More Effective Interpolations in Software Model Checking

Cong Tian*, Zhao Duan*, Zhenhua Duan*, and C.-H. Luke Ong†
*ICTT and ISN Lab, Xidian University, Xi'an 710071, P.R. China
ctian@mail.xidian.edu.cn, duanzhao@stu.xidian.edu.cn, zhhduan@mail.xidian.edu.cn
†Department of Computer Science, University of Oxford, UK
Luke.Ong@cs.ox.ac.uk

***Abstract*—An approach to CEGAR-based model checking which has proved to be successful on large models employs Craig interpolation to efficiently construct parsimonious abstractions. Following this design, we introduce new applications, *universal safety interpolant* and *existential error interpolant*, of Craig interpolation that can systematically reduce the program state space to be explored for safety verification. Whenever the universal safety interpolant is implied by the current path, all paths emanating from that location are guaranteed to be safe. Dually whenever the existential error interpolant is implied by the current path, there is guaranteed to be an unsafe path from the location. We show how these interpolants are computed and applied in safety verification. We have implemented our approach in a tool named INTERPCHECKER by building on an open source software model checker. Experiments on a large number of benchmark programs show that both the interpolations and the auxiliary optimization strategies are effective in improving scalability of software model checking.**

I. INTRODUCTION

Software model checking [1], [2] is an approach to program verification that promises accurate analysis with push-button automation. Model checking approaches can achieve precision because they are path-sensitive. On the flip side, because they often track too many facts, state explosion gets in the way of scalability.

An extensively studied method, called *Counterexample Guided Abstraction Refinement* (CEGAR) [4], [5], [6], [7], [8], can automatically tune the precision of the analysis using false positives. In a CEGAR analysis, predicate abstraction [9] is used to extract a coarse abstract model from a program. The model is iteratively refined by adding facts to make the abstraction precise enough to refute spurious counterexamples. These "facts" are predicates that relate values of program variables.

The scalability of CEGAR-based analyses depends crucially on the ability to efficiently analyze a false positive so as to learn from it a *small* set of sufficiently accurate predicates, and to use the discovered predicates parsimoniously. To this end, *Craig interpolation* [10] has been employed effectively to construct abstractions that are locally useful, and only those that are required for proving correctness [25], [5]. By integrating various techniques, notably lazy abstraction [28] and parsimonious abstraction via Craig interpolation [25], great strides have been made in the development of efficient model checking that scales to large programs. Software model checkers such as BLAST [5] and CPAChecker [13], [12] have achieved impressive success in recent software verification competitions [26], [27], [29], [30], [31]. However (see Section V for details), the development of *precise* and *scalable* model checking tools that are fit for real-world applications remains a daunting challenge.

Our approach to scalable CEGAR-based model checking [13] is to exploit Craig interpolation to learn abstractions that can systematically reduce the program state space which must be explored for a given safety verification problem. In addition to the interpolants for parsimonious abstraction [25] (which is called *reachability interpolants* in this paper for clarity), we introduce two new kinds of interpolants, called *universal safety interpolants* and *existential error interpolants*. A *universal safety interpolant* (or *safety interpolant* for short) is useful for determining whether all the paths emanating from a state are safe, without exploring all the possible branches from it; while an *existential error interpolant* (or *error interpolant* for short) is useful for determining whether there exists an unsafe path emanating from a state, without exploring all the possible branches from it. The safety interpolant at a location of a control flow graph (CFG) collects predicates that are relevant to a yes-instance of the safety verification, so that whenever the safety interpolant is implied by the current path, all paths emanating from this location are guaranteed to be safe. Dually, whenever the error interpolant at a location of a CFG is implied by the current path, there is guaranteed to be an unsafe branch from it, and so, one can immediately conclude that the program is unsafe. We show how safety interpolants and error interpolants are learnt from spurious error traces throughout the CEGAR-based program verification process. To maximise the effect of the proposed interpolations, we also present two kinds of optimizing strategies.

We have implemented the approach in a tool named INTERPCHECKER by augmenting the open source tool CPAChecker [12], [13] with the proposed interpolations and the optimizing strategies. To evaluate it, a large number of experiments on more than 58 million lines of C programs (mostly linux driver programs between 10 to 80 KLOC) have been carried out. Empirical results show that the proposed interpolations are effective in reducing the explored state space so that more

Zhenhua Duan is the corresponding author.

programs can be successfully verified within the given time bound. In particular, the experiments indicate that the tool is most effective when all three kinds of interpolations and the optimization strategies are applied together.

The rest of the paper is structured as follows. Section II presents the preliminaries and a motivating example. In Section III, we introduce error interpolants and safety interpolants, and discuss their formalization and use in detail. Two optimizing strategies are then presented in Section IV. In Section V, we present an empirical evaluation of our approach. Finally, Section VI discusses related work and Section VII concludes the paper.

II. Preliminaries

This section briefly presents control flow graphs, abstract reachability trees, Craig interpolation, and an interpolation-aided CEGAR approach to program verification with a motivating example.

A. Control Flow Graphs

A *control flow graph* (CFG) is a directed graph that captures the control flow of a program. Formally, a CFG is a tuple $G = (L, T, l_0, f)$, where L is the set of program locations, $l_0 \in L$ is the initial location, $f \in L$ is the final location, $T \subseteq L \times Ops \times L$ is the transition relation, and *Ops* is the set of instructions. An instruction $op \in Ops$ is (i) a basic assignment statement, (ii) an assume predicate corresponding to the condition that must hold for the control to flow across an edge, (iii) a function call with call-by-value parameters, or (iv) a return statement. A transition $t \in T$ is a triple $t = (l, op, l')$ denoting the flow of control from l to l' by executing the instruction op. An example CFG is depicted on the LHS of Fig. 1.

To clearly express our approach, we further decorate a CFG with three *location attributes*, E, S, and R respectively, and one *transition attribute* W. Intuitively, $l(E)$, $l(S)$, and $l(R)$, respectively, denote the *E-Interp*, *S-Interp*, and *R-Interp* (to be defined later) at a location l; and $t(W)$ gives the *weight* of the transition t. How these attributes are initialized and updated throughout the verification process will be discussed later.

B. Abstract Reachability Trees

An *abstract reachability tree* (ART) is generated by unwinding a CFG. An ART $A = (S_A, E_A)$ obtained from a CFG $G = (L, T, l_0, f)$, consists of a set S_A of abstract states and a set E_A of edges. An *abstract state* $s \in S_A$ is a triple $s = (l, c, p)$ where l is a location in the CFG, c is the current call stack (i.e. a sequence of CFG locations representing return addresses), and p is an abstract predicate indicating the reachable region of the current state. (As we shall see, the reachable region, i.e. $s[2]$, of a state s is determined by the reachable interpolant, *R-Interp*, of the location $s[0]$ in the CFG.) Given two states s and s', we say s is *covered* by s' just if $s[0] = s'[0]$, $s[1] = s'[1]$, and $s[2] \rightarrow s'[2]$. (Notation: for a tuple e, we write $e[i]$ for the i-th component of e.) Further, if s is covered by s' and all the future of s' (i.e. all abstract states reachable from s') has been explored, then the future of s can be saved from exploring (because the result will be the same as the future of s'). An edge $e \in E_A$ is a triple $e = (s, op, s')$ where s and s' are abstract states in S_A, and op is an instruction in *Ops*, including assignment expression, assume expression, function call and return expression.

