, if we ever reach a state with $i = j$, we can maintain that $i = j$ by taking the "if" branch and incrementing j.
2. If we are in the second loop with $i < j$, we can reduce the gap between i and j by *not* taking the "if" branch, so i will be incremented but j will not. If $j - i \leq 1000000$ then we can eventually have i "catch up" with j by repeatedly taking the "else" branch.
3. Therefore, if we begin the second loop with $0 \leq j \leq 1000000$, we can eventually reach a state with $i = j$ and from there eventually exit the loop with $i = j$, at which point the assertion will be violated.
4. We can enter the second loop with $0 \leq j \leq 1000000$ quite easily. For example, if $0 \leq j \leq 999900$ then any path through the first loop will land us at the start of the second loop in such a state.
5. There are several valid initial states with $0 \leq j \leq 999900$, and so the assertion can certainly be violated.

This argument is quite unlike the argument that an existing automated bug finder would use. We have not provided a concrete error trace, or even a concrete initial input, but we have still been able to prove that there is definitely an error in the program. It is worth noting that this proof is *much* shorter than a full error trace (which would be at least 1000100 steps long), it is much easier for a human to understand than the full, explicit error trace and indeed it is *much easier to find*.

The proof outlined above makes use of several techniques usually associated with safety proving: abstraction (we described sets of states symbolically), induction (e.g., we argued by induction that the state $i = j$ could be maintained once reached) and compositional reasoning (we proved a lemma about each loop separately, then combined these lemmas into a proof that the program as a whole had a bug). At the same time, such a proof does not admit false alarms.

In the remainder of this paper, we will show how this intuitive notion of symbolically proving the existence of a bug without providing an explicit error trace can be made precise by introducing the concept of a danger invariant. Our definition is presented abstractly, so that any method of symbolic reasoning or invariant generation (including manual annotation by a verification engineer) can be used to generate and verify danger invariants. We will also show how the constraints defining a danger invariant can be solved using program synthesis.

3 Danger Invariants

In this section, we formalise the notion of a danger invariant. We represent a program P as a transition system with state space X and transition relation $T \subseteq X \times X$. For a state $x \in X$ with $T(x, x')$, x' is said to be a successor of x under T. We denote initial states by I and error states by E. We start by defining some background notions.

Definition 1 (Execution Trace) *An execution trace $\langle x_0 \ldots x_n \rangle$ is a (potentially infinite) sequence of states such that any two successive states are related by the program's transition relation T, i.e. $\forall 0 \leq i < n . T(x_i, x_{i+1})$.*

Definition 2 (Counterexample Trace) *A finite execution trace $\langle x_0 \ldots x_n \rangle$ is a counterexample iff x_0 is an initial state, $x_0 \in I$, and x_n is an error state, $x_n \in E$.*

A counterexample trace is a proof of the existence of a reachable error state (i.e., a state where some safety assertion is violated).

The question we try to answer in this paper is whether we can derive a compact representation of a danger proof that does not require us to explicitly write down every intermediate state. For a loop $L(I, G, T, A)$ (I denotes the initial states, G is the guard, T is the transition relation and A is the assertion immediately after the loop), this is captured by the notion of *danger invariant*, defined next.

Definition 3 (Danger Invariant) *A predicate D is a danger invariant for the loop $L(I, G, T, A)$ iff it satisfies the following criteria:*

$$\exists x_0 . I(x_0) \wedge D(x_0) \tag{1}$$

$$\forall x . D(x) \wedge G(x) \rightarrow \exists x' . T(x, x') \wedge D(x') \tag{2}$$

$$\forall x . D(x) \wedge \neg G(x) \rightarrow \neg A(x) \tag{3}$$

A danger invariant is a dual of a safety invariant that captures the fact that there is some trace containing an error state starting from an initial state: (1) captures the fact that D is reachable from an initial state x_0, (2) shows that there exists some transition with respect to which D is inductive and (3) checks that the assertion is violated on exit from the loop.

The existential quantifier for x' in (2) is important for nondeterministic programs, where it is enough for the danger invariant to capture the existence of some error trace for only one nondeterministic choice. We make this explicit by introducing a Skolem function S that chooses the successor x':

$$\exists S . \forall x . D(x) \wedge G(x) \rightarrow T(x, S(x)) \wedge D(S(x)) \tag{4}$$

Our definition of an execution trace (Definition 1) includes infinite traces. Thus, the trace containing the error may be infinite and the error state will not be reachable at all. For example, consider Fig. 2b. A danger invariant is 'true', which meets all of the criteria (1), (2) and (3).

