POPL: U: Expediting Verification of Assertions in Loops by Isolation

Murad Akhundov
murad.akhundov@mail.utoronto.ca
University of Toronto
Toronto, Canada

KEYWORDS

Program Verification, Bounded Model Checking, Program Analysis

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. For each input the verifier will either prove assertions successfully, find counter examples for an assertion; or fail to prove or terminate. These verifiers often struggle to check assertions in unbounded loops, loops with high bounds, or nested loops.

In this paper, we will focus on Bounded Model Checking [5], a popular verification technique. CBMC [8], a state of the art Bounded Model Checker (BMC) for C, takes over 10 minutes to prove the example in Listing 1 when SIZE = 200. 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  //check if any two elements in arr add to s
2  int two_sum_prev(int *arr, int s){
3   // loop1
4   for (int i = 0; i < SIZE; i++){
5     //SIZE is some fixed number
6     // loop2
7     for (int j = 0; j < i; j++){
8       assert(j < SIZE);
9       if(arr[j] + arr[i] == s) printf("found!");
10      }
11    }
12
13    // loop3
14    for (int i = 0; i < SIZE; i++){
15      printf("%d\n", a[i]);
16    }
17 }
```

The key observation that our technique exploits is that inner portions 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 on line 8 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 is bounded (i.e., `j` is eventually `i`). We propose a technique to find these inner portions and expedite verification. Our tool, Qicc (Quick Isolating Checker for C), performs control-flow analysis to identify these sufficient portions and isolate them so that they can be verified individually, quickly, and concurrently.

2 BACKGROUND

This section introduces necessary terminology and background to understand our approach. Specifically, this section provides a brief overview of Bounded Model Checkers (BMCs) and introduces *control-flow graphs*, which we use to model programs, and *extracted loops*, which are the isolated components of control-flow-graphs that Qicc verifies individually.

Bounded Model Checking Programs

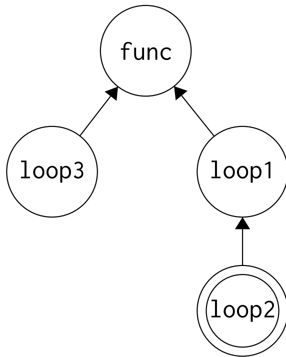
Bounded Model Checking [4] is a popular verification technique originally defined for transition systems. The technique is commonly used for verification of both software and hardware [5]. When verifying programs, Bounded Model Checkers (BMCs) unfold all loops in a program with assertions up to n iterations, we call n the *unfolding limit*. They then generate and solve a SAT expression.

BMCs often struggle to check assertions in unbounded loops, loops with high bounds, or nested loops. This is because, for unbounded loops, the number of unfoldings isn't statically known; with high loop bounds, a large number of loop unfoldings is required; and with nested loops, even a small number of loop unfoldings results in a large SAT encoding. Consider the example in Listing 1. A BMC needs to unfold the loops $O(\texttt{SIZE}^2)$ times to prove the assertion, when `SIZE = 200`, CBMC [8] takes over ten minutes.

Control Flow Graphs and Loops

A *Control-Flow Graph (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* (resp. exit) if there is only one edge coming into (resp. leaving) G'

Figure 1: Caller Tree of Listing 1



Circled node represents an extracted segment immediately containing an assertion

from (resp. to) the rest of graph G. G' is *local* if it has a single entry and exit. We say G' is a *loop* if G' contains at least two nodes, and if every node in G' can be reached from every other node in G'. 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 extracted functions. We claim, but do not prove in this paper, that if an assertion holds when evaluating only the extracted loop, then the assertion will hold for the program as a whole. Qicc uses CBMC [8] but the same approach can be used with other tools.

The next three subsections describe three major steps of our approach: verification, loop extraction, and loop identification.

Verification

When checking extracted loops, Qicc first forms a *caller tree* from functions of extracted loops. Qicc can guarantee that each extracted loop function has exactly one caller because it is a loop body copied from only one location. Next Qicc finds all functions/tree nodes with assertions. For each individual extracted loop, Qicc calls the verifier. 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.

To allow for concurrent solving, for each node, the verifier is spawned as a child process. Every node of the tree contains a mutex that needs to be acquired before a child process can be spawned, preventing repeated work in case two children of a node fail to be verified and both try to call the verifier on the same node. When a verifier succeeds at a node, all its children are labeled as solved.

Figure 1 shows the caller tree produced by Qicc from Listing 1. Qicc will initially only execute CBMC on $loop_2$, as it is the only loop containing an assertion. After failing to prove the assertion at $loop_2$, Qicc will execute CBMC on $loop_1$, and successfully prove the assertion without attempting to prove the entire function.

Loop Extraction

Listing 2: Original Code

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

Listing 3: After Extraction