A branch (path) Π of an ART, denoting a possible execution of the program, is a finite alternating sequence of states and edges, $\Pi = \langle s_0, e_0, \cdots, e_{n-1}, s_n \rangle$, such that for all $0 \leq i < n$, $e_i[0] = s_i$ and $e_i[2] = s_{i+1}$. The length of a path is the number of edges occurring in it. Given a path Π of an ART, we write $P_f(\Pi)$ for the path formula

$$\textsc{ssa}(e_0[1]) \wedge \cdots \wedge \textsc{ssa}(e_{n-1}[1])$$

obtained from Π. Here $\textsc{ssa}(op)$ is the static single assignment (SSA) form [5] of an operation op where every variable occurring in Π is assigned a value at most once. In this paper, the SSA form is obtained by introducing a new subscript to a variable whenever it is newly assigned. Fig. 2 gives an example of an ART.

C. Craig Interpolation

Given two formulas A and B such that $A \wedge B$ is unsatisfiable, a *Craig interpolant* C is a formula that satisfies the following conditions: 1) A implies C; 2) the conjunction $C \wedge B$ is unsatisfiable; and 3) all variables in C are common to A and B. For convenience, we use

$$C = \mathit{Craig}(A, B)$$

to denote the Craig interpolant of formulas A and B. In software model checking, Craig interpolation has been used successfully with abstraction refinement so that more precise abstractions can be constructed from spurious counterexamples in order to eliminate them. For clarity, the resulting interpolants are called *reachability interpolants*, or *R-Interp* for short, in this paper.

Definition 1 (Reachability Interpolant, R-Interp*):* Let $\Pi = \langle s_0, e_0, \cdots, e_{n-1}, s_n \rangle$ be a spurious path of an ART. For $0 < i < n$, set $\mathit{R\text{-}Interp}(s_i[0]) := \mathit{R\text{-}Interp}(s_i[0]) \cup \mathit{Craig}\big(P_f(s_0, \cdots, s_i), P_f(s_i, \cdots, s_n)\big)$. Note that for every location l of a CFG, initially, $\mathit{R\text{-}Interp}(l) = \{\mathit{true}\}$.

D. Interpolation-Aided CEGAR Verification Approach

We present a version of CEGAR safety verification that uses *R-Interp*. The procedure starts with the most abstract model (no predicates are considered) and checks whether a counterexample (i.e. error path) can be detected. If no error path is found, the procedure terminates, reporting the non-existence of counterexamples. Otherwise, if a counterexample path Π is found, we check satisfiability of the relevant path formula, i.e. $P_f(\Pi)$, to determine if Π is genuine. In case $P_f(\Pi)$ is satisfiable, the procedure terminates by reporting Π as a counterexample. If $P_f(\Pi)$ is unsatisfiable, new predicates are discovered [25] at each location involved in Π according to Definition 1. Observe that at each location of the path Π, we

infer the relevant predicates as an interpolant between the two formulas that define the past and the future segments of the path. Each interpolant, $R\text{-}Interp(l)$, is a relationship between current values of program variables, and is relevant only at the particular location l.

In the following, we use an example program `Exa.c` shown in the LHS of Fig. 1 to illustrate how *R-Interp*-aided CEGAR works and motivate our work at the same time. In program `Exa.c`, if the code in Line 16 is executed, the program is unsafe; otherwise, it is safe. The RHS of Fig. 1 gives the CFG of the program where L_{11} is the error location. Thus the problem of whether the program is unsafe is reduced to the reachability analysis of L_{11} in the CFG.

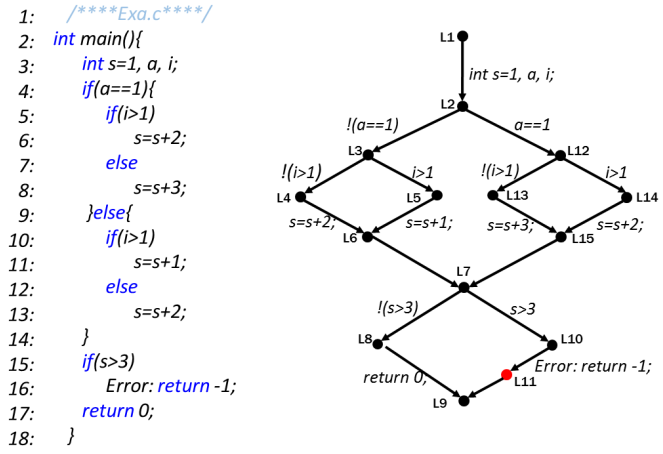


Fig. 1. Program `Exa.c` and its CFG

In the *R-Interp*-aided CEGAR approach, by depth-first traversal of the CFG, the branch P_1 in the ART of Fig. 2 is explored in the first run. In Fig. 2, we annotate each abstract state s (i.e. node) of the ART with the *R-Interp*, $R : \mathcal{R}$, and reachability predicate, $p : \psi$, at that abstract state; i.e. $R\text{-}Interp(s[0]) = \{\phi \mid \phi \in \mathcal{R}\}$ and $s[2] = \psi$. Note that for clarity, if the $R\text{-}Interp$ at a location is updated, we ignore the element $true$ in Fig. 2. Note that for every state s in P_1, we have $R\text{-}Interp(s[0]) = \{true\}$ and $s[2] = true$. Then, the verification process proceeds straightforwardly to the second run where P_2 is explored since L_{11} is not reached in P_1. Even though the error location is reached in P_2, we need to determine if the path is genuine or spurious.

To do so, we check satisfiability of the path formula $P_f(P_2) =$ "$(s_1 = 1) \wedge !(a_1 == 1) \wedge !(i_1 > 1) \wedge (s_2 = s_1 + 2) \wedge (s_2 > 3)$". Obviously $P_f(P_2)$ is unsatisfiable since $s_2 = 3$ and $s_2 > 3$ are contradictory. Thus P_2 is a spurious counterexample. As a result, the $R\text{-}Interp$ at each location of the CFG involved in P_2 is updated, and $s[2]$ at each state of P_2 is updated, accordingly, as illustrated in Fig. 2. Note that $p = false$ at L_{10} of P_2 indicates that L_{10} cannot be reached. Subsequently, in the third run, P_3 is explored. When L_8 is analyzed, it is found that L_8 in P_3 is covered by L_8 in P_1. Thus, we can conclude that P_3 is not a counterexample at L_8 without further exploration. This process is repeated until a real counterexample, i.e. P_8, is found. Note that, similar to L_8 in P_3, L_8 in P_5 and P_7 are also covered by L_8 in P_1.

In summary, in Fig. 2, in the worst case, 3 states are saved from being explored before the real counterexample P_8 is found. Nevertheless, as we can see, the explored state space is still large. Thus, we are motivated to seek further reduction of the state space by using more interpolations throughout the verification process. Two ideas can be gleaned from this example. (1) If we already know L_{10} in P_2 is unreachable, then L_{10} in P_4 is also unreachable since the value of s at L_7 of P_4 is obviously smaller than the value of s at L_7 of P_2. (2) By analyzing P_6, we can infer that the error location is reachable from L_{15} if $s > 3$ holds there. Thus, when exploring L_{15} in P_7, we can conclude that a real counterexample can certainly be found if the path formula of the prefix from L_1 to L_{15} satisfies '$s > 3$'.