However, we can actually prove partial correctness of the program – the program contains no terminating traces and so the assertion is never even reached. To ensure that the error traces are finite, we will introduce a *ranking function*, which will serve as a proof of termination. Below we recall the definition of a ranking function:

Definition 4 (Ranking function) *A function $R : X \rightarrow Y$ is a* ranking function *for the transition relation T if Y is a well-founded set with order $>$ and R is injective and monotonically decreasing with respect to T.*

We assume that programs have unbounded but countable nondeterminism, and so require that our ranking functions' co-domains are recursive ordinals. In particular, we will consider ranking functions with co-domain ω^n, i.e., n-tuples of natural numbers ordered lexicographically. This is the final piece we need to define a partial danger invariant:

Definition 5 (Partial Danger Invariant) *A predicate D_p is a danger invariant for the loop $L(I, G, T, A)$ in the context of partial correctness iff it satisfies the following criteria:*

$$\exists x_0 . I(x_0) \wedge D_p(x_0) \tag{5}$$

$$\begin{aligned} \exists R, S . \forall x . D_p(x) \wedge G(x) \rightarrow R(x) > 0 \wedge T(x, S(x)) \wedge \\ D_p(S(x)) \wedge R(S(x)) < R(x) \end{aligned} \tag{6}$$

$$\forall x . D_p(x) \wedge \neg G(x) \rightarrow \neg A(x) \tag{7}$$

Note that the ranking function R does not guarantee the termination of all possible executions, but only the termination of some erroneous one. It is also important to notice that D_p is *not an underapproximation* of the reachable program states – there may well be D_p-states that are unreachable, and there may well be D_p-states that do not violate the assertion. However, every $(D_p \wedge \neg G)$-state does violate the assertion, and it is certainly the case that at least one such state is reachable.

Example 1 *With Def. 5, for the example in Fig. 2b there exists no danger invariant.*

For the program in Fig. 1b a danger invariant is $D_p(x, y) = y = (x < 1?1 : x)$ and ranking function $R(x, y) = 10-x$. Essentially, this invariant says that y must not be incremented for the first iteration of the loop (until x reaches the value 1), and from that point, for the remaining iterations, y gets always incremented such that $x = y$. For this case, D_p is a compact and elegant representation of a feasible counterexample trace. The witness Skolem function that we get is $S_y(x, y) = (x < 1?y : y + 1)$.

In Sec. 1, we have seen that the counterexample trace for the modified version of the program in Fig. 1b (the one with a larger guard) was much longer than that for the original version of the program. However, both the original and the modified programs have the same danger invariant $D_p(x, y) = y = (x < 1?1 : x)$ and the same Skolem function. This supports our conjecture that danger invariants are independent on the depth of bugs. A ranking function for the modified program in Fig. 1b is $R(x, y) = 1000000 - x$, which is also a valid ranking function for the original one.

Danger invariants for total correctness. While Def. 5 defines a danger invariant for partial correctness, we argue that the danger invariant in Def. 3 proves the existence of an erroneous trace in the context of total correctness. This trace may either be an error trace leading to an assertion violation, or a recurrence set denoting an infinite execution trace. We can differentiate between the two scenarios by checking whether the loop guard G holds for all the states in D, i.e. $\forall x.D(x) \Rightarrow G(x)$. If this is true, then Formula 3 is always vacuously true and D is a proof of the existence of a recurrence set. Otherwise, D is a proof of the existence of an assertion violation.

Example 2 *With Def. 3, a possible danger invariant for the example in Fig. 2b is $D(x) = x{<}10$. As the guard of the loop holds for all the D-states, this is a recurrence set.*

4 Generating Second-Order Verification Conditions

In this section, we present an algorithm for generating second-order constraints describing the existence of a danger proof for a program with potentially nested loops. We only give the algorithm for partial correctness as it is the more complex one (the corresponding procedure for total correctness does not have to generate the constraints for the ranking functions). We define the notion of a danger proof with respect to two assertions A and B:

Definition 6 *A danger proof of a triple (A, P, B) shows the existence of a finite path through the program P from a state x to a state x' such that $A(x)$ and $\neg B(x')$.*

The generation of the verifications conditions is performed by Algorithm 1. This algorithm allows danger invariants for pieces of a program to be composed together into a danger proof for the whole program. We discuss solving these constraints in the next section.