```
1 int main(){
2   int n, x = 1;
3   while (n < 1000){
4     e1(&n, x);
5   }
6 }
7
8 // extracted loop
9 int e1(int *n, int x){
10   assume(*n < 1000);
11   assert(x == 1);
12   *n += x;
13 }
```

In order to prove sufficient segments in isolation, we extract code from each qualifying loop body to a new function, and replace it with a call to said functions. Pointer depth is calculated at each step, and derefereces are adjusted accordingly.

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 seen in the listings, the assertion in `e1` cannot be shown to always hold with the information present from `e1`, as `x` is taken as a 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). In instances where Qicc executes the verification tool on the main function, it is unable to beat the verification tool as-is, and will terminate slower than the unmodified verification tool due to added overhead from failed proofs. This is because the extracted functions are insufficient to prove the assertions.

Loop Identification

Our extraction technique can only be performed on local loops, as loops with other exits such as breaks or goto statements that jump out of the loop cannot be reasoned about in isolation. To find local loops, we first identify all *Strongly*

Algorithm 1: Loop Identification

```
   Data: Reducible CFG, Γ
   Result: List of loops
 1 σ ← new Stack()
 2 output ← new List()
 3 σ ← push(σ, Γ)
 4 while ¬isEmpty(σ) do
 5 |  γ ← pop(σ)
 6 |  components ← Tarjan(γ)
 7 |  for φ ← components do
 8 |  |  if size(φ) < 2 then
 9 |  |  |  continue
10 |  |  else
11 |  |  |  if isLocal(φ) then
12 |  |  |  |  output ← push(output, φ)
13 |  |  |  end
14 |  |  |  φ_body ← removeRoot(φ)
15 |  |  |  σ ← push(σ, φ_body)
16 |  |  end
17 |  end
18 end
19 return output
```

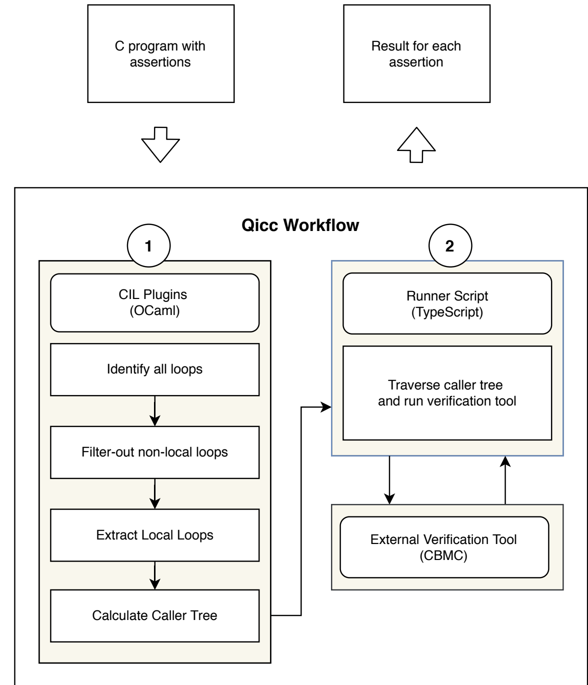


Figure 2: Qicc Workflow

Connected Components (SCCs) in the CFG using Tarjan's Algorithm [12]. By definition, every node in an SCC can be reached from every other node, and each SCC contains all such nodes [1, 12]. Performing a CFG-based analysis (rather than AST/syntax-based) allows us to easily test for loop locality and also handle `goto` statements.

Algorithm 1 describes our CFG analysis. Notice that finding SCCs does not detect nested loops, as nested loops are part of the outer loop SCC. To find nested loops, we remove the root of the loop (using `removeRoot` function on line 14) and apply the algorithm to the rest of the nodes. This process breaks apart the SCC and reveals any nested loops. Once the loops are identified, each loop is checked for locality using depth-first search on the loop root (using `isLocal` function on line 11). We ignore SCCs of size one as they do not fit our definition of a loop and can represent any program statement.

Implementation

Figure 2 displays the workflow of our tool. We implemented the identification and extraction algorithms in OCaml as CIL plugins [11]. We used TypeScript to implement the concurrent algorithm that utilizes CIL plugins, CBMC, and the caller tree to produce the result. Our implementation is limited to a subset of C that does not support array syntax (we encode arrays as pointers instead) and recursion, but we plan to eliminate these limitations in the future. The tool has been tested on Ubuntu 18.04 with OCaml v4.05, GCC v4.8, and Node.js v8.10. Please refer to supplementary material for the tool source and usage instructions [2].

4 EVALUATION

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

Experimentation

We compared performance of Qicc to CBMC on ten small programs written to cover a broad range of cases related to loops. The programs had varying locations of assertions, depth of loops, and required loop iterations. We highlight five results in Table 1. The remaining examples are available in the supplemental material [2]. The `2 Sum fixed` examples are same as Listing 1; `variable` accepts arrays with arbitrary bounds instead; `Factorial Sum` has an unbounded loop but its body is sufficient to prove the assertion, and `Loop mult.` has a loop body insufficient to prove the assertion. `Constraint Board` has an expensive-to-unfold loop preceding another loop with assertion. For fair comparison, we tested with the