III. More Interpolations

To further reduce the state space to be explored, *universal safety interpolation* and *existential error interpolation*, which we abbreviate to *S-Interp* and *E-Interp* respectively, are formalized in this section.

A. Universal Safety Interpolation

*Definition 2 (*Universal Safety Interpolation*,* S-Interp*):* Let l be a location of a CFG. The *universal safety interpolation* (or *safety interpolation* for short) of l is a pair $S\text{-}Interp(l) = (F, I_s)$, where F is a variable with value f or h indicating whether the interpolant is *full* or *half*; and I_s is a conjunction of predicates.

Initialization: For each location l in a CFG, the default value of its safety interpolant is:

$$S\text{-}Interp(l) := \begin{cases} (f, false) & \text{if } l \text{ is an error location} \\ (f, true) & \text{if } l \text{ is a final location} \\ (h, true) & \text{otherwise} \end{cases}$$

Update S-Interp: The safety interpolant at each location is updated whenever a spurious counterexample is found (in an ART). Let $\Pi = \langle s_0, e_0, \cdots, e_{n-1}, s_n \rangle$ be a prefix of a spurious path $\langle s_0, e_0, \cdots, e_{m-1}, s_m \rangle$ where s_n is the last reachable state and $n < m$. For each $0 < i \leq n$, the first component (F-value) of the safety interpolant at $s_i[0]$, i.e. $F(s_i[0])$, is updated by:

$$F(s_i[0]) := \begin{cases} f & \text{if } S\text{-}Interp \text{ of all successors of } s_i[0] \\ & \text{are full, or } i = n \\ h & \text{otherwise.} \end{cases}$$

The second component (I_s-value) of the safety interpolant at $s_i[0]$, i.e. $I_s(s_i[0])$, is updated by:

$$I_s(s_i[0]) := \begin{cases} I_s(s_i[0]) \wedge Craig(A, B) & \text{if } 0 < i < n \text{ and} \\ & F(s_{i+1}[0]) = f \\ I_s(s_i[0]) \wedge !P_f(s_i, e_i, s_{i+1}) & \text{if } i = n \end{cases}$$

where $A = P_f(s_0, \cdots, s_i)$ and $B = P_f(s_i, e_i, s_{i+1}) \wedge !(I_s(s_{i+1}[0]))$.

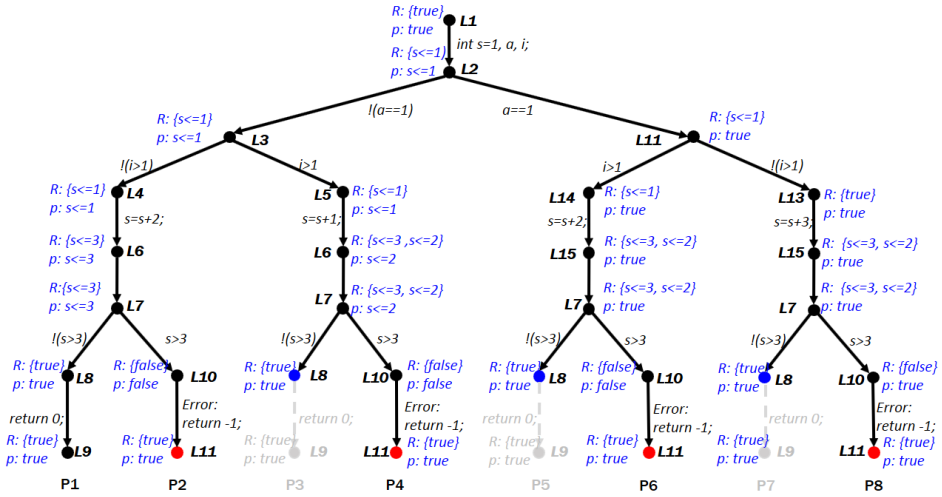


Fig. 2. Explored paths

Uses: Safety interpolants are helpful in checking whether all the paths departing from a state are safe without exploring all the possible branches emanating from it. For any prefix-path $\Pi = \langle s_0, e_0, \cdots, e_{i-1}, s_i \rangle$ where $i \geq 0$, we can conclude that all the paths that have Π as a prefix are safe if

$$F(s_i[0]) = f \text{ and } P_f(s_0, e_0, \cdots, e_{i-1}, s_i) \rightarrow I_s(s_i[0])$$

When applying safety interpolation in verifying program `Exa.c`, the state space explored is depicted in Fig. 3. First, P_1 is explored as usual. When P_2 is explored, we still check whether it is spurious and find that L_7 is the last reachable state. Then, by the update rule of *S-Interp*, we have

$$F(L_7) = f, \text{ and } I_s(L_7) = true \wedge (s \leq 3).$$

Subsequently, the *S-Interp* of L_6, L_4, L_3 and L_2 are also updated as shown in Fig. 3. Note that the *S-Interp* of L_1 is not updated since the *S-Interp* of L_2 is half. After that, when exploring L_6 of P_3, we can conclude that P_3 is safe since

$$F(L_6) = f, \text{ and } P_f(L_1 \cdots L_6) \rightarrow I_s(L_6)$$

holds. Here $P_f(L_1 \cdots L_6) =$ "$(s_1 = 1) \wedge (!(a_1 == 1)) \wedge (i_1 > 1) \wedge (s_2 = s_1 + 1)$" and $I_s(L_6) =$ "$s_2 \leq 3$" (obtained in P_2). This process is repeated until the real counterexample P_8 is found. Obviously, more states are saved from being explored than using only *R-Interp* as shown in Fig. 3. Note that in this ART, all the grey states (unexplored) reachable from the states in green are pruned because of safety interpolation.

B. Existential Error Interpolations

In contrast to universal safety interpolation, *existential error interpolation* is for checking whether there exists an unsafe path departing from the current state without exploring all the future ones.

*Definition 3 (*Existential Error Interpolation, E-Interp*):* Let l be a location of a CFG. The *existential error interpolation* (or *error interpolation* for short) at l is $E\text{-}Interp(l) = I_e$, where I_e is a disjunction of predicates.

Initilization: For each location l of a CFG, its default error interpolant is:

$$E\text{-}Interp(l) := \begin{cases} true & \text{if } l \text{ is an error location} \\ false & \text{if } l \text{ is a final location} \\ false & \text{otherwise} \end{cases}$$

Update E-Interp: Error interpolants are also updated whenever a spurious counterexample is found. Given a spurious counterexample $\Pi = \langle s_0, e_0, \cdots, e_{n-1}, s_n \rangle$ where $s_n[0]$ is an error location, let $\Pi' = \langle s_i, e_i, \cdots, e_{n-1}, s_n \rangle$ with $0 < i \leq n$ be the maximal feasible suffix of Π. The $E\text{-}Interp$ of locations involved in $\Pi = \langle s_0, e_0, \cdots, s_{i-1}, e_{i-1}, s_i \rangle$ are updated by:

$$E\text{-}Interp(s_i[0]) := E\text{-}Interp(s_i[0]) \vee Craig(A_1, B_1)$$

where $A_1 = P_f(s_i, \cdots, s_n) \wedge E\text{-}Interp(s_n[0])$ and $B_1 = P_f(s_0, \cdots, s_i)$. For each $0 < j < i$, set

$$E\text{-}Interp(s_j[0]) := E\text{-}Interp(s_j[0]) \vee Craig(A_2, B_2)$$

where $A_2 = P_f(s_j, e_i, s_{j+1}) \wedge E\text{-}Interp(s_{j+1}[0])$ and $B_2 = P_f(s_0, \cdots, s_j)$.