Algorithm 1 is split into two procedures. The ExistsDangerPath procedure generates the constraints showing the existence of some erroneous execution trace that might not be reachable from the initial states (it overapproximates the initial states). Overapproximating invariants are easier to compose than underapproximating ones, which enables us to construct a modular constraint generation technique for arbitrary programs and only add the reachability constraints at the outer level in the DangerConstraints procedure.

Proposition 1. *The constraints generated by a call to the function* ExistsDangerPath(A, P, B) *are satisfiable iff there is a finite path through the program P from a state x to a state x' such that $A(x)$ and $\neg B(x')$.*

The high-level strategy for the ExistsDangerPath procedure is the following. Given a program P, introduce fresh function symbols denoting Skolem functions for the n nondeterministic assignments, as well as to the danger invariants and ranking functions required by each of the loops.

The most interesting branch of the algorithm is the one for a loop with guard G and transition relation T. In this case, we need to emit the constraints necessary for a danger invariant. As previously stated, at this point we do not check that the danger invariant is reachable from the initial states. Instead, the first emitted constraint captures the fact that the danger invariant D_p is an over-approximation of the initial states A. The second constraint captures the fact that the negation of the post-state B must hold on exit from the loop and the third constraint captures the fact that the ranking function R is bounded from below. The inductiveness and the ranking function's monotonicity are proven through the recursive call to ExistsDangerPath, where the pre-state denotes the LHS of the inductiveness proof and the post-state represents the RHS plus the monotonicity of the ranking function. Note that the negation in the post-state ensures the fact that the generated verification conditions correspond to the situation where the inductiveness and monotonicity hold. The additional fresh variables $\underline{\mathbf{v}}^f$ are needed to express the (relational) monotonicity condition for the ranking function.

Procedure DangerConstraints adds the necessary constraints such that the danger proof is reachable from an initial state $\underline{\mathbf{v_0}}$.

The end result of Algorithm 1 is a set of second-order constraints, where the freshly introduced second-order variables (for the Skolem functions, danger invariants and ranking functions) are existentially quantified. If the resulting system of second-order constraints is satisfiable, then the solution (i.e., an assignment to the uninterpreted function symbols) is a danger proof for the full program. In other words, the second-order constraints generated are satisfiable iff the program contains a finite error trace.

Example 3 *In Figure 3 we illustrate how Algorithm 1 works by using it to generate a danger proof for the nondeterministic program at the level 0 call to* DangerConstraints *with the generic pre- and post-states being A and B, respectively. The explicit levels in the figure denote the call stack together with the constraints generated for each of them. Additionally, when going from level 3 to level 4, we omit the recursive call for the sequential composition and simply apply the weakest precondition for the whole code, resulting in the following VC:*

$$D_p(i) \wedge i{\leq}10 \Rightarrow wp((\texttt{i}^f\texttt{=i;\ if(*)i=i+1}), D_p(i) \wedge R(i^f){>}R(i))$$

The overall verification condition is the conjunction of the constraints generated at each level, where the second-order entities D_p, R, S and C are existentially quantified. The existential quantifier over i_0 ranges over all the emitted VCs. If we consider $A(i) = true$ and $B(i) = (i{=}10)$, then a satisfying assignment for these constraints is:

$$i_0 \mapsto 0, \quad D_p(i) \mapsto i{\leq}11, \quad R(i) \mapsto 12{-}i, \quad S(i) \mapsto true, \quad C(i) \mapsto i{\leq}11$$

The recursive constraint generation technique given in Algorithm 1 makes it easy to generate verification conditions for nested loops in a modular manner. One example with nested loops is given the extended version of the paper [10].

