YOGAR-CBMC: CBMC with Scheduling Constraint Based Abstraction Refinement (Competition Contribution)

Liangze Yin[1], Wei Dong[1(✉)], Wanwei Liu[1], Yunchou Li[2], and Ji Wang[1]

[1] National University of Defense Technology, Changsha, China
yinliangze@163.com, wdong@nudt.edu.cn
[2] Beijing Institution of Tracking and Telecommunication Technology, Beijing, China

Abstract. This paper presents the YOGAR-CBMC tool for verification of multi-threaded C programs. It employs a scheduling constraint based abstraction refinement method for bounded model checking of concurrent programs. To obtain effective refinement constraints, we have proposed the notion of *Event Order Graph (EOG)*, and have devised two graph-based algorithms over EOG for counterexample validation and refinement generation. The experiments in SV-COMP 2017 show the promising results of our tool.

1 Verification Approach and Software Architecture

Bounded model checking (BMC) is among the most efficient techniques for concurrent program verification [1]. However, due to non-deterministic interleavings, a huge encoding is required for an exact description of the thread interaction.

YOGAR-CBMC is a verification tool for multi-threaded C programs based on shared variables under *sequential consistency (SC)*. For these programs, we have observed that the *scheduling constraint*, which defines that "for any pair $\langle w, r\rangle$ s.t. r reads the value of a variable v written by w, there should be no other write of v between them", significantly contributes to the complexity of the behavior encoding. In the existing work of BMC, the scheduling constraint is encoded into a complicated logic formula, the size of which is cubic in the number of shared memory accesses [2].

To avoid the huge encoding of scheduling constraint, YOGAR-CBMC performs abstraction refinement by weakening and strengthening the scheduling constraint [3]. Figure 1 demonstrates the high-level overview of its architecture. We initially ignore the scheduling constraint and then obtain an over-approximation abstraction φ_0 of the original program (w.r.t. the given loop

This work was supported by the National key R&D program of China (No. 2017YFB1001802); the 973 National Program on Key Basic Research Project of China (No. 2014CB340703); and the National Nature Science Foundation of China (No. 61690203, No. 61532007).

D. Beyer and M. Huisman (Eds.): TACAS 2018, LNCS 10806, pp. 422–426, 2018.
https://doi.org/10.1007/978-3-319-89963-3_25

unwinding depth). If the property is safe on the abstraction, then it also holds on the original bounded program. Otherwise, an abstraction counterexample is obtained and the abstraction will be refined if the counterexample is infeasible.

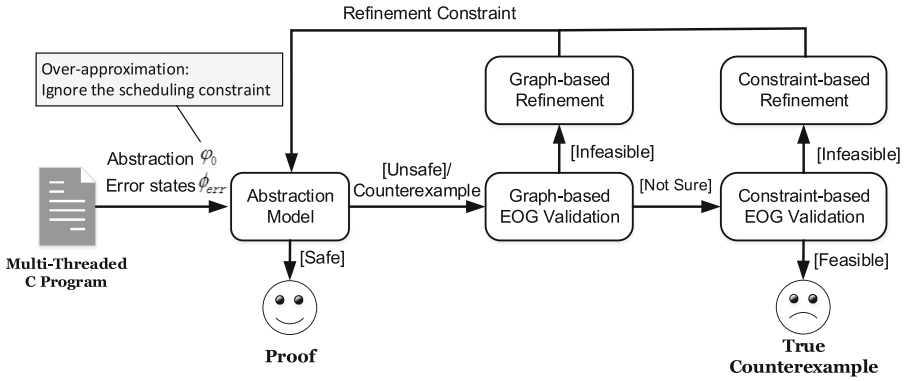

Fig. 1. High-level overview of YOGAR-CBMC architecture.

The performance of this method significantly depends on the generated refinement constraints. Ideally, a refinement constraint should have a small size yet a large amount of space should be reduced during each iteration. To achieve this goal, we have proposed the notion of *Event Order Graph (EOG)*, and have devised two graph-based algorithms over EOG for counterexample validation and refinement generation. Given an abstraction counterexample π, the corresponding EOG G_π captures all the event order requirements of π defined in the scheduling constraint. The counterexample π is feasible iff the EOG G_π is feasible. To validate the feasibility of G_π, we have proposed several deduction rules to deduce those implicit order requirements of G_π. If any cycle exists in G_π, then both π and G_π are infeasible. A graph-based refinement algorithm is then employed to analyze all the possible "kernel reasons" of all cycles. By eliminating those "redundant" kernel reasons, we can usually obtain a small set of "core kernel reasons", which can usually be encoded into a small refinement constraint. The experimental results show that: (1) Our graph-based EOG validation method is powerful enough in practice. Given an infeasible EOG, it can usually identify the infeasibility with rare exceptions. (2) Our graph-based refinement method is effective. If some cycle exists in G_π, it can usually obtain a small refinement constraint which reduces a large amount of search space.