Table 1: Experiment Results

Test/Tool	Time (seconds)	
	CBMC	Qicc (with 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
Loop mult.	0.2	3.2
Constraint Board	>10m	6.3

unwinding limit set to the minimum required to verify the assertions (if a minimum exists).

We conducted our experiments on Ubuntu 18.04 with 8 GB of memory, and a quad-core Intel Core i7 processor at 1.8Ghz. We ran each expriment twice and presented the average of two runs.

Results and analysis

The `2 Sum fixed` examples are cases where Qicc hits; but, with small input size Qicc's overhead is larger than CBMC's running time. However, as seen on Figure 3, with large input size, Qicc's performance is much better than that of CBMC, as the number of unfoldings 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. In the `Loop mult.` example, where Qicc misses, the overhead is similar to the `2 Sum fixed` examples.

We argue that the overhead is manageable when considering the performance boost gained for loops with arbitrary or large bounds. Qicc was also able to quickly check examples like `Factorial Sum` and `Constraint Board`, where CBMC timed-out. For `variable` loops where Qicc misses, both tools timed-out.

Threats to validity

We have identified two threats to validity of our evaluation. First, our testing was limited to a small number of short programs, and we did not test Qicc on complex code. However, since Qicc primarily targets loops, we have varied the depth, size, and positions of assertions relative to the loops in target programs. This gave us a good estimation of how well Qicc will do on different types of loops.

Second, we have also not investigated the frequency of hits or misses in real-world programs. However, our results show that the cost of a miss is likely to be far smaller than the benefit of a hit. We plan to study how Qicc performs on real programs in the future.

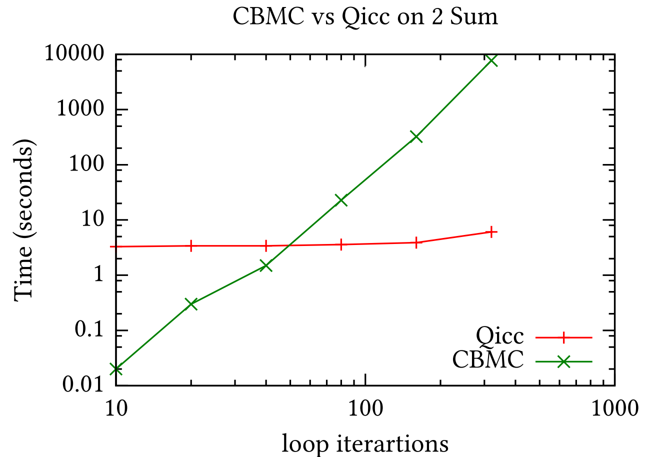


Figure 3: Runtime of Qicc and CBMC on 2 Sum fixed example, with varied `SIZE`, Log Scale

5 RELATED WORK

In this section, we describe tools and techniques most relevant to our approach, including previous attempts with similar goals, and techniques similar to our loop identification and isolation.

Program Slicing. Program slicing looks for portions of a program affected by a criteria [13]. Slicing can be used to narrow the potential region [6] impacting an assertion and can be used in complement to Qicc.

Large Block Encoding. Large Block Encoding (LBE) [3] splits programs into segments in a similar way to Qicc, and is used to allow reasoning about larger chunks of programs at a time, reducing number of paths and allowing for easier verification. Unlike program slicing, LBE does not analyse the potential impact of a statement or region.

SymDiff. SymDiff [9] is a differential program verifier. It performs a similar loop transformation to Qicc, by turning loops into tail-recursive procedures [10]. However, both SymDiff and LBE differ from Qicc as they are not mechanisms to reason about segments in isolation.

Checking Array Properties. To check properties in large arrays, Jana et. al. [7] removed loop root and in-lined the body, assigning loop variables non-deterministic values. Unlike Qicc, their approach still performs verification on the entire program rather than an isolated segment, and is limited to syntactic loops (not gotos). However, this approach may be more desirable when Qicc misses.

6 CONCLUSION

Nested loops and 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 unfoldings. As demonstrated in this paper, isolated segments of the program-under-test can often be sufficient to prove the 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. Qicc does so with manageable overhead, allowing users to quickly verify assertions in loops without running expensive verification on the entire program.

