Collaborative Verification of Uninterpreted Programs

Yide Du[1], Weijiang Hong[1,2], Zhenbang Chen[1](✉), and Ji Wang[1,2]

[1] College of Computer, National University of Defense Technology, Changsha, China
[2] State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
{dyd1024,hongweijiang17,zbchen,wj}@nudt.edu.cn

Abstract. Given a set of uninterpreted programs to be verified, the trace abstraction-based verification method can be used to solve them once at a time. The verification of different programs is independent of each other. However, the individual verification for each one is a waste of resources if the programs behave similarly. In this work, we propose a framework for the collaborative verification of a set of uninterpreted programs, which accumulates and reuses the abstract models of infeasible traces to improve the verification's efficiency. We have implemented the collaborative verification framework and the preliminary result demonstrate that our collaborative method is effective on the benchmark.

Keywords: Collaborative verification · Uninterpreted programs · CEGAR.

1 Introduction

An uninterpreted program [11] is a program that works with arbitrary data models and all of its functions have only a signature information and satisfy the common property, *i.e.*, same inputs produce same outputs. Given a program $\mathcal{P}$ to be verified, we can over-approximate $\mathcal{P}$ by an uninterpreted version $\mathcal{P}_u$ and do the verification. In most cases, the verification of $\mathcal{P}_u$ has a lower complexity. However, even for uninterpreted programs, the verification problem is in general *undecidable* [11]. Recently, a decidable class of uninterpreted programs called *coherent* ones has been discovered [11]. Based on this result, a more effective trace abstraction-based CEGAR-style [7] verification method for general uninterpreted programs is proposed [8].

We notice that all this work are focus on the verification problem of a single program, but there are many scenarios that a set of programs need to be verified. For example, the incremental verification plays an important role in the area of regression verification as it can significantly improve the verification efficiency. During software development, only part of the software be changed and due to the high complexity of program verification, there is no need to verify the changed version from scratch. There are similar behaviors between different programs. Exploiting the similarity between softwares and reusing the verification results is an effective way to improve the efficiency of verification.

In this work, we propose a collaborative verification method to reuse the abstract models of infeasible traces during the CEGAR-style verification across different programs to improve the efficiency of verification.

The main contributions of this paper are as follows:

– We propose a framework for collaborative verification that can reuse the abstract models of infeasible traces to improve the efficiency of verification.

– We have implemented our framework in a prototype for uninterpreted programs and the preliminary result demonstrate that our collaborative method is effective on the benchmark.

Structure. The remainder of this paper is organized as follows. Section 2 gives a motivation example. The collaborative verification framework will be given in Section 3. Section 4 gives the preliminary evaluation results. Finally, Section 5 compares the related work, Section 6 introduces the next steps, and Section 7 concludes the paper.

2 Motivation

```
x := y;

if (z != n1){
  x := g(x);
  y := g(y);
} else {
  x := f(x);
  y := f(y);
}

assert(x = y);
```

(a) $\mathcal{P}_0$

```
x := y;

if (z != n1){
  x := h(x);
  y := h(y);
} else {
  x := f(x);
  y := f(y);
}

assert(x = y);
```

(b) $\mathcal{P}_1$

Fig. 1. The motivation example.

In Figure 1 there are two uninterpreted programs in which $\mathcal{P}_1$ is obtained by modifying the program $\mathcal{P}_0$. Notice that all the traces of these two programs satisfy the equality of x and y at beginning, then apply the same functions on both x and y, so all of them satisfy the assertion in the end. If we use the CEGAR method with the congruence-based abstraction in [8] to verify them, we need 2 iterations, respectively.

We observe that $\mathcal{P}_1$ is the modified version of $\mathcal{P}_0$ and the false branch of them are the same. Therefore, we can conclude that the false branch of $\mathcal{P}_1$ is correct when $\mathcal{P}_0$ is verified to be correct, only the true branch of $\mathcal{P}_1$ need to be verified. In the scenario of software development and evolution, most of the traces are the same between two successive versions, it's no need to verify them from scratch. Based on these observations, we propose a collaborative verification method that reuses the abstract model of infeasible traces to improve the verification's efficiency. Next, we demonstrate the process of our collaborative verification method on this motivation example.