If no cycle exists in G_π, we are not sure whether the EOG is feasible or not. We employ a constraint-based EOG validation process to further validate its feasibility by constraint solving. If an infeasibility is determined, a constraint-based refinement generation process is performed to refine the abstraction, which obtains only one kernel reason of the infeasibility. Enhanced by these two constraint-based processes, we have proved that our method is sound and complete w.r.t the given loop unwinding depth.

Consider the example shown in Fig. 2. We attempt to verify that it is impossible for both m and n to be 1 after the exit of threads `thr1` and `thr2`, which has a modular proof in this program. In this example, we have observed that:

(1) Excluding the 3049 CNF clauses encoding the `pthread_create` and `pthread_join` functions, the encodings of this program with and without the scheduling constraint have 10214 and 1018 CNF clauses, respectively. It indicates that the scheduling constraint significantly contributes the complexity of the program encoding.
(2) During the verification, all the abstraction counterexamples are infeasible. All of them have been identified to be infeasible by our graph-based EOG validation method. It indicates that our graph-based EOG validation method is powerful enough in practice.
(3) The property is verified through only three refinements, and only 7 simple CNF clauses are added during the refinement processes. It indicates that the refinement constraints usually have small sizes yet reduce large amount of the search space, and our graph-based refinement method is effective.

```
int x = 1, y = 1, m = 0, n = 0;
void* thr1(void * arg) {
  x = y + 1;
  m = y;
  x = 0;
}
void* thr2(void * arg) {
  y = x + 1;
  n = x;
  y = 0;
}
void main() {
  pthread_t  t1, t2;
  pthread_create(&t1, 0, thr1, 0);
  pthread_create(&t2, 0, thr2, 0);
  pthread_join(t1, 0);
  pthread_join(t2, 0);
  assert (!(m == 1 && n == 1));
}
```

Fig. 2. An illustration example.

2 Strengths and Weaknesses

The strengths of our tool include: (1) Our approach is a general purpose technique for multi-threaded C program verification, not assuming any special characteristics of the programs. Our tool supports nearly all features of C and PThreads. (2) Our approach is efficient in practice. Without the scheduling constraint, the size of the encoding can be dramatically reduced. Moreover, it can usually verify the property with a small number of refinements, while the refinement constraints usually have small sizes. (3) Enhanced by the constraint-based counterexample validation and refinement generation processes, our approach is sound and complete w.r.t. the given loop unwinding depth. It provides both proofs and refutations for the property. If the property is found to be false, a counterexample will be provided. (4) As the abstractions usually have small sizes, our tool generally consumes less memory than those tools giving an exact description of the scheduling constraint. In this sense, our tool is more scalable.

We have applied Yogar-CBMC to the benchmarks in the concurrency track of SV-COMP 2017. Our tool has successfully verified all these examples within

1550 s and 43 GB of memory. It has won the gold medal in the Concurrency Safety category of SV-COMP 2017 [4].

However, for those programs where the scheduling constraint is not the major part of the encoding, our method may still need dozens of refinements. Given that the abstractions may have similar size with the monolithic encoding, our tool may run worse than those monolithic encoding tools. Moreover, for those real-world programs with a large number of read/write accesses and complex data structures, how to reduce the number of refinements and how to deal with the shared structure members more efficiently, are still challenging problems.

3 Tool Setup and Configuration

The binary file of YOGAR-CBMC for Ubuntu 16.04 (x86_64-linux) is available at https://gitlab.com/sosy-lab/sv-comp/archives. It is implemented on top of CBMC-4.9[1]. Its setup and configuration are same as that of CBMC. The tool-info module and benchmark definition of our tool is "yogar-cbmc.py" and "yogar-cbmc.xml" respectively.

Our tool needs two parameters of CBMC: `--no-unwinding-assertions` and `--32`. The unwind bound of YOGAR-CBMC is dynamically determined through a syntax analysis. Particularly, the bound is set to 2 for programs with arrays, and n if some of the program's `for` loops are upper bounded by a constant n, which is the same as for MU-CSEQ [5]. To run YOGAR-CBMC for a program ⟨`file`⟩, just use the following command:

```
./yogar-cbmc --no-unwinding-assertions --32 ⟨file⟩
```

Participation/Opt Out. YOGAR-CBMC competes only in the *concurrency category*.

4 Software Project and Contributors

YOGAR-CBMC is developed at HPCL, School of Computers, National University of Defense Technology, and includes contributions by the authors of this paper. Its source code is available at https://github.com/yinliangze/yogar-cbmc. For more information, contact Liangze Yin.

References

1. Beyer, D.: Software verification with validation of results - (Report on SV-COMP 2017). In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 331–349. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_20
2. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_9

[1] Download from https://github.com/diffblue/cbmc/releases on Nov 20, 2015.

3. Yin, L., Dong, W., Liu, W., Wang, J.: Scheduling constraint based abstraction refinement for multi-threaded program verification (2017). https://arxiv.org/abs/1708.08323
4. SV-COMP: 2017 software verification competition (2017). http://sv-comp.sosy-lab.org/2017/
5. Tomasco, E., Nguyen, T.L., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: MU-CSeq 0.4: individual memory location unwindings (competition contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 938–941. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_65