Uses: Error interpolants are useful for checking whether there exists an unsafe path departing from a state without exploring all the possible branches emanating from it. For any prefix-path $\Pi = \langle s_0, e_0, \cdots, e_{i-1}, s_i \rangle$ with $i \geq 0$, it can be concluded that there exists at least one unsafe path with Π being prefix if

$$P_f(s_0, e_0, \cdots, s_i) \text{ is satisfiable, and}$$
$$P_f(s_0, e_0, \cdots, s_i) \rightarrow E\text{-}Interp(s_i)$$

Now we show how error interpolation is used in verifying program `Exa.c`. As shown in Fig. 4, when P_2 is explored as a spurious counterexample, we first find out the maximal

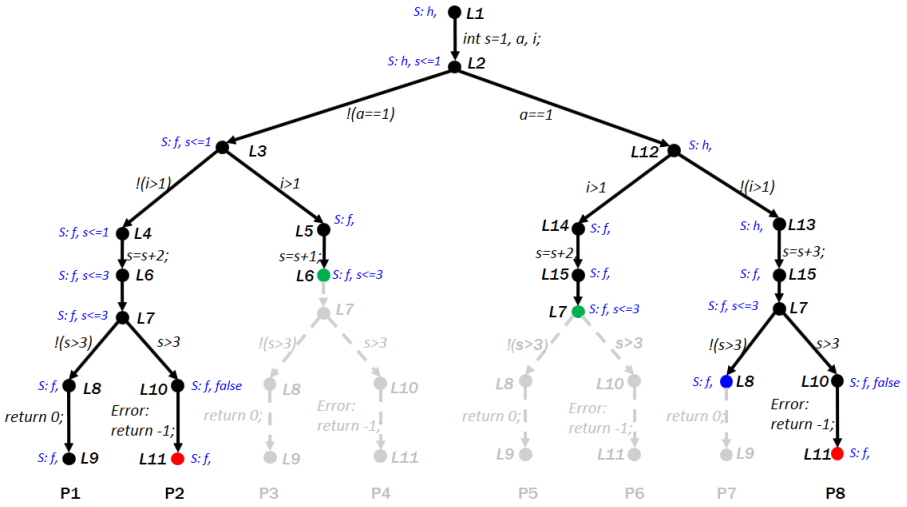


Fig. 3. Explored paths under *R-Interp* and *S-Interp*

feasible suffix $\langle L_{10}, L_{11}\rangle$ of P_2. Then, $E\text{-}Interp(L_{10})$ is updated by:

$$\begin{aligned} & E\text{-}Interp(L_{10}) \\ = \; & E\text{-}Interp(L_{10}) \vee Craig\big(true, P_f(L_1, \cdots, L_{10})\big) \\ = \; & true \end{aligned}$$

Then, $E\text{-}Interp$s of locations L_7, L_6, L_4, L_3, and L_2 are updated in order:

$$\begin{aligned} E\text{-}Interp(L_7) &= s > 3 \\ E\text{-}Interp(L_6) &= s > 3 \\ \\ E\text{-}Interp(L_4) &= s > 1 \\ E\text{-}Interp(L_3) &= s > 1 \\ E\text{-}Interp(L_2) &= s > 1 \end{aligned}$$

When exploring L_{15} of P_7, we can conclude that there exists at least one feasible unsafe path since

$$P_f(L_1 \cdots L_{15}) \rightarrow E\text{-}Interp(L_{15})$$

holds. Here $P_f(L_1 \cdots L_{15}) =$ "$(s_1 = 1) \wedge (a_1 == 1) \wedge (!(i_1 > 1)) \wedge (s_2 = s_1 + 3)$" and $E\text{-}Interp(L_{15}) =$ "$s_2 > 3$". The eventually explored state space is depicted in Fig. 4. The states in grey are not explored while the ones reachable from the yellow state are pruned because of error interpolation.

C. Interpolations Together

Now we show how the three kinds of interpolations, *R-Interp*, *S-Interp*, and *E-Interp*, work together to reduce state space to be explored for checking safety properties of programs.

Given a CFG whose locations are enriched with default values of *R-Interp*, *S-Interp*, and *E-Interp*, we produce the ART for exploring a real counterexample by starting from the root, i.e. $s_0 : (l_0, -, true)$. The flowchart in Fig. 5 gives a bird's eye view of our approach to safety verification with reachability, safety and error interpolations. When a state $s : (l, c, p)$ is being explored and l is not an error location:

(1) Reversely traverse the current path for other possibilities if one of the following three conditions holds:
   - $p = false$;
   - $p \neq false$, $F(l) = f$, and $P_f(s_0, \cdots, s) \rightarrow I_s(l)$; or
   - $p \neq false$ and s is covered by a visited state s'.

(2) Report the program is unsafe, if $p \neq false$ and $P_f(s_0, \cdots, s) \rightarrow E\text{-}Interp(l)$.

(3) Explore the succeeding state $s'' : (suc(l), c, p')$, otherwise.

When l of the current state $s : (l, c, p)$ is an error location, we first check whether the current path $\Pi = \langle s_0, \cdots, s\rangle$ is spurious. If Π is not spurious, we conclude that the program is unsafe. Otherwise, by **update S-Interp**, **update E-Interp**, and **update R-Interp** (Definition 1), the *S-Interp*, *E-Interp*, and *R-Interp* of locations involved in Π are updated, respectively. Subsequently, we reversely track the current path for other possibilities and treat a new current state $s : (l, c, p)$ in the same way until the program is reported as unsafe or there are no states can be explored (the program is safe).

With the three interpolations working together, we apply the verification procedure to the example program `Exa.c`. In the worst case, when the program is proved unsafe, we present the state space explored in Fig. 6. In this Figure, the unexplored states reachable from a state in green are pruned because of safety interpolants, and those reachable from a state in yellow are pruned because of error interpolants. As we can see, more states are saved from being explored than *R-Interp*, *R-Interp+E-Interp*, or *R-Interp+S-Interp*.

IV. Optimization

This section presents optimizing strategies in two directions: pruning, and accelerating formation of full safety-interpolants by *weight-guided search strategy*.