We demonstrated effectiveness of our approach on C programs, but the same technique can be used on other languages as our analysis is based on generic Control Flow Graphs, independent of the language constructs.

7 FUTURE WORK

We intend to expand Qicc's support of the C language constructs, including arrays and recursion. This will allows us to benchmark Qicc on real-world code repositories, and to find the frequency of hits and misses, as well as get a better sense of the performance overhead added by Qicc when testing real world programs.

To improve performance, we also plan to add a mechanism to reuse information from previously attempted proofs and insert additional facts into loop bodies as assumptions, similar to what Qicc already does for `while` loop conditions. We also intend to insert additional facts such as constant variables that can be identified using static analysis. This will help us outperform the underlying verification tool in a broader range of cases.

We have discussed how our method allows portions of a program to be verified concurrently in Section 3, however we did not study if this yields tangible performance improvements. We plan to study this by running Qicc on programs that produce wider caller trees.

In addition to independently verifying extracted loops, we intend to test the same caller-tree-based verification technique on function calls. While functions do not need to be isolated further, they can be called in multiple places, which creates additional complexity when generating the caller tree.

Finally, we are in the process of formally defining Qicc's loop identification algorithm, and proving its termination and soundness. We also plan to provide a formal proof that proving assertions within inner segments identified by Qicc is sufficient to claim that the assertion holds for the program as a whole.

ACKNOWLEDGMENTS

I was mentored for this work by Federico Mora and advised by Marsha Chechik. Qicc is part of the Client Specific Equivalence Checking project. For more information, see https://client-specific-equivalence-checker.github.io

REFERENCES

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, techniques, and tools second edition.* Pearson Education Addison Wesly New York, 2007.

[2] Akhundov, M. Qicc: Quick Isolating Checker for C. https://github.com/MuradAkh/Qicc, 2020.

[3] Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M. E., and Sebastiani, R. Software model checking via large-block encoding. In *2009 Formal Methods in Computer-Aided Design* (2009), IEEE, pp. 25–32.

[4] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems* (1999), Springer, pp. 193–207.

[5] Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., Zhu, Y., et al. Bounded model checking. *Advances in computers 58*, 11 (2003), 117–148.

[6] DeMillo, R. A., Pan, H., and Spafford, E. H. Critical slicing for software fault localization. In *ACM SIGSOFT Software Engineering Notes* (1996), vol. 21, ACM, pp. 121–134.

[7] Jana, A., Khedker, U. P., Datar, A., Venkatesh, R., and Niyas, C. Scaling bounded model checking by transforming programs with arrays. In *International Symposium on Logic-Based Program Synthesis and Transformation* (2016), Springer, pp. 275–292.

[8] Kroening, D., and Tautschnig, M. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2014), Springer, pp. 389–391.

[9] Lahiri, S. K., Hawblitzel, C., Kawaguchi, M., and Rebêlo, H. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification* (2012), Springer, pp. 712–717.

[10] Lahiri, S. K., McMillan, K. L., Sharma, R., and Hawblitzel, C. Differential assertion checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), ACM, pp. 345–355.

[11] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction* (2002), Springer, pp. 213–228.

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

[13] Weiser, M. Program slicing. In *Proceedings of the 5th international conference on Software engineering* (1981), IEEE Press, pp. 439–449.