5 Generating Danger Invariants using Synthesis

Since the programs we are analysing are either safe or unsafe, and assuming that a proof is expressible in our logic, a program either accepts a safety invariant SI

Initial call to DangerConstraints**:**

DangerConstraints(

A,

```
while (i ≤ 10) {
   if (*) i := i+1;
},
```

B)

(**Level 0**)

Emitted VCs:

$$\exists i_0 . A(i_0)$$

Initial call to ExistsDangerPath**:**

ExistsDangerPath(

$\langle i \rangle, \mathit{true}$,

```
i = i0;
while (i ≤ 10) {
   if (*) i := i+1;
},
```

B)

(**Level 1**)

Recursive calls:

ExistsDangerPath(

$\langle i \rangle, \mathit{true}$,

```
i = i0,
```

$\neg C$)

ExistsDangerPath(

$\langle i \rangle, C$,

```
while (i ≤ 10) {
   if (*) i := i+1;
},
```

B)

(**Level 2**)

Emitted VCs:

$$\forall i . \mathit{true} \Rightarrow C(i_0) \wedge$$
$$C(i) \Rightarrow D_p(i) \wedge$$
$$D_p(i) \wedge i{>}10 \Rightarrow \neg B(i) \wedge$$
$$D_p(i) \wedge i{\leq}10 \Rightarrow R(i){>}0$$

Recursive call:

ExistsDangerPath(

$\langle i, i^f \rangle, D(i) \wedge i{\leq}10$,

```
i^f = i;
if (*) i := i+1,
```

$\neg(D(i) \wedge R(i^f){>}R(i)))$

(**Level 3**)

Emitted VCs:

$$\forall i . D_p(i) \wedge i{\leq}10 \Rightarrow (S(i) \wedge D_p(i{+}1) \wedge R(i){>}R(i{+}1)) \vee (\neg S(i) \wedge D_p(i) \wedge R(i){>}R(i))$$

(**Level 4**)

Fig. 3: Generating verification conditions for a program with nondeterminism

Algorithm 1 Generate VCs for the triple (A, P, B) over program variables $\underline{\mathbf{v}}$

```
procedure ExistsDangerPath( $\underline{\mathbf{v}}$, $A, P, B$))
    switch $P$ do
        case while($G$) do $T$ end
            $D_p \leftarrow$ Fresh
            $R \leftarrow$ Fresh
            $\underline{\mathbf{v}^f} \leftarrow$ FreshCopy($\underline{\mathbf{v}}$)
            Emit($\forall \underline{\mathbf{v}}. A(\underline{\mathbf{v}}) \Rightarrow D_p(\underline{\mathbf{v}})$)
            Emit($\forall \underline{\mathbf{v}}. D_p(\underline{\mathbf{v}}) \wedge \neg G(\underline{\mathbf{v}}) \Rightarrow \neg B(\underline{\mathbf{v}})$)
            Emit($\forall \underline{\mathbf{v}}. D_p(\underline{\mathbf{v}}) \wedge G(\underline{\mathbf{v}}) \Rightarrow R(\underline{\mathbf{v}}) > 0$)
            ExistsDangerPath($\underline{\mathbf{v}} + \underline{\mathbf{v}^f}$,
                $D_p(\underline{\mathbf{v}}) \wedge G(\underline{\mathbf{v}})$,
                $\underline{\mathbf{v}^f} := \underline{\mathbf{v}}; T$,
                $\neg(D_p(\underline{\mathbf{v}}) \wedge R(\underline{\mathbf{v}^f}) > R(\underline{\mathbf{v}}))$)
        case $x := *$
            $S \leftarrow$ Fresh
            ExistsDangerPath($\underline{\mathbf{v}}$, $A, x := S(\underline{\mathbf{v}}), B$)
        case $P_1; P_2$
            $C \leftarrow$ Fresh
            ExistsDangerPath($\underline{\mathbf{v}}$, $A(\underline{\mathbf{v}}), P_1, \neg C(\underline{\mathbf{v}})$)
            ExistsDangerPath($\underline{\mathbf{v}}$, $C(\underline{\mathbf{v}}), P_2, B(\underline{\mathbf{v}})$)
        case default
            Emit($\forall \underline{\mathbf{v}}. A(\underline{\mathbf{v}}) \Rightarrow \mathbf{wp}(\neg B, P)(\underline{\mathbf{v}})$)

procedure DangerConstraints($A, P, B$)
    $\underline{\mathbf{v}} \leftarrow$ fv($P$)
    $\underline{\mathbf{v_0}} \leftarrow$ FreshCopy($\underline{\mathbf{v}}$)
    Emit($\exists \underline{\mathbf{v_0}}. A(\underline{\mathbf{v_0}})$)
    ExistsDangerPath( $\underline{\mathbf{v}}$, $\top$, $\underline{\mathbf{v}} := \underline{\mathbf{v_0}}$; $P$, $B(\underline{\mathbf{v}})$)
```

or a danger invariant D_p. For a loop $L(I, G, T, A)$, we model this as a disjunction as stated in Definition 7. The generalised safety formula is a theorem of second-order logic, and our decision procedure will always be able to find witnesses SI, D_p, S, R, y_0 demonstrating its truth, provided such a witness is expressible in our logic. The synthesised predicate SI is a purported safety invariant and the D_p, N, R, y_0 constitute a purported danger invariant.

