

Stack Abstraction for Interprocedural Software Verification

Márk Somorjai 

Critical Systems Research Group
Budapest University of Technology and Economics
Budapest, Hungary
somorjai.mark@edu.bme.hu

Mihály Dobos-Kovács 

Critical Systems Research Group
Budapest University of Technology and Economics
Budapest, Hungary
mihaly.doboskovacs@edu.bme.hu

Abstract—One of the biggest challenges of software verification is the large state space of programs that grows exponentially with the number of program variables. In programs with procedures, the growth is further escalated by the addition of the location stack, which can stretch infinitely deep in recursive programs.

A widely used technique for combating the state space explosion caused by the data state of the program is to apply abstraction to the values of variables. In my work, I propose the extension of abstraction to location stacks, in order to reduce the size of the abstract state space and ultimately make interprocedural verification more efficient.

I. INTRODUCTION

As software enters more and more parts of our lives, the need for software verification in safety-critical systems increases. However, this is not an easy task due to the state-explosion problem: the size of the state space of a program grows exponentially with the number of variables [1]. Counterexample-Guided Abstraction Refinement (CEGAR) [2] is an iterative model checking algorithm that reduces the state space of the program by employing abstraction.

Procedures allow modularity, structure, as well as the reuse of already written software. On the other hand, they disrupt the sequential flow of program execution, and obfuscate variable tracking with their template-like behaviour. On top of that, they extend the state of the program with the call stack, which can stretch infinitely deep in recursive programs and as a result, lead to an orthogonal state space explosion.

In my research, I propose the extension of abstraction to the call stack. With parts of the call stack abstracted away, similarities between states with different stacks can be detected and the size of the abstract state-space can further be reduced. This improves the efficiency of interprocedural software verification, especially in the case of recursive programs, where the call stack has a greater effect on the size of the state space. The proposed technique was implemented and evaluated in the CEGAR-based verification tool THETA [3].

II. BACKGROUND

In order to verify software, a formal representation is often used to model a program and its states. A *state* of a program can be described by a program location and its data state, i.e., the values of all program variables. All possible states of the program make up its *state space*, which grows

exponentially with the number of variables in the program. To make verification feasible, reduction techniques are employed, such as abstraction.

A. Abstraction

An *abstract state* represents a set of concrete states of a program. It overapproximates the concrete data states by omitting some variables as in *explicit abstraction*, or by tracking predicates (e.g. $x > 0$) instead of the variable values, in the case of *predicate abstraction*. Using abstraction, the state space of a program is reduced into the abstract state space, which can be represented by an *Abstract Reachability Graph (ARG)*: a directed acyclic graph, where the nodes represent abstract states and the directed edges represent transitions between them. If an abstract state is encountered during exploration, the data state of which is overapproximated by a previously visited abstract state's data state in the same location, then it is *covered* and need not be expanded further, since all outgoing paths from it are already present on the overapproximating state.

B. CEGAR

Counterexample-Guided Abstraction Refinement (CEGAR) is an abstraction-based model checking algorithm. The core of the algorithm can be seen on Figure 1.

The abstractor explores the abstract state space. When an abstract error state is reached, an abstract counterexample is provided to the refiner. If the counterexample is feasible in the state space of the program, the algorithm terminates with a concrete counterexample to the program's safety. Otherwise

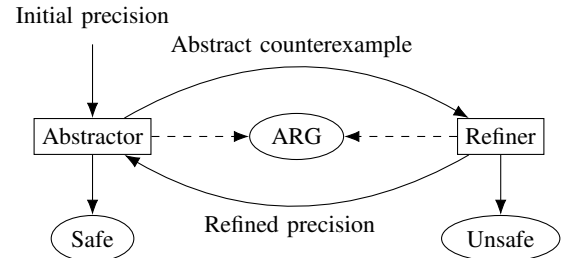


Fig. 1. The CEGAR loop

the abstraction is refined and exploration continues. Once all abstract states have been visited, the algorithm terminates and the program is deemed safe.

III. INTERPROCEDURAL VERIFICATION

Procedures come with procedure calls, which makes verification more difficult. While in some cases procedures can be *inlined*, the verification of certain (e.g. recursive) programs requires a more nuanced interprocedural analysis.

A. Procedures

When verifying a program with procedures, the program's state is extended by a location stack that keeps track of locations of procedure calls. Consequently, if two states of the program are in the same location with the same variable values, it no longer means that the states are identical, since they could have different location stacks underneath. The location stack can stretch infinitely deep in recursive programs, resulting in further expansion to the already huge state space, that is orthogonal to the explosion caused by the data state.