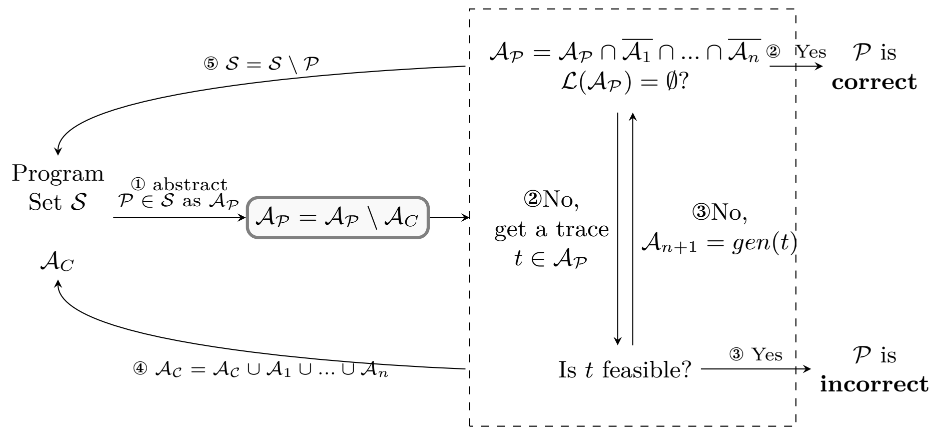
We use $\mathcal{A}_C$ to represent the accumulated model of infeasible traces, and $\mathcal{L}(\mathcal{A}_C)$ is empty at the beginning. First, $\mathcal{P}_0$ is verified to be correct by the CEGAR method, and the abstract models of the infeasible traces in $\mathcal{P}_0$ are merged with $\mathcal{A}_C$. When $\mathcal{P}_1$ is to be verified, its false branch can be removed by performing $\mathcal{A}_{\mathcal{P}_1} = \mathcal{A}_{\mathcal{P}_1} \setminus \mathcal{A}_C$ and the true branch can be verified by the CEGAR method. It takes only 2 and 1 iterations of refinement to successfully verify $\mathcal{P}_0$ and $\mathcal{P}_1$ respectively. Intuitively, the closer the programs to be verified, the more effective our collaborative verification method is.

3 Collaborative Verification Framework

We propose a collaborative verification framework based on the scheme of CEGAR for trace abstraction, which accumulates the abstract models of infeasible traces during the verification procedure, and the accumulated abstract models are later reused to facilitate other program's verification. The details can be found in Figure 2.

Our collaborative verification framework introduces an initial empty automata $\mathcal{A}_C$ to accumulates the abstract models during the verification progress. When verifying a set of programs $\mathcal{S}$, we pick one program $\mathcal{P}$ from them and abstract it to an FSA $\mathcal{A}_{\mathcal{P}}$ which include all the traces of $\mathcal{P}$. Then we wipe off those infeasible traces included in $\mathcal{A}_C$ by $\mathcal{A}_{\mathcal{P}} = \mathcal{A}_{\mathcal{P}} \setminus \mathcal{A}_C$. During the verification, we can conclude that $\mathcal{P}$ is correct if $\mathcal{L}(\mathcal{A}_{\mathcal{P}}) = \emptyset$ holds, otherwise, a trace t can be extracted from $\mathcal{L}(\mathcal{A}_{\mathcal{P}})$. If t is feasible, a real counter-example is found, and we can conclude that $\mathcal{P}$ is incorrect. Otherwise, we can abstract an FSA $\mathcal{A}_{n+1}$ from t which accept all the infeasible traces that with the same reason with t. Then, $\mathcal{A}_{\mathcal{P}}$ can be refined by $\mathcal{A}_{n+1}$. The CEGAR process continues until a feasible counter-example is found or $\mathcal{P}$ is concluded to be correct.