If SI is really a safety invariant, the program is safe, otherwise D_p (with witnesses to the existence of an error trace with Skolem function S, initial state y_0 and ranking function R) will be a danger invariant and the program is unsafe. Exactly one of these proofs will be valid, i.e., either SI will satisfy the criteria for a safety invariant, or D_p, S, R, y_0 will satisfy the criteria for a danger invariant. We can simply check both cases and discard whichever "proof" is incorrect. We omit the algorithm for generating safety verification conditions for a whole program as this is well covered in the literature [11].

Synthesis engine We employ Counterexample-Guided Inductive Synthesis (CEGIS) to synthesise programs for SI, D_p, S, R. The processes is graphically illustrated in Fig. 5. Our synthesis engine conjectures solution programs based on

Definition 7 (Generalised Safety Formula)

$$\exists SI, D_p, S, R, y_0 . \forall x, x', y. \begin{pmatrix} I(x) \rightarrow SI(x) \wedge \\ SI(x) \wedge G(x) \wedge T(x, x') \rightarrow SI(x') \wedge \\ SI(x) \wedge \neg G(x) \rightarrow A(x) \end{pmatrix} \vee$$

$$\begin{pmatrix} I(y_0) \wedge D_p(y_0) \wedge \\ D_p(y) \wedge G(y) \rightarrow R(y) > 0 \wedge T(y, S(y)) \wedge D(S(y)) \\ \qquad \wedge R(y) > R(S(y)) \wedge \\ D_p(y) \wedge \neg G(y) \rightarrow \neg A(y) \end{pmatrix}$$

Fig. 4: General second-order safety formula

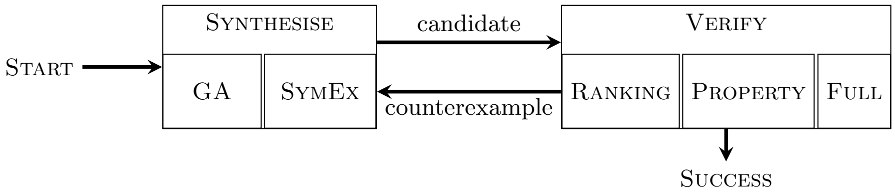

Fig. 5: Synthesis loop with multiple backends

a limited set of counterexamples C. These solutions are guaranteed to satisfy all known counterexamples $c_i \in C$ and are refined with each new c_i. Each conjecture is verified by a verifier component, which terminates the process if the constraint holds (Success). Otherwise the resulting c_j is added to C and provided to the synthesiser for further refinement. As mentioned earlier, for our particular use case the synthesiser must always find a solution (although in practice this might take a very long time as discussed in the experimental section).

In order to efficiently synthesise SI, D_p, S, R simultaneously, our algorithm implements concurrent backends in both the synthesis and verification stage. In the synthesis stage, a symbolic execution (SymEx) as well as a genetic algorithm (GA) backend concurrently search for new candidates satisfying C. GA is an alternative way to traverse the space of possible solutions, simulating an evolutionary process using selection, mutation and crossover operators. It maintains a large population of programs which are paired using crossover operation, combining successful program features into new solutions. In order to avoid local minima, the mutation operator replaces instructions by random values at a comparatively low probability. The backends share information about synthesised candidates and pass a complying solution on to the verification component. Synthesis components use different instruction sets for SI, D_p, S, R optimised for their clause in the full danger constraint.

To facilitate concurrent synthesis of multiple programs, the verification component searches for different counterexamples in the same iteration. It restricts the full danger constraint to either find a c_i witnessing an inconsistent ranking (Ranking) or a violation of the user property for which we are proving danger (Property). Furthermore, the engine provides one counterexample over the full, unrestricted danger constraint (Full). This ensures that the synthesis component receives sufficient information at each iteration to refine all synthesised programs SI, D_p, S, R. The GA synthesis backend considers these counterexamples in its selection and crossover operators. Candidates that solve distinct sets of counterexamples have a higher probability of selection as crossover partners in order to produce solutions that satisfy all types of counterexamples and hence implement SI, D_p, S, R correctly. This is preferable over fitness values based on solved counterexamples only, since it avoids local minima where candidates may solve a multitude of counterexamples of one particular kind.