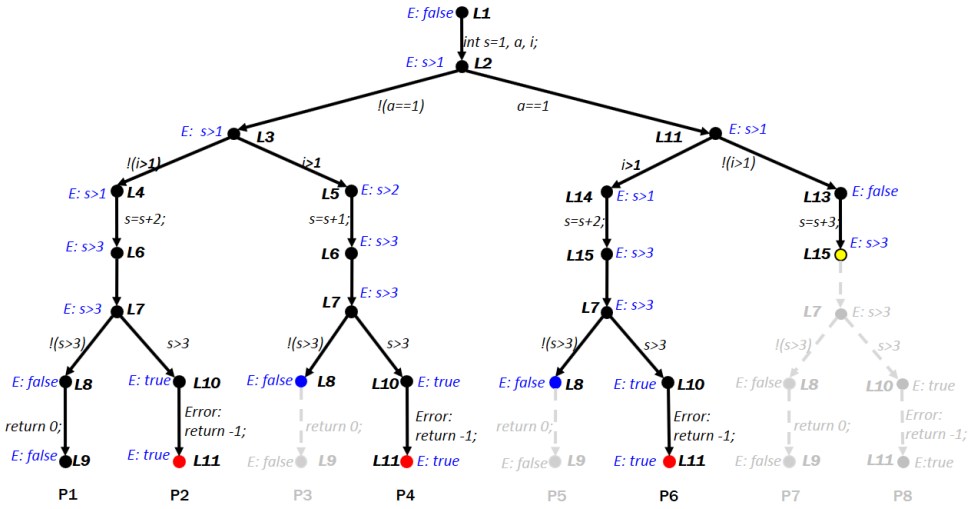


Fig. 4. Explored paths under *R-Interp* and *E-Interp*

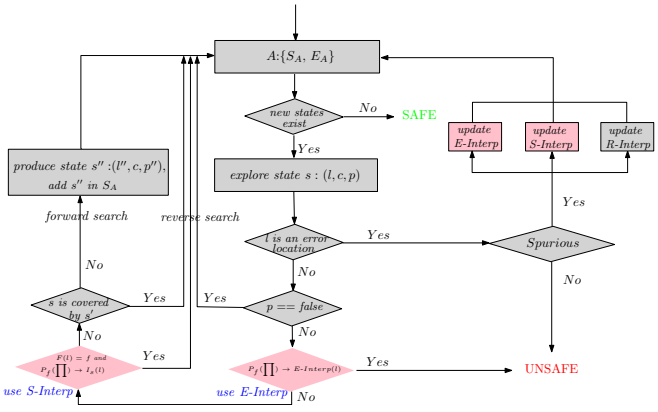


Fig. 5. Interpolations work together

A. Pruning CFG

When verifying real-world programs, there may exist some locations in a CFG which can never reach any error location. For instance, for the example program `Exa.c` in Fig. 1, L_8 and L_9 in the CFG can never reach the error location L_{11}. To avoid exploring these locations when verifying the program, we prune the CFG by removing these locations and the relative control flow edges before generating the ART. To do that, we start from the error location and traverse the CFG *against* the flow of control in depth-first order; then we remove all of edges and locations which are not visited. With the aid of such a pruning strategy, the state space explored when utilizing all the three kinds of interpolations shown in Fig. 6 can be further reduced by eliminating the suffix of P_1 starting from L_7, as depicted in Fig. 7.

B. Weight-Guided Search Strategy

As discussed in the previous section, a safety interpolant works only when it is full. Hence, the earlier full safety-interpolants are formed, the better the effect will be. The intuition is that if one side of a branch is explored, we expect to explore the other side as early as possible so as to form full interpolants. To achieve the goal, we introduce an attribute *weight* to transitions of a CFG. When generating an ART, the branch with the largest weight will be explored first. Note that the default weight value of each transition of a CFG is undefined (denoted as $\perp$).

Throughout the verification process, for a transition $t : (l, op, l')$, we reset

- $weight(t) = 0$ if $F(l')$ is changed from h to f;
- $weight(t) = |\{(l', -, -) \mid (l', -, -)$ is a succeeding transition of $t\}| - 1$ if l' is the last reachable state in the current path of the ART and $F(l') = h$.
- $weight(t) = \Sigma_{weight((l',-,-)) \neq \perp} weight((l', -, -)) + |\{(l', -, -) \mid weight((l', -, -)) = \perp\}|$ for any $t : (l, op, l')$, if the weight value of some transition departing from l' is changed.

Note that a weight value can be an integer larger than 0 (denoted (>0)), $\perp$, and 0. Here we decree: $(>0) > \perp > 0$. If the weight values of all the possible transitions are the same, we just randomly explore one the them.

As an example, for the CFG in Fig. 8, the weight of all edges are undefined, initially. When P_1 is explored as shown in Fig. 8 (1), the weight of all the reachable edges are updated. Then, P_2 is explored as shown in Fig. 8 (2). Since L_{11} is unreachable, no weights are updated. Subsequently, in the third run, P_3, the real counterexample, is explored under the rule since the weight values of both $(L_2, s > 1, L_4)$ and $(L_6, !(a < 3), L_{11})$ are $\perp$ while those of their opposites are both 0. Note that without the guidance of weight values, in the worst case, 4 paths are required to be explored in order to find the real counterexample.

V. Implementation and Experiments

We have implemented the proposed interpolations and optimization strategies in a tool called InterpChecker,