After the verification of each program is completed, the abstract models obtained during the verification process can be merged with $\mathcal{A}_C$ for reusing. When a new program is to be verified, $\mathcal{A}_C$ can be used to refine the program abstraction. The infeasible traces included in $\mathcal{A}_C$ can be removed directly without the CEGAR



⑤ $\mathcal{S} = \mathcal{S} \setminus \mathcal{P}$
$\mathcal{A}_\mathcal{P} = \mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap ... \cap \overline{\mathcal{A}_n}$
$\mathcal{L}(\mathcal{A}_\mathcal{P}) = \emptyset$?
② Yes
$\mathcal{P}$ is correct
Program Set $\mathcal{S}$
① abstract $\mathcal{P} \in \mathcal{S}$ as $\mathcal{A}_\mathcal{P}$
$\mathcal{A}_\mathcal{P} = \mathcal{A}_\mathcal{P} \setminus \mathcal{A}_C$
②No, get a trace $t \in \mathcal{A}_\mathcal{P}$
③No, $\mathcal{A}_{n+1} = gen(t)$
$\mathcal{A}_C$
④ $\mathcal{A}_C = \mathcal{A}_C \cup \mathcal{A}_1 \cup ... \cup \mathcal{A}_n$
Is t feasible?
③ Yes
$\mathcal{P}$ is incorrect

Fig. 2. Collaborative verification framework.

process. Thus, our framework shares the abstract models of infeasible traces within the programs in $\mathcal{S}$ to improve the verification's efficiency.

4 Preliminary Result

We have implemented a prototype for our collaborative verification framework in OCaml. We prepare to evaluate our collaborative verification framework on a set of similar programs. In the scenario of regression verification, during the software development and evolution, only part of the software be changed between two successive software versions and most of the program behavior is similar. Based on this scenario, we evaluate our collaborative method to answer the following question.

Efficiency, *i.e.*, how efficient is our collaborative verification method compared to verifying each program individually when there are a set of uninterpreted programs to be verified?

Considering that there is no standard benchmark for similar programs, we convert some real-world programs from SV-COMP[15] to uninterpreted programs. These programs can be used as original programs in regression verification. To simulate the software evolution process, we randomly extract 10 correct programs as the initial programs and mutate them. Therefore we have 10 groups of similar programs that simulate software evolution.

All the experiments are carried out on a server with eight cores and 32G memory. The operating system is Ubuntu 18.04. We use the average value of three runs to eliminate the experimental errors.

The collaborative verification method is effective if the abstract models we collected reduces the number of refinements and further reduces the time cost of verification. We evaluate the collaborative verification method on the benchmark that simulates software evolution. Our preliminary results demonstrate that our collaborative verification method is effective, for the average speedups of 2.70x (1.11x $\sim$ 3.84x) on the benchmark.

5 Related Work

There are many existing works for collaborative verification. In some works [4][5][3], different verification tools are collaborated in different ways to improve verification efficiency and the ability to detect assertions. These methods have achieved good results, but all of them focus on a single program, while our work consider a set of similar programs. Such as the scenario of regression verification, the incremental verification is an efficient way to improve the verification's efficiency. There are different approachs to implement collaborative verification in different verification tools. For example, the function summaries [14], the abstraction precisions[2], the procedure summaries the state-space modeled by automata [1,10], and the loop summaries [6], the assertions in predicate analysis [16] and the counter-example traces [3] are reused to improves the efficiency of regression verification. Our approach reuses the abstract models of infeasible traces to improve the efficiency of verification.

Uninterpreted programs and their verification problems have been studied in many works. A decidable class of uninterpreted programs was found by Mathur *et al.* [11], based on this result, many decidable results [13,9,12] are proposed for different types of programs. For the verification problem of general uninterpreted program, CEGAR-based verification provide a general framework. A congruence-based trace abstraction method for infeasible traces was proposed by Hong *et al.*[8] and is more efficient than the interpolant-based trace abstraction method [7]. In this work, we implemented our collaborative verification framework based on Hong *et al.*[8]'s work.