6 Experimental Results

6.1 Experimental setup

To evaluate our algorithm, we have implemented the Dangerzone module for the bounded model checker CBMC 5.5.[1] It generates a danger specification from a given C program and implements a second-order SAT solver as discussed in [12] to obtain a proof. We ran the resulting prover on 50 programs from the loop acceleration category in SV-COMP 2016 [5]. We picked this specific category as it has benchmarks with deep bugs and we were interested in challenging our hypothesis that danger invariants are well-suited to expose deep bugs and can complement the capabilities of existing approaches such as BMC. Unfortunately we had to exclude programs that make use of arrays, since these are not yet supported by the synthesiser. In addition to this, we also introduced altered versions of the selected SV-COMP 2016 benchmarks with extended loop guards to create deeper bugs, challenging our hypothesis even further.

For each benchmark we try to synthesise both a partial danger invariant (i.e. a danger invariant, a ranking function, an initial state and Skolem functions witnessing the nondeterminism corresponding to partial correctness in Def. 5) and a total danger invariant (i.e. a danger invariant, an initial state and Skolem functions corresponding to total correctness in Def. 3). To provide a comparison point, we also ran two state-of-the-art bounded model checking (BMC) tools, CBMC 5.5 [7] and SMACK+CORRAL 1.5.1 [6] on the same benchmarks. In addition to this, we ran the benchmarks against CPAchecker 1.4 [9], the overall winner of SV-COMP 2015, and Seahorn 2.6 [8], the second-placed tool in the loops category after CPAchecker. We reproduced each tool's SV-COMP 2015 configuration, with small alterations to account for the benchmarks where we increased loop guards. Finally, we manually translated the benchmarks to be compatible with Microsoft's Static Driver Verifier Research Platform (SD-VRP [13]) with the Yogi 2.0 [14] back end. Yogi's main algorithms are Synergy, Dash, Smash and Bolt.

[1] `https://github.com/diffblue/cbmc/archive/bbae05d8faecfec18a42724e72336d8f8c4e3d8d.zip`

We say that a benchmark contains a deep bug if it is only reachable after at least 1'000'000 unwindings. Each tool was given a time limit of 300 s, and was run on a 12-core 2.40 GHz Intel Xeon E5-2440 with 96 GB of RAM. The full result table of these experiments is given in the extended version of the paper [10].

6.2 Discussion of results

The results demonstrate that the Dangerzone module outperforms all other tools on programs with deep bugs. It solves 37 (partial) and 38 (total) out of the 50 benchmarks in standalone mode, and 46 when used with CBMC. By itself, CBMC only finds 27, SMACK+CORRAL 24, CPAchecker 26 and Seahorn 31 bugs. This result can be explained by the fact that the complexity of finding a danger invariant is orthogonal to the number of unwindings necessary to reach it. Dangerzone's success is not determined by how deep the bug is, but by the complexity of the invariant describing it. As a result, we perform comparably on both deep and shallow bugs and are able to expose 18 out of the 20 deep bugs in the benchmark set. This supports our hypothesis that danger invariants are well-suited for this category of errors.

Danger invariants and BMC complement each other perfectly in our experiments and together solve 46 out of the 50 problems. We consider this further evidence for our hypothesis that danger invariants extend existing model checkers' capabilities to expose deep bugs.

6.3 Manually solving a danger constraint

As a case study we also tried using danger invariants to analyse a bug in Sendmail that has been proposed as a challenge for verification tools [15]. This program makes use of arrays, which our program synthesiser does not support. We decided that it would be interesting to see whether danger invariants could be used to semi-automatically prove the existence of such a difficult bug, and so wrote the danger invariant by hand. We then used CBMC to verify that the danger invariant we had written did indeed satisfy all of the criteria for a danger invariant as given in Def. 5, thereby proving the existence of the bug. This process was successful, with the verification step taking 0.23 s. We therefore believe that danger invariants could be used in semi-automatic tools to aid humans in finding complex bugs without the need for full blown automatic tools.

7 Related Work

Compositional may/must analysis. Compositional approaches to property checking such as [16] involve decomposing the whole-program analysis into several sub-analyses and summarising the results of these sub-analyses for later uses. The summaries are either may or must summaries.