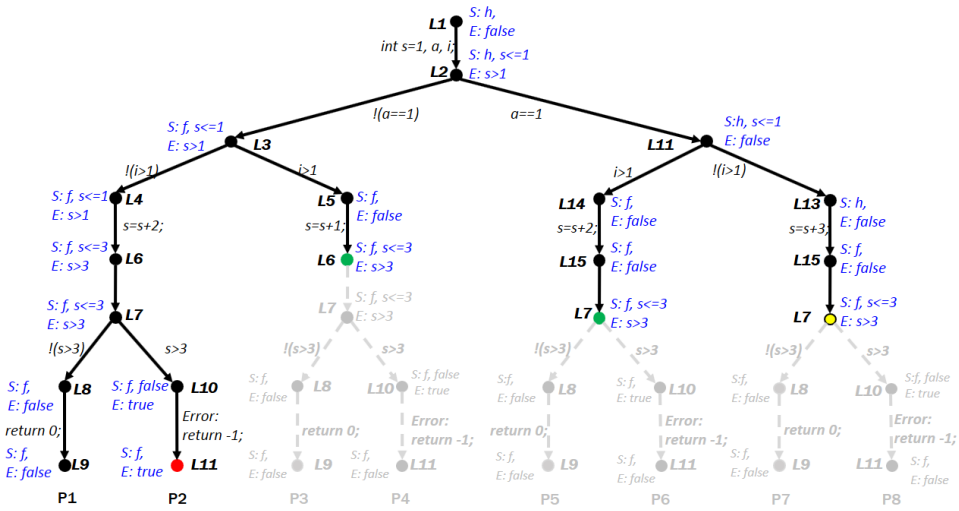


Fig. 6. Explored paths with all interpolations

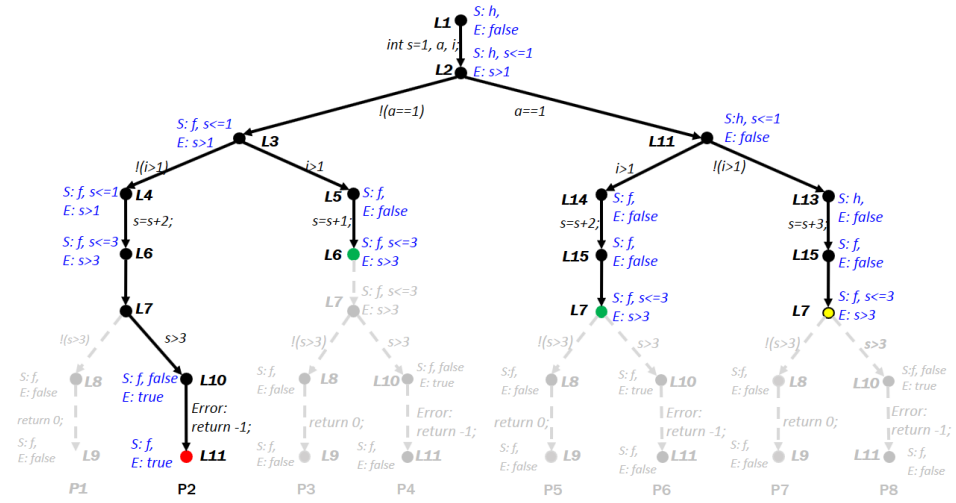


Fig. 7. Explored paths by pruning strategy

by building on the open source software model checker CPAChecker [13] (ABE configuration [16]) which supports reachability interpolation aided CEGAR verification. Our tool augments CPAChecker with safety interpolations and error interpolations, as well as the optimizing strategies proposed in Section IV. Note that ABE configuration of CPAChecker combines predicate abstraction with CEGAR to verify the programs. Almost all of the new features (some of them are not publicly available) [38], [39] of CPAChecker are also implemented based on this configuration.

To evaluate the proposed interpolations in the safety verification of programs, we selected 11 packages, as shown in Table I, from the benchmark suite of SV-COMP[1] where a large number of programs cannot be successfully verified within a specified time bound. These 11 packages constitute the category "Device Drivers Linux 64" in SV-COMP.

[1]https://github.com/sosy-lab/sv-benchmarks/tree/master/c

All experiments in this paper were done on a Linux virtual machine, which is configured on a PC running octa-core Windows 7 with 4 GHz and 64GB RAM. The virtual machine applies Ubuntu 12.04 LTS operation system with 4GHz and 4GB RAM.

Table I describes the results of the original CPAChecker (ABE) and another three tools, Smack+Corral [36], UAutomizer[5], and SATABS [37], which perform well in the competitions. The third column of Table I gives the number of programs contained in each package, and the fourth column gives the total number of lines of programs in each package. The remaining four columns, give the respective numbers of programs which are successfully verified in 15 mins, using CPAChecker (ABE), Smack+Corral, SATABS, and UAutomizer. As shown in Table I, CPAChecker (ABE) performs best with still 33.1% of the programs fail to be verified within the time bound.

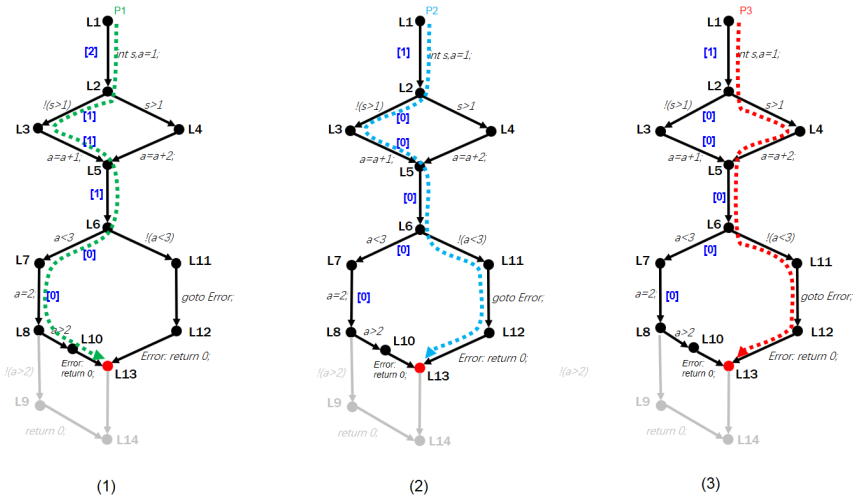


Fig. 8. Exploration with weight

TABLE I
BENCHMARK PROGRAMS

No.	Package-names	#Programs	#KLOC	CPAChecker(ABE)	Smack+Corral	SATABS	UAutomizer
				#Suc.	#Suc.	#Suc.	#Suc.
1	ldv-linux-3.7.3	11	246.7	7	6	0	0
2	ldv-challenges	15	448.8	3	3	0	0
3	ldv-validator-v0.6	21	212.8	10	13	2	0
4	ldv-validator-v0.8	27	265.5	7	20	0	0
5	ldv-linux-3.12-rc1	40	478.2	11	28	5	0
6	ldv-linux-3.0	41	755.2	25	35	15	13
7	ldv-consumption	163	2720.8	74	24	85	3
8	ldv-commit-tester	56	477.0	34	34	29	20
9	ldv-linux-3.16-rc1	159	2134.7	45	60	97	0
10	ldv-linux-4.2-rc1	432	10040.8	53	107	13	26
11	ldv-linux-3.4-simple	1163	40763.4	1155	1087	1099	689
Total		2128	58543.7	1424 66.9%	1417 66.6%	1345 63.2%	751 35.3%

The most recent SV-COMP competitions [26], [27] also show that CPAChecker [5] was the strongest performer on this category[2]. Hence we benchmark the performance of our implementations of safety interpolations and error interpolations against CPAChecker (ABE). By comparison with CPAChecker (ABE), it can embody directly the advantage of our approach. Note that the result of CPAChecker presented in Table I might be inconsistent with the one reported in the competitions because of different experimental environment.

To examine the effectiveness of the various interpolations, we verify the programs in Table I using our tool in 5 different modes, namely, R+E, R+S, R+S+E, R+S+W, and R+S+E+W, where R, E and S denote *R-Interp*, *E-Interp*, and *S-Interp* respectively; and W indicates the *weight*-guided search strategy. (Recall that *weight*-based searching strategy is only for the formation of safety interpolants.) Table II presents the verification results where the sub-column #Suc. gives the number of programs that are verified successfully, and the sub-column #T.o. gives the sum of programs that failed to be verified within the time bound of 15 mins. It is emphasized that a program is successfully verified with a tool indicates that the verification results (SAFE or UNSAFE) is correctly reported within the given time bound. That is false positives or false negatives are not counted in the sub-column #Suc.

From the experimental results, we observe that:

(1) Verification using each of R+E (*R-Interp* and *E-Interp*) and R+S (*R-Interp* and *S-Interp*) is more accurate than verification using only R (*R-Interp*).
(2) More programs are verified using R+S+E than either R+E or R+S.
(3) Verification using R+S+W is more accurate than R+S.
(4) Verification using R+S+E+W is the most accurate.

Thus we can say that each of the proposed interpolations and the optimization strategies improves the *accuracy, and hence effectiveness*, of program verification. Further, false positive may occur in principle since the interpolations are over-approximations of a program. However, our experience shows that SMTInterpol (tool for computing Craig interpolation in CPAChecker) always produces good predicates in practice. In the experimental results, no false positives are introduced because of the new interpolations.

[2]This category is not included in SV-COMP 2017.

