Expediting Verification of Assertions in Loops by Isolation

MURAD AKHUNDOV, FEDERICO MORA, and MARSHA CHECHIK, University of Toronto

1 INTRODUCTION

Program verification engines, or verifiers, are tools that take in a program with assertions, and attempt to prove that the assertions always hold. In this paper, we will focus on Bounded Model Checking, a popular verification technique. Bounded Model Checkers (BMCs) [3], often struggle to check assertions in unbounded loops, loops with high bounds, or nested loops. This is because, for unbounded loops, no number of loop unrollings is sufficient; with high loop bounds, a large number of loop unrollings is required; and with nested loops, even a small number of loop unrolling results in a large SAT encoding. Consider the example in Listing 1. A BMC needs to unroll the loops $O(\texttt{SIZE}^2)$ times to prove the assertion, when `SIZE = 200`, a state-of-the-art BMC, CBMC [4] takes over 10 minutes. In this paper, we propose a novel technique that proves the same assertion in under 10 seconds, and improves the performance of CBMC on many other similar cases while incurring a manageable overhead cost.

Listing 1. Nested Loop

```
1 void two_sum_prev(int *arr, int s){ //check if two elements in arr add to
      s
2    for (int i = 0; i < SIZE; i++){ //SIZE is some fixed number
3        for (int j = 0; j < i; j++){
4            assert(j < SIZE);
5            if(arr[j] + arr[i] == s) printf("found!"); }}}
```

The key observation that our technique exploits is that inner segments of the program-under-test are often sufficient to prove that an assertion always holds. As seen in Listing 1, the information needed to prove the assertion is entirely contained in the body of the outer loop and its condition. Specifically, to prove that `j < SIZE`, it is sufficient to know that `j < i` and `i < SIZE`, and that the loop will terminate (i.e., `j` is eventually `i`). We propose a technique to find these segments and expedite verification. Our tool, Qicc finds and isolates these segments so that they can be verified individually, quickly, and concurrently.

2 BACKGROUND

This section introduces *control-flow graphs (CFGs)*, which we use to model programs, and *extracted loops*, which are the isolated components of control-flow-graphs that Qicc verifies individually.

A *CFG* is a directed graph representing flow-of-control between statements in programs [1]. We say that a sub-graph G' of a graph G has a *single entry* (respectively, exit) if there is only one edge coming into (respectively, leaving) G' from (respectively, to) the rest of graph G. G' is *local* if it has a single entry and exit. We say G' is a *loop* if every node in G' can be reached from every other node in G', and if G' contains at least two nodes. G' is a *local loop* if it is local and a loop, and if its single entry is its single exit. We call the entry/exit node the *root* of the local loop. We call the sub-graph of a local loop without its root the *body* of the local loop. A local loop is *extracted* when all nodes in its body are moved to a new function.

3 OUR APPROACH

Qicc speeds up verification of nested assertions by extracting local loops and calling the verification tool on the new functions. Qicc uses CBMC [4] but the same approach can be used with other tools.

Qicc takes three steps: it finds local loops, extracts them, and checks them individually. For each individual check, Qicc executes the verifier as a child process, to allow for concurrent solving. If the verifier succeeds, Qicc labels the function/node as successfully verified, otherwise, Qicc attempts verification on the caller/parent of that function. Qicc reports success if all assertions have been verified; Qicc reports failure if it finds a function with no parent that cannot be verified.

Listing 2. Original Code

```
1 void main(){
2     int n, x = 1;
3
4     while (n < 1000){
5         assert(x == 1);
6         n += x; }}
```

Listing 3. After Extraction

```
1 void main(){
2     int n, x = 1;
3     while (n < 1000) e1(&n, x);}
4
5 int e1(int *n, int x){
6     assume(*n < 1000);
7     assert(x == 1);
8     *n += x;}
```

Listings 2 and 3 show the result of a single extraction. Qicc extracts the loop condition and inserts it into the extracted function, `e1`, that can be also used for checking any assertions in the loop. As shown in the listings, the assertion in `e1` cannot be shown to always hold with the information present from `e1`, as `x` is taken as parameter. In such cases, the underlying verifier will fail and Qicc will execute that tool on the caller of `e1` (in this case, that is the main function).

To find local loops, we first identify all *Strongly Connected Components (SCCs)* in the CFG using Tarjan's Algorithm [6]. By definition, every node in an SCC can be reached from every other node, and each SCC contains all such nodes [1] [6]. To find nested loops, we remove the root of the loop and apply the algorithm to the rest of the SCC, this process will break apart the SCC and will reveal any nested loops. Once the loops are identified, each loop is checked for locality by using depth-first search on the root. We implemented the identification and extraction algorithms in OCaml using CIL [5]. Tool source is available with supplementary material [2].

4 EVALUATION

Since Qicc was designed to take advantage of cases when loop bodies are sufficient to prove the assertion, we say that Qicc *hits* when that is the case and *misses* otherwise. We had the following research questions: How much faster is Qicc when it hits? How costly are misses?

We compared performance of Qicc to CBMC on 10 benchmarks selected to cover a broad range of cases; we highlight four in Table 1. The `2 Sum fixed` examples are the motivating example in Section 1; `variable` accepts arrays with arbitrary bounds instead; `Factorial Sum` has an unbounded loop but its body is sufficient to prove the assertion.

The `2 Sum fixed` examples are cases where Qicc hits; however, with small input size Qicc's overhead is larger than CBMC's running time. With large input size, however, Qicc's performance is much better than that of CBMC, as the number of unrollings required grows much faster with nested loops. These examples also illustrate that Qicc's performance scales very well with input size, as the increase in Qicc's running time is minimal. We argue that the overhead is manageable when considering the performance boost gained for loops with large bounds. Qicc was also able to quickly check examples like `Factorial Sum`, where CBMC timed-out. For `variable` loops where Qicc misses, both tools timed-out. We did not study how frequent hits and misses are in the real world, and this is a threat to validity. We plan to study this in the future.

Table 1. Benchmark Results

Test/Tool	Time (seconds)	
	CBMC	Qicc (CBMC)
2 Sum - fixed 10	0.7	4.9
2 Sum - fixed 100	98.8	5.8
2 Sum - variable	>10m	>10m
Factorial sum	>10m	8.4

5 CONCLUSION AND FUTURE WORK

Loops with high or arbitrary bounds are very common in programs, and it can often be helpful to verify assertions in them. BMCs like CBMC struggle with such loops due to the number of required unrollings. As demonstrated in this abstract, isolated segments can be sufficient to prove assertions and the proof can be done quickly. Qicc builds on these ideas and is able to considerably improve performance of CBMC for such cases, while introducing only a manageable overhead. In the future, we intend to benchmark Qicc on real-world programs to find the frequency of hits and misses.

REFERENCES

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, techniques, and tools second edition*. Pearson Education Addison Wesly New York, 2007.

[2] Murad Akhundov. Qicc. https://github.com/MuradAkh/Qicc, 2019.

[3] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu, et al. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.

[4] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[5] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, pages 213–228. Springer, 2002.

[6] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.