The must summaries used in [16] (denoted $\phi_1 \xrightarrow{must} \phi_2$) are proofs that for every state $y \in \phi_2$, there exists a state $x \in \phi_1$ such that there is an execution trace from x to y. In the terminology of [17], this is a must$^-$ summary. The underapproximating nature of such summaries allows checking for bugs by inspecting the intersection between the must$^-$ set (the states reachable from the initial states via must$^-$ transitions) and the error states. Any state in this intersection must be reachable from an initial state, and therefore is a true bug. By contrast, Danger Invariants can be seen as a form of must$^+$ analysis, where we prove facts of the form $\phi_1 \xrightarrow{must^+} \phi_2$, which means that every $x \in \phi_1$ can reach a state $y \in \phi_2$. The two styles of must analysis are compared in Figure 6: to prove that an assertion A can be violated starting from initial states I, you can either use a must$^-$ analysis to find an underapproximation of the reachable states and show that these intersect with the error states, or you can use a must$^+$ analysis to find a non-empty underapproximation of the initial states that can reach an error state.

In [16], the authors use automated random testing techniques (DART) [18] to compute the must$^-$ summaries (required to show the existence of bugs). DART is based on single-path execution, which means that deep loops will cause the exploration of a large number of paths (corresponding to executing the loop once, twice, etc.), which may cause an exponential blow-up. As opposed to this approach, danger invariants are must$^+$ summaries which may encompass multiple paths through a loop, which can avoid exponential blow-up in many cases. Thus, the two approaches could be complementary.

$I \xrightarrow{must^-} \phi_2$	$\phi_1 \xrightarrow{must^+} \neg A$
Check that $\phi_2 \cap \neg A$ is non-empty.	Check that $\phi_1 \cap I$ is non-empty.

Fig. 6: Danger proofs using must$^-$ and must$^+$ analyses.

Temporal logic. With respect to the verification of temporal properties, a danger invariant for a loop with an assertion A essentially proves the CTL property $\models \mathbf{EF} \neg A$ over the loop. While there exist CTL verifiers based on a reduction to exist-forall quantified Horn clauses [19, 20], we specialise the concept for finding deep bugs and describe a modular constraint generation technique over arbitrary programs, rather than for transition systems.

Underapproximate acceleration. Another successful technique for finding deep bugs without false alarms is loop acceleration [21, 22]. This approach works by taking a single path at a time through a loop, computing a symbolic representation of the exact transitive closure of the path (an accelerator) and adding it back into the program before using an off-the-shelf bug finder such as a bounded model checker. Loop acceleration requires that each accelerated path can be represented in closed-form by a polynomial over the program variables, which is not always possible. In contrast, danger invariants are complete – a program has a corresponding danger invariant iff it has a bug.

Constraint Solving. There is a lot of work on the generation of linear invariants of the form $c_1x_1 + \ldots + c_nd_n + d \leq 0$ [23, 24]. The main idea behind these techniques is to treat the coefficients $c_1, \ldots, c_n, d$ as unknowns and generate constraints on them such that any solution corresponds to a safety invariant.

In [24], Colon et al. present a method based on Farkas' Lemma, which synthesises linear invariants by extracting non-linear constraints on the coefficients of a target invariant from a program. In a different work, Sharma and Aiken use randomised search to find the coefficients [24]. It would be interesting to investigate how these methods can be adapted for generating constraints on the coefficients $c_1, \ldots, c_n, d$ such that solutions correspond to linear danger invariants.

Doomed Program Locations. The term "doomed program point" was introduced in [25] and denotes a program location that will inevitably lead to an error regardless of the state in which it is reached. The notion is more restrictive than a danger invariant D. Our experiments revealed multiple unsafe benchmarks for which we could synthesise a danger proof, but no doomed program location exists.[2]

Error Invariants. The concept of error invariant [26] was introduced in order to localize the cause of an error in an error trace. An error invariant is an invariant for a position in an error trace that only captures states that will still produce the error. As opposed to an error invariant, a danger invariant is inductive and may describe multiple traces through the program.

Program Synthesis. Counterexample-Guided Inductive Synthesis (CEGIS) relies on inductive conjectures and refinement through counterexample information. This learning pattern is used in a multitude of learning applications, including Angluin's classic DFA learning algorithm L^* [27]. Syntax-Guided Synthesis (SyGuS) by Alur et al. is based on the same principle [28]. They employ a CEGIS loop with a grammar to restrict the space of possible programs. Our implementation focuses on concurrent synthesis of multiple danger constraint programs.