6 Next Steps

The abstract models of infeasible traces are critical for the verification's efficiency, the better the generalization of the trace abstract models are, the less number of the program's refinement need. We studied the existing trace abstraction method [8] and found that the method does not distinguish the different reasons why a trace is infeasible. So we intend to propose a fine-grained generalization method to improve the generalization's ability.

Except for the scenario of regression validation, we are intend to consider more scenarios in which our approach is applicable, such as the component-based software development. In this scenario, programs are obtained by composing several designed components, and their behavior are similar. In the later evaluation, we are intend to explore the effect of different factors on the efficiency of our collaborative verification method, such as the verification order of programs and the different proportion of correct programs.

Furthermore, we plan to extend our collaborative verification framework to more types of programs and different verification tools.

7 Conclusion

This paper propose a collaborative verification framework for a set of uninterpreted programs. In some scenarios such as software development, there are similar traces between this set of uninterpreted programs. So we preserve the abstract models of infeasible traces during the verification process, when a new program is to be verified, the saved abstract models can be reused to do a refinement on it, thereby speeding up the overall verification speed. We have implemented our method and the preliminary results demonstrate that our method performs better on the benchmark.

Acknowledgments. This research was supported by the NSFC Programs (No. 62172429 and 62032024).

References

1. Beyer, D., Holzer, A., Tautschnig, M., Veith, H.: Information reuse for multi-goal reachability analyses. In: European Symposium on Programming. pp. 472–491. Springer (2013)
2. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 389–399 (2013)
3. Beyer, D., Wendler, P.: Reuse of verification results. In: International SPIN Workshop on Model Checking of Software. pp. 1–17. Springer (2013)
4. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: International Symposium on Formal Methods. pp. 132–146. Springer (2012)
5. Csallner, C., Smaragdakis, Y.: Check'n'crash: Combining static checking and testing. In: Proceedings of the 27th international conference on Software engineering. pp. 422–431 (2005)
6. He, F., Yu, Q., Cai, L.: Efficient summary reuse for software regression verification. IEEE Transactions on Software Engineering (2020)
7. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: International Static Analysis Symposium. pp. 69–85. Springer (2009)
8. Hong, W., Chen, Z., Du, Y., Wang, J.: Trace abstraction-based verification for uninterpreted programs (accepted). In: Formal Methods, FM 2021, Beijing, China, November 20-26, 2021, Proceedings (2021)
9. La Torre, S., Parthasarathy, M.: Reachability in concurrent uninterpreted programs. In: 39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)

10. Lauterburg, S., Sobeih, A., Marinov, D., Viswanathan, M.: Incremental state-space exploration for programs with dynamically allocated data. In: 2008 ACM/IEEE 30th International Conference on Software Engineering. pp. 291–300. IEEE (2008)
11. Mathur, U., Madhusudan, P., Viswanathan, M.: Decidable verification of uninterpreted programs. Proceedings of the ACM on Programming Languages **3**(POPL), 1–29 (2019)
12. Mathur, U., Madhusudan, P., Viswanathan, M.: What's decidable about program verification modulo axioms? In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 158–177. Springer (2020)
13. Mathur, U., Murali, A., Krogmeier, P., Madhusudan, P., Viswanathan, M.: Deciding memory safety for single-pass heap-manipulating programs. Proceedings of the ACM on Programming Languages **4**(POPL), 1–29 (2019)
14. Sery, O., Fedyukovich, G., Sharygina, N.: Incremental upgrade checking by means of interpolation-based function summaries. In: 2012 Formal Methods in Computer-Aided Design (FMCAD). pp. 114–121. IEEE (2012)
15. SV-benchmarks: https://github.com/sosy-lab/sv-benchmarks
16. Yu, Q., He, F., Wang, B.Y.: Incremental predicate analysis for regression verification. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–25 (2020)