Poster: Accelerating Counterexample Detection in Software Model Checking

Cong Tian, Zhao Duan, and Zhenhua Duan
ICTT and ISN Lab, Xidian University, Xian 710071, P.R. China
{ctian,zhhduan}@mail.xidian.edu.cn,duanzhao@stu.xidian.edu.cn

1 INTRODUCTION AND MOTIVATION

Model checking is an automatic approach in enhancing correctness of systems. However, when it is applied to discover flaws in software systems, most of the respective verification tools lack scalability due to the state-space explosion problem. Abstraction technique is useful in reducing the state space of systems. It maps a concrete set of states to a smaller set of states that is actually an approximation of the system with respect to the property of interest. Predicate abstraction [3] is one of the most often used methods for attaining a finite abstract model from a concrete program which is often even an infinite state system. With predicate abstraction, a finite set of predicates, which determines the precision of the abstraction, is selected to keep track of certain facts about the program variables. The model obtained via predicate abstraction is an over-approximation of the original program. Thus, spurious paths may exist when an insufficient set of predicates are considered.

In order to eliminate spurious counterexamples (false alarms) found in an abstract model, predicate abstraction has been paired with Counterexample-Guided Abstraction Refinement (CEGAR) [1] where a spurious counterexample is automatically analyzed so as to obtain additional predicates to eliminate it. Interpolation [4] is often used to discover new predicates for refining the abstract model. Currently, CEGAR has been popular in most of the software model checkers such as BLAST, SLAM, CPAChecker, and so forth. To further improve the efficiency, lazy abstraction [2] is introduced in CEGAR to reduce the cost of time used for refinement. As a result, different parts of the model may exhibit different degrees of precision. But it is enough for verifying the property.

In CEGAR based counterexample detecting, the challenge is loops in programs since the computation and data related to loops can have statically unknown bounds. This paper devotes to speeding up counterexample detection in CEGAR based software model checking by summarizing *simple loops*. Specifically, for *simple loops*, instead of unwinding it for a large number of times, a novel approach by summarizing is given to analyze the program in a not only fast but also accurate way. With this approach, we always assume that the loop body has been iterated for n, $n \leq 0$, times; and after the $n+1^{th}$ execution of the loop body, the loop condition will not be satisfied. As a result, a linear equation which can be easily solved by an SMT solver is formalized directly.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3194955

2 APPROACH OVERVIEW

Actually, the value of each variable inside a loop varies regularly with respect to the operations on it. In a real world program, most of the loops are *simple* ones where the value of each variable after n, $n \geq 0$, times iterations of the loop body can easily be expressed with its initial value before entering the loop as well as n. Here n is a variable whose value ranges over the set of non-negative integers.

We use program `Loop.c` to present our basic idea about how to handle *simple loops*. We will not enumerate each iteration of the loop to obtain path formulas as usual. Instead of it, we just consider two possibilities: (1) the loop body is executed for 0 times, i.e. the loop condition is unsatisfiable initially; (2) the loop is able to be executed for $n+1$, $n \geq 0$, times, i.e. the loop condition is satisfied until the $(n+2)^{th}$ iteration. As a result, a *simple loop* will be treated like an `if-else` branch structure. The difference is the effect of variables after n times of iterations should be presented in the latter case.

```
1  /* Loop.c */
2  int* a;
3  int i=0;
4  int c=0;
5  while(i<10000){
6        i=i+1;
7        c=c-2;
8        }
9  if(a==NULL) goto Err;
10 c=*a;
11 Err:
```

As shown in Fig. 1, we modify the sub-CFA identified by the left rectangle relative to the *simple loop* in program `Loop.c` by (1)

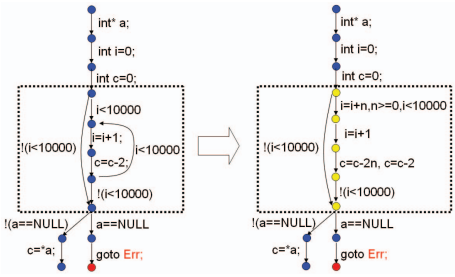

Figure 1: CFA of *Loop.c*

eliminating the flow which points back to execute the loop body again; (2) adding extra statements to express the effect that the loop body has been executed for n times. As shown in the right side rectangle of Fig. 1, there are two branches in the resulting sub-CFA. The one with the condition `!(i<10000)` means that the loop body will not be executed, while the other one indicates that the loop body can be executed for $n+1$, $n \geq 0$, times totally; and at the current time, it has been executed for n times, and is being executed for

the $(n+1)^{th}$ time. To express the situation of variables inside the loop after it has been executed for n times, before the first time a variable occurring in the loop body, a new statement is inserted. For instance, before the first occurrence of variable `i`, i.e. `i<10000`, in the rectangle, a new statement `i=i+n` as well as a new predicate `n` $\geq$ `0` are inserted since `i` is added by 1 in each run of the loop body. Similarly, before the statement `c=c-2`, a new statement `c=c-2*n` is inserted.