8 Conclusions

In this paper, we introduced the concept of *danger invariants* – the dual to safety invariants. Danger invariants summarise sets of traces that are guaranteed to reach an error state. As the size of a danger invariant is independent of the depth of its corresponding bug, it can enable bug finding techniques for which the computational effort is also independent of the depth of bugs, and thus have the potential to find deep bugs more efficiently. As an exemplar of an analysis using danger invariants, we presented a bug finding technique based on a synthesis engine.

[2] More details in the extended version [10].

References

1. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. (1977) 238–252
2. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design **19**(1) (2001) 7–34
3. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. (1994) 1512–1542
4. McMillan, K.L.: Lazy abstraction with interpolants. In: Computer Aided Verification (CAV). LNCS, Springer (2006) 123–136
5. SV-COMP 2016: `http://sv-comp.sosy-lab.org/2016/`.
6. Haran, A., Carter, M., Emmi, M., Lal, A., Qadeer, S., Rakamarić, Z.: SMACK+Corral: A modular verifier (competition contribution). In Baier, C., Tinelli, C., eds.: Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Volume 9035 of Lecture Notes in Computer Science., Springer (2015) 450–453
7. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2004) 168–176
8. Gurfinkel, A., Kahsai, T., Navas, J.: SeaHorn: A framework for verifying C programs (competition contribution). In Baier, C., Tinelli, C., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 9035 of Lecture Notes in Computer Science. Springer (2015) 447–450
9. Beyer, D., Keremoglu, M.: CPAchecker: A tool for configurable software verification. In Gopalakrishnan, G., Qadeer, S., eds.: Computer Aided Verification. Volume 6806 of Lecture Notes in Computer Science. Springer (2011) 184–190
10. David, C., Kesseli, P., Kroening, D., Lewis, M.: Danger invariants (extended version). `https://www.cs.ox.ac.uk/files/8323/danger-paper-extended.pdf`
11. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: Programming Language Design and Implementation (PLDI). (2008) 281–292
12. David, C., Kroening, D., Lewis, M.: Using program synthesis for program analysis. In: Logic for Programming, Artificial Intelligence, and Reasoning LPAR. (2015) 483–498
13. Ball, T., Bounimova, E., Levin, V., Kumar, R., Lichtenberg, J.: The Static Driver Verifier Research Platform. In: Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings. Springer, Berlin, Heidelberg (2010) 119–122
14. Nori, A.V., Rajamani, S.K.: An empirical study of optimizations in Yogi. In: International Conference on Software Engineering (ICSE), Association for Computing Machinery, Inc. (May 2010)
15. Dullien, T.: Exploitation and state machines. In: Infiltrate. (2011)
16. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: Principles of Programming Languages, POPL. (2010) 43–56
17. Ball, T., Kupferman, O., Yorsh, G.: Abstraction for falsification. In: Computer Aided Verification CAV. (2005) 67–81
18. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Programming Language Design and Implementation, PLDI. (2005) 213–223

19. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified Horn clauses. In: Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. (2013) 869–882
20. Beyene, T.A., Brockschmidt, M., Rybalchenko, A.: CTL+FO verification as constraint solving. In: 2014 International Symposium on Model Checking of Software, SPIN 2014, Proceedings, San Jose, CA, USA, July 21-23, 2014. (2014) 101–104
21. Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. In: Computer Aided Verification CAV. (2013) 381–396
22. Kroening, D., Lewis, M., Weissenbacher, G.: Proving safety with trace automata and bounded model checking. In: Formal Methods (FM). Volume 9109 of LNCS., Springer (2015) 325–341
23. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: Computer Aided Verification (CAV). (2003) 420–432
24. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: CAV. (2014) 88–105
25. Hoenicke, J., Leino, K.R.M., Podelski, A., Schäf, M., Wies, T.: It's doomed; we can prove it. In: FM 2009: Formal Methods. Springer (2009) 338–353
26. Ermis, E., Schäf, M., Wies, T.: Error invariants. In: Formal Methods, FM. (2012) 187–201
27. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2) (1987) 87–106
28. Alur, R., et al.: Syntax-guided synthesis. In: FMCAD. (2013)