The most straightforward solution to this problem is to inline the procedures. However, it can result in a huge verification input and cannot handle recursive programs. Other approaches, such as summaries [4] try to capture the effects of a procedure to the program's state, but these require complex algorithms to derive the logic formula. The process can be simplified using contracts [5] on the parameters of a procedure, however, these have to be manually specified by the developers. I propose an approach that can make interprocedural verification more efficient with an adjustment to the CEGAR algorithm.

B. Stack Abstraction

In intraprocedural verification, the covering relation between abstract states ensures that substantial parts of the abstract state space do not need to be explored. A sound way of adapting this relation to interprocedural analysis would be to require one abstract state's data state to overapproximate the other's while both abstract states have the exact same location stack, since this definition guarantees that all outgoing paths from the covered abstract state are present on the covering one. However, it misses out on some of the power of abstraction: not considering similarities between the data state of abstract states because they have different stacks, leads to redundant calculations. Consider situations when two abstract states are in the same location and one's data state overapproximates the other's, while their stacks are not necessarily the same. In such a situation, it is not true that the overapproximated state's outgoing paths are all present on the covering abstract state, since after the current procedure ends, the paths might continue from different calling locations. However, up to that return point they will overapproximate one another.

In my work, I propose an approach that eliminates such redundant calculations. The main idea is to abstract the bottoms of the stacks away, so that the two abstract states overapproximate one another because without the bottom of

the stack, all paths would end at the final location of the current procedure. Therefore, the exploration can continue from the calling location of the covered state and the calculation of outgoing paths up to that point is omitted. To achieve this, covering only needs to require that the top location of the stacks being compared match, instead of the whole stack. In such case, the top location of the covered state is popped and exploration continues from the calling location, eliding the redundant calculations.

The adjustments described above happen during the exploration of the abstract state space, meaning they can be integrated into the abstractor of the CEGAR algorithm. Additionally, changes need to be made to the refiner as well in order to detect new kinds of spurious counterexamples that arise from popping the top location at some other point than the final location of a procedure. If such a transition is found, the popped abstract state can be marked in the precision so that it is not popped in the next iteration.

IV. PRELIMINARY RESULTS

The reduction technique was implemented in THETA and evaluated against the tool's baseline version (NOPOP) with different configurations of THETA. A set of 1219 C programs were verified from the SV-COMP benchmark repository¹. The number of solved tasks per configuration within a 15 minute timeout can be seen on Table I.

TABLE I
NUMBER OF SOLVED TASKS BY CONFIGURATION.

abstraction	version	control	eca	loops	recursive
EXPLICIT	NOPOP	9	197	70	38
EXPLICIT	POP	9	177	60	36
PREDICATE	NOPOP	7	262	294	23
PREDICATE	POP	7	263	157	38

None of the configurations gave a wrong answer on any of the 1219 tasks. The technique showed some computational overhead in the three non-procedural groups of tasks (*control-flow*, *eca*, *loops*). For recursive programs, the technique improved the performance of predicate abstraction by over 65%. Moreover, 20% of those tasks were ones that no previous configuration of THETA was able solve. Since recursion can be detected before verification, the technique can be applied for recursive tasks only, which is how the tool was used at SV-COMP '24 [6].

V. CONCLUSION

The proposed technique has proven to increase the efficiency of verification with predicate abstraction. In my ongoing work, I examine if a subset of the abstract data state could be used for covering, to make the technique applicable in more situations. Another approach I am looking into is reusing some information from the covering abstract state's data state after the procedure. The additional information could further eliminate some paths from the covered abstract state, when exploration continues after popping.

¹<https://github.com/sosy-lab/sv-benchmarks/tree/master/c>

REFERENCES

- [1] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, “Model checking and the state explosion problem,” in *LASER Summer School on Software Engineering*. Springer, 2011, pp. 1–30.
- [2] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement for symbolic model checking,” *J. ACM*, vol. 50, no. 5, p. 752–794, sep 2003.
- [3] T. Tóth, A. Hajdu, A. Vörös *et al.*, “Theta: a Framework for Abstraction Refinement-Based Model Checking,” in *17th Conference on Formal Methods in Computer-Aided Design*, D. Stewart and G. Weissenbacher, Eds., 2017, pp. 176–179.
- [4] G. Yorsh, E. Yahav, and S. Chandra, “Generating precise and concise procedure summaries,” *SIGPLAN Not.*, vol. 43, no. 1, p. 221–234, jan 2008.
- [5] B. Meyer, “Applying ‘design by contract’,” *Computer*, vol. 25, no. 10, pp. 40–51, 1992.
- [6] L. Bajczi, C. Telbisz, M. Somorjai, Z. Ádám, M. Dobos-Kovács, D. Szekeres, M. Mondok, and V. Molnár, “Theta: Abstraction based techniques for verifying concurrency (competition contribution),” in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds. Cham: Springer Nature Switzerland, 2024, pp. 412–417.