TABLE II
COMPARING WITH R

No.	R(CPAChecker ABE)		R+E		R+S		R+S+E		R+S+W		R+S+E+W	
	#Suc.	#T.o.	#Suc.	#T.o.	#Suc.	#T.o.	#Suc.	#T.o.	#Suc.	#T.o.	#Suc.	#T.o.
1	7	4	7	4	7	4	7	4	7	4	7	4
2	3	12	3	12	3	12	3	12	4	11	4	11
3	10	11	10	11	10	11	10	11	10	11	11	10
4	7	20	7	20	7	20	8	19	9	18	10	17
5	11	29	12	28	13	27	12	28	15	25	16	24
6	25	16	25	16	25	16	30	10	25	16	29	11
7	74	89	74	89	76	87	76	87	78	85	77	86
8	34	22	39	17	39	17	41	15	39	17	40	16
9	45	114	48	111	67	92	68	91	76	83	79	80
10	53	379	58	374	61	371	64	368	78	354	80	350
11	1155	8	1156	7	1155	8	1156	7	1157	6	1157	6
Total	1424 66.9%	704 33.1%	1439 67.6%	689 32.4%	1463 68.8%	665 31.2%	1475 69.3%	653 30.7%	1498 70.4%	630 29.6%	1510 71%	618 29%

TABLE III
TIME CONSUMPTION

No.	Time consumption (s)					
	R	R+E	R+S	R+S+E	R+S+W	R+S+E+W
1	3950.1	3872.7	4582.1	4675.7	4511.7	4568.5
2	10951.6	10935.6	10944.5	10960.6	10025.3	10029.3
3	10559.1	10355.3	10533.5	9863.3	10735.2	9877.5
4	18421.5	18423.6	18418.6	17629.2	17069.7	16216.5
5	26510.1	25927.6	25210	25514.3	23058.4	22221.5
6	14586.6	14617.1	14606.2	10198.9	15122.9	11260.5
7	82261.3	82035.2	80796.7	80852.4	79918.4	80698.6
8	20105.1	15799.5	15752.7	14077	15633.4	14828.7
9	105364.2	102110.2	84636.7	84086	77140	74989.9
10	344786.1	340653.6	339106	338033.6	325391.5	325703
11	8816.5	7822.3	8772.5	7894.1	6287.9	6273.6
Total	646312.2	632552.7	613359.5	603785.1	584894.4	576667.3

In addition to the number of programs which are successfully verified with different kinds of interpolations, we are also interested in the ratio of programs successfully verified using *S-Interp* and *E-Interp*, to those successfully verified using the original CEGAR+*R-Interp* (CPAChecker ABE). Similarly we are interested in the ratio of programs that fail to be verified using *S-Interp* and *E-Interp*, to those that fail to be verified using the original CEGAR+*R-Interp*. Fig. 9 (a) shows the percentage ($N_{\text{suc}}/1424$) of the programs verified under each mode; and Fig. 9 (b) shows the percentage ($N_{\text{fail}}/704$) of the programs that fail to be verified under each mode. Note that N_{suc} is the number of verified programs (in the respective modes) from the original set of R-$Interp$-verified 1424 programs, and N_{fail} is the number of the unverified programs (in the respective modes) from the original set of R-$Interp$-unverified 704 programs.

As shown in Fig. 9 (a), 100% of the programs verified under *R-Interp* remain successfully verified under each of R+E, R+S, and R+E+S modes, and 98.8% of them remain successfully verified under R+S+W, and R+E+S+W. The rate is lower than the modes without weight-guided searching strategy, since the order in which the branches are explored are changed. As shown in Fig. 9 (b), the ratios of the unverified programs under R+E, R+S, and R+E+S to the unverified programs under *R* are in descending order, and the rate is lower when the weight strategy is utilized.

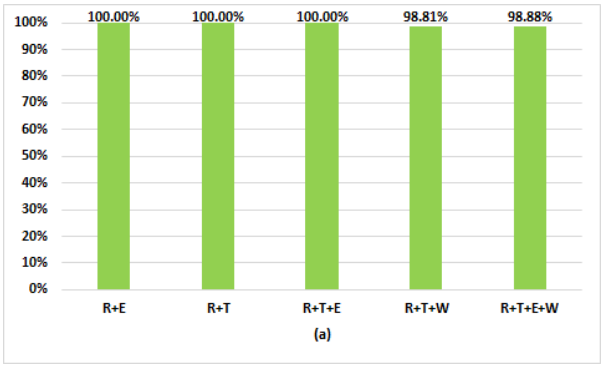
We also compare the time consumption of each mode. As shown in Table III, R+E and R+S take less time than R; R+E+S takes less time than both R+E and R+S; R+S+W takes less time than R+S; and R+S+E+W takes less time than all others. Thus, both the interpolations and the optimization strategy are useful in improving the runtime efficiency of software model checking.

VI. RELATED WORK

In recent years, Craig interpolation has been extensively applied to software model checking, symbolic execution, and testing. An important advantage of these applications is a much reduced program state space.

A. Program Verification

In pioneering work [14], McMillan computes interpolants to build unbounded symbolic model checking of finite state systems according to the refutations (counterexamples) produced in bounded model checking. The approach was extended to the verification of infinite state systems in [11] by employing *lazy abstraction* to refine the abstract model on demand, thus producing a sequence of interpolants according to spurious



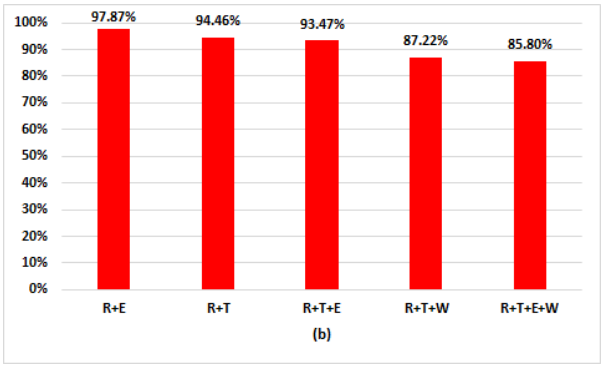


Fig. 9. Experimental results

counterexamples. Vizel and Grumberg [15] then applied the idea of interpolant sequence to SAT-based unbounded model checking. A three-step interpolant computation process was proposed by Cabodi, Loiacono and Vendraminetto [18] to reduce the size of the generated Craig interpolants in SAT-based unbounded model checking. It improves over standard interpolation by reducing memory and time. Chu and Jaffar [23] proposed a framework to synergize partial order reduction with state interpolation, so as to reduce the state explosion problem in the safety verification of concurrent programs. Wachter, Kröning and Ouaknine [24] combined lazy abstraction with interpolants and partial-order reduction: when a spurious counterexample is found, Craig interpolation is used to refine and adjust the precision. The approach by Brillout et al. [19] uses an expressive interpolating calculus that extends to the full theory of quantifier-free Presburger arithmetic with uninterpreted predicates. This setting enables the synthesis of quantified invariants about arrays. The algorithm WHALE introduced by Albarghouthi et al. [20] uses interpolation to compute a function summary by generalizing from an under-approximation of a function. It can verify recursive programs and produce modular safety proofs. Cardinality-constrained extension of Craig interpolation is proposed by von Gleissenthall et al. [17] to synthesize formulas that satisfy a given cardinality constraints based on CEGAR. In [33], it casts the new concept of error invariants for fault localization. An error trace provides sufficient information to repeat the program's behavior that violates the correctness assertion. In order to localize the cause of an error efficiently, it uses error invariants to rule out irrelevant transitions from an error trace and compact the actual cause of an error. Error invariants are also computed by Craig interpolants. The work in [34] extracts interpolants in both forward and backward manner and exploits them for an interwined approximated forward and backward reachability analysis. It applies Craig Interpolants to obtain useful information, that is, computes forward and backward interpolants. In this paper, Error interpolants represent an over-approximation of the pre-image of the bad states. We extract the useful information from spurious counterexample paths by Craig Interpolants.