Subsequently, with respect to the modified CFA illustrated in the right hand side of Fig. 1, there are two paths that are possible to reach the `Err` label. For the shortest one, its path formula is:

(1) `a1=NULL;` (2) `i1=0;` (3) `c1=0;`
(4) `!(i1<10000)` (5) `a1==NULL`

This path is obviously infeasible since $i1 = 0$ and $!(i1 < 10000)$ conflict with each other. For the other path, the path formula is formalized as:

(1) `a1=NULL;` (2) `i1=0;` (3) `c1=0;`
(4) `i2=i1+n1;` (5) `n1>=0` (6) `i2<10000`
(7) `i3=i2+1;` (8) `c2=c1-2*n1;` (9) `c3=c2-2;`
(10) `!(i3<10000)` (11) `a1==NULL`

Then, if an non-negative integer `n1` exists such that the path formula is satisfiable, the corresponding path is feasible. With an SMT solver, it is obtained that in the case $n = 9999$, the path formula is satisfied. Therefore, the path with the loop iterated for 9999 times is a counterexample that violates the desired property. As a result, instead of invoking Crain interpolants for 9999 times to refine the spurious paths, whether or not a real counterexample is hidden inside the loop is quickly discovered.

Obviously, with the above approach, whether or not a program with loops violates a desired property can be checked out fast and a large number of unwinding of the loop can be avoided.

3 IMPLEMENTATION AND EVALUATION

We have implemented the proposed approach in HyVer to verify C programs. We evaluate HyVer on 18 problem sets (totally 737 programs with about 6.58 million lines) of the benchmark programs[1] for program verification task. HyVer is developed upon CPAChecker by replacing the relative part with the new proposed approach for dealing with loops. To evaluate efficiency of the new proposed technique, we compare HyVer with CPAChecker.

Table 1 illustrates a bird's eye view of the whole experiments. The first column describes the package name of the problem set, the second one shows the amount of the program files, and the third one indicates the total amount of the code in the package. The rest two columns present the percentage of the programs in each package that are successfully verified with CPAChecker and HyVer, respectively. A symbol $\star$ marks that more programs in the package are verified by the relative tool with the timeout threshold being 4 hours. As a summary, HyVer wins the game on 9 of the 18 packages. For other packages, the game ends in a draw.

In addition to the verification success rate, we also evaluate the efficiency. Fig. 2 shows the comparison of CPAChecker and HyVer on the average time (ms) consumed by each package. Note that in this figure, all the programs that are successfully verified by the specific tool are taken into account. That is for the package *list-ext-properties*, the average time consumed by CPAChecker is the average time consumed by 15 programs while by HyVer is the average time consumed by all 20 programs. It can be observed in the figure that HyVer performs well on most of the 18 packages except for package *list-ext-properties*. The reason for that is in this package, CPAChecker runs out of time when verifying 5 of the 20 programs while HyVer successfully verifies all 20 programs but consumes a large amount of time on these 5 programs. Actually, there are totally 335 loops (all in simple) contained in these 5 programs. That exactly illustrates the efficiency of our proposed approach for dealing with *simple loops*.

[1] https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp15/

Table 1: Verification success rate evaluation

Packages	Files	Lines	Percentage (%)	
			CPA.	HyVer
bitvector	36	10193	72.2	72.2
bitvector-loops	2	48	100	100
bitvector-regression	14	296	92.9	92.9
ddv-machzwd	13	80317	100	100
float-benchs	61	3761	70.5	91.8 $\star$
list-properties	13	755	92.3	100 $\star$
systemc	33	24564	63.6	100 $\star$
float-cbmc-regression	32	1056	87.5	87.5
float-cdfpl	40	1464	100	100
list-ext-properties	20	1408	75	100 $\star$
loops	65	5026	84.6	84.6
loop-lit	16	290	81.2	87.5 $\star$
ntdirvers	20	100022	70	70
ntdirver-simplified	10	18105	90	90
product-lines	311	409763	94.5	100 $\star$
loop-acceleration	35	766	20	91.4 $\star$
loop-new	8	75	50	100 $\star$
loop-invgen	18	589	38.9	94 $\star$

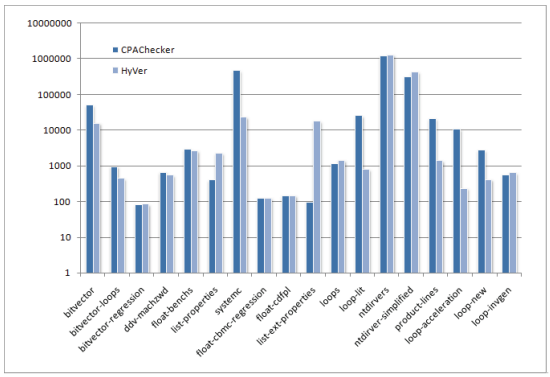

Figure 2: Efficiency evaluation

ACKNOWLEDGEMENT

This research is supported by the NSFC under grant No. 61420106004, 61751207 and 61732013. Cong Tian and Zhenhua Duan are the corresponding authors.

REFERENCES

[1] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In Proc. CAV 2011, pages 184-190, Springer, 2011.
[2] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. Proc. Symp. Principles of Programming Languages, pp. 58-70, 2002.
[3] Graf. S, Saidi. H, Construction of abstract state graphs with PVS. In: Computer aided verification (CAV 1997), LNCS 1254. Springer, Berlin, pp. 72-83, 1997.
[4] Henzinger TA, Jhala R, Majumdar R, McMillan KL. Abstractions from proofs. In: Principles of programming languages. ACM Press, New York, pp 232-244, 2004.