B. Abstraction-Refinement-Based Verification

Henzinger et al. [25] have successfully applied Craig interpolation to efficiently construct, given an infeasible abstract error trace, a refined abstraction that removes the trace. The approach has been integrated into an explicit-value analysis, which tracks explicit values for a specified set of variables, by Beyer and Löwe [16]. They use Craig interpolation to refine spurious counterexamples in order to construct more precise abstractions of the explicit-value domain. In this paper, we refer to interpolants thus employed in [25], [16] as reachability interpolants. The difference is that we additionally compute safety interpolants and error interpolants from spurious counterexamples, so as to further reduce the state space to be explored when verifying safety properties of programs.

C. Symbolic Execution and Testing

In [21], Jaffar, Murali and Navas applied interpolations to program testing to subsume paths with similar actions. In symbolic execution, when the search fails to reach a goal, an annotation on the CFG of the program, called lazy annotation, is constructed by Craig interpolation. These notations are used to check whether the current state can reach the goal [22].

VII. Conclusion

In this paper, we have introduced new applications of Craig interpolation designed to systematically reduce the program state space to be explored in safety verification. Experiments on a large number of benchmark programs show that the new interpolants and the auxillary optimization strategies are effective in improving scalability of software model checking.

In future work, we plan to develop further optimization techniques, and extend our approach to verify liveness properties of programs. An important problem is the reduction of overheads in the construction of interpolants throughout the verification process.

Acknowledgement

This research is supported by the National Natural Science Foundation of China under grant No. 61420106004 and 61732013. The work was done partially while Duan and Ong were visiting the Institute for Mathematical Sciences, National University of Singapore in 2016. The visit was partially supported by the Institute.

REFERENCES

[1] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4): 21, 2009.

[2] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5): 1512-1542, 1994.

[3] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1-30. Springer, 2011.

[4] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement for symbolic model checking, *J. ACM*, vol. 50, no. 5, pp. 752-794, 2003.

[5] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software Verification with Blast. *Proceedings of the 10th SPIN Workshop on Model Checking Software (SPIN)*, LNCS 2648, Springer-Verlag, pages 235-239, 2003.

[6] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. *In POPL 02: Principles of Programming Languages*, pages 1-3. ACM, 2002.

[7] Cong Tian, Zhenhua Duan, and Zhao Duan. Making CEGAR More Efficient in Software Model Checking. In *IEEE Transactions on Software Engineering (TSE)*, Vol 40(12), 1206-1223, Dec 2014. DOI:10.1109/TSE.2014.2357442, 2014.

[8] Cong Tian and Zhenhua Duan. Detecting Spurious Counterexamples Efficiently in Abstract Model Checking. In *the 35th International Conference on Software Engineering (ICSE 2013)*, 202-211, 2013.

[9] Dirk Beyer and Philipp Wendler. Algorithms for software model checking: Predicate abstraction vs. impact. In *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pages 106-113, IEEE, 2012.

[10] William Craig. Linear reasoning. a new form of the Herbrand-Gentzen theorem. *The Journal of Symbolic Logic*, 22(03): 250-268, 1957.

[11] Kenneth L McMillan. Lazy abstraction with interpolants. In *International Conference on Computer Aided Verification*, pages 123-136, Springer, 2006.

[12] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 25-32, IEEE, 2009.

[13] Dirk Beyer, M Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, pages 189-198, FMCAD Inc, 2010.

[14] Kenneth L McMillan. Interpolation and SAT-based model checking. In *International Conference on Computer Aided Verification*, pages 1-13, Springer, 2003.

[15] Yakir Vizel and Orna Grumberg. Interpolation-sequence based model checking. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 1-8, IEEE, 2009.

[16] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 146-162, Springer, 2013.

[17] Klaus von Gleissenthall, Boris Köpf, and Andrey Rybalchenko. Symbolic polytopes for quantitative interpolation and verification. In *International Conference on Computer Aided Verification*, pages 178-194, Springer, 2015.

[18] Gianpiero Cabodi, Carmelo Loiacono, and Danilo Vendraminetto. Optimization techniques for Craig interpolant compaction in unbounded model checking. *Formal Methods in System Design*, 46(2):135–162, 2015.

[19] Angelo Brillout, Daniel Kroening, Philipp Rümmer, Thomas Wahl. Program verification via Craig interpolation for Presburger arithmetic with arrays. In *VERIFY@ IJCAR*, pages 31-46, 2010.

[20] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 39-55, Springer, 2012.

[21] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A Navas. Boosting concolic testing via interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 48-58, ACM, 2013.

[22] Kenneth L McMillan. Lazy annotation for program testing and verification. In *International Conference on Computer Aided Verification*, pages 104-118, Springer, 2010.

[23] Duc-Hiep Chu and Joxan Jaffar. A framework to synergize partial order reduction with state interpolation. In *Haifa Verification Conference*, pages 171-187, Springer, 2014.

[24] Björn Wachter, Daniel Kroening, and Joel Ouaknine. Verifying multi-threaded software with impact. In *FMCAD*, pages 210–217, 2013.

[25] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. *In Proc. POPL*, 2004, pp. 232-244.

[26] Dirk Beyer. Software verification and verifiable witnesses (Report on SV-COMP 2015). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 401-416, Springer, 2015.

[27] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 887-904, Springer, 2016.

[28] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *In POPL*, pages 58-70, ACM, 2002.

[29] Dirk Beyer, Georg Dresler, and Philipp Wendler. Software verification in the google app-engine cloud. In *International Conference on Computer Aided Verification*, pages 327-333, Springer, 2014.

[30] Bugs found in linux kernel with CPAChekcer: https://cpachecker.sosy-lab.org/achieve.php

[31] Emanuel Kolb, Ondřej Šerỳ, and Roland Weiss. Applicability of the blast model checker: An industrial case study. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 218–229, Springer, 2009.

[32] Dirk Beyer. Competition on software verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 504–524. Springer, 2012.

[33] Evren Ermis, Martin Schäf, and Thomas Wies. Error invariants. In *International Symposium on Formal Methods*, pages 187–201. Springer, 2012.

[34] Yakir Vizel, Orna Grumberg, and Sharon Shoham. Intertwined forward-backward reachability analysis using interpolants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 308–323. Springer, 2013.

[35] David Gries. *The science of programming*. Springer Science & Business Media, 2012.

[36] Arvind Haran, Montgomery Carter, Michael Emmi, Akash Lal, Shaz Qadeer, and Zvonimir Rakamarić. Smack+ corral: A modular verifier. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 451–454. Springer, 2015.

[37] Gérard Basler, Alastair Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl. Satabs: a bit-precise verifier for c programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 552–555. Springer, 2012.

[38] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Refinement selection. In *Model Checking Software*, pages 20–38. Springer, 2015.

[39] Daniel Wonisch. Block abstraction memoization for cpachecker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 531–533. Springer, 2012.