GLEE: A WLP-based Bounded Verification Tool for GCL Programs

Thomas van Maaren (9825827)
Tjalle Schoonderwoerd (4771931)
Lars Slijkoord (7767080)
Utrecht University
Utrecht, Netherlands

Abstract

We present the tool GLEE, a tool for performing automated analysis of GCL programs using bounded symbolic verification based on weakest liberal preconditions.

GLEE can handle all basic functionality of GCL, as well as make use of user-annotated loop invariants.

Its implementation features multiple optimisations and heuristics, making ranging from a couple percent, to around 90% faster verification in some situations.

1 Introduction

Symbolic execution is a program analysis technique which can be used to verify whether a program meets its specification. In contrast to regular testing, the program is not executed using concrete states, but using symbolic values in place of variables. Such symbolic execution will create one or multiple formulae expressing properties about the correctness of the program, which can be checked for validity using model checker, typically a *satisfiability modulo theories* (SMT) solver [5]. By using such a technique, automatic verification programs attempt to reason about all potential executions of a program[1] rather than a finite number of them as in traditional testing.

Symbolic execution techniques can be divided into two categories: forwards and backwards. In forwards symbolic execution, a symbolic state of the program, mapping variables to symbolic values, is kept and updated throughout the process, checking if any assertions encountered are implied by the symbolic state. In contrast, with backwards symbolic execution, the process works backwards from the specification, calculating which conditions must hold before executing the program in order to ensure the assertions hold.

In this paper, we will mainly focus on backwards symbolic execution.

1.1 Key Challenges

In theory, symbolic execution can guarantee that a program meets its specification, without ever giving a false positive (where the program has a bug which is not caught) or false negative (where the program is correct, but is not identified as such) as a result.

In practice however, there are multiple key challenges which make it impossible to actually exhaustively analyse every program path [1]. Some of these are:

- *Representing memory* – Programs typically make use of complex data structures and pointers, which are more difficult to represent than simple data types such as numbers.
- *Representing environment* – In addition to complex data structures, many programs also interact with an environment, e.g. a file system or network. These side effects also need to be modelled, and it might prove infeasible to check every potential side effect.
- *Path explosion* – Many language constructs cause conditional execution, which might lead to an exponentially increasing number of execution paths which need to be verified, increasing both the load on the memory and the time taken to verify all of them. Loops can even cause a potentially infinite number of paths, making it impossible to exhaustively verify all paths.[2]
- *Constraint solving* – While modern model checkers are capable of verifying large formulae, doing so may take a significant amount of time, especially for large formulae. Even if it is possible to fully verify a program, solving all constraints might still take too long for any practical application.

In this paper, we will focus mainly on the problems of path explosion and constraint solving.

1.2 Related Work

There are several other ways to keep path explosion minimised besides the technique we present in this paper. One of the ways to keep it minimised is by prioritising path exploration with the use of heuristics as explained in [3]. Their paper lists 4 different approaches to limit path explosion with heuristics. The first effective approach is by using the static control-flow graph (CFG) to guide exploration toward the path closest from an uncovered instruction. The second approach is by using random exploration across paths, choosing randomly at branches to either explore one before the other. The third approach is to interleave symbolic exploration with random testing, which combines random testing to quickly reach deep exploration states with symbolic execution. The fourth and final approach discussed is combining symbolic execution with evolutionary search. This uses a fitness function to decide on exploration of certain paths.

Besides heuristics, their paper also discusses sound program analysis techniques. These include merging explored paths statically that are then passed to the constraint solver. Another technique is defined as using compositional techniques to improve symbolic execution by caching and reusing analysis of lower-level functions. This is not needed nor implementable in our case, since the variation of GCL we verify does not have function calls. Lastly there is the possibility of pruning redundant paths during exploration, when

[1]In practice, there are multiple reasons why it might still be impossible to reason about every potential execution. See section 1.1.

[2]However, there are cases where it is possible, and there exist heuristics to attempt to perform sound verification of loops using a bounded number of paths.

a path reaches a point in the program with the same constraints, after which the rest of the path is already known and verified.

The previous part looked at some forms of reduction in path exploration, but another way to reduce the runtime is by optimising the generation of constraints. In [3], irrelevant constraint elimination is mentioned, where feasibility is for example checked on an negated branch was would be the case with WLP. If the input generated contains variables not found in the negated branch, they can be removed since they are irrelevant. Another way to optimise would be to do something called incremental solving. Constraint sets generated often have large parts which are similar between different sets. This means the solution of a certain set can be reused in another set. This is mostly used in bounded symbolic execution such as KLEE where it can be used in for example the so called counter-example caching scheme [2]. Our program also has some kind of incremental solving, in the sense that the sets are stored in the environment of the solver incrementally.

1.3 Our Work

We present a prototype tool for the automatic verification of GCL programs, using backwards symbolic execution based on weakest liberal preconditions. The tool supports all basic features of GCL, including integer, boolean and array types, if-statements, while loops, and specifications using `assert` and `assume`. On top of that, it supports verifying annotated loop invariants.

The implementation includes various optimisations and heuristics like incremental calculations, pruning infeasible paths and front-end simplification. Details of the techniques used are explained in section 3.

2 Preliminaries

We use a variant of GCL, a simple programming language first defined by Edsger Dijkstra in [6]. The exact syntax, as well as a parser for this language (written in Haskell) can be found on GitHub.[3] Here, the benchmark programs we used can be found as well.

Bounded Verification. Since it is not possible to exhaustively verify all execution paths (see section 1.1), we only verify *full* paths of some bounded length, controlled by a parameter K. This means only paths that reach the end of the program within K steps are verified.

Weakest Preconditions. The weakest precondition (WP) of a statement S with respect to some boolean formula Q (the postcondition), is the weakest formula P such that the Hoare triple $\{P\}\ S\ \{Q\}$ is valid and S terminates when executed from some state satisfying P [6]. It is denoted $P = wp(S, Q)$.

A weakest liberal precondition (WLP) is similar to a WP, but drops the termination requirement. It is denoted $P = wlp(S, Q)$.

3 Implementation Techniques

Next, we will highlight the techniques we used and the choices we made for implementing GLEE.[4]

GLEE is implemented using Haskell. The reasons for this choice are that it is a lazy language, and in particular because algebraic data types provide an elegant way to represent programs.[5] We use the Z3 theorem prover[4] as the back-end SMT solver. Z3 is used to check any expression created by GLEE for satisfiability.

3.1 Basic implementation

Execution tree. The abstract syntax tree representing the program is first transformed into an execution tree. Every node contains a single statements, and any number of children. Any path from the root of the execution tree is a program path, so any node with more than one child is a branching point.

The tree is constructed as follows:

- Any statement of the form `if` g `then` S_1 `else` S_2 is transformed into a single `skip`-node, with one child branch containing the assumption that g holds, S_1 and then the rest of the program, the other containing the assumption that $\neg g$ holds, S_2 and then the rest of the program.
- Loops are unrolled into series of if-statements (this will be explained in more detail later), and are then handled identically to if-statements.
- In a sequence of statements S_1 and S_2, first, an execution tree for both is created. Then all leaves of the execution tree of S_1 are replaced with nodes that have the second execution tree as child.

During the construction of the tree, the program keeps track of any variables that are defined. This information is used in the transformation of blocks, and expressions with quantifiers, where any duplicate variable names are renamed in order to prevent shadowing.

Loop unrolling. In order to deal with loops in the program code, they are unrolled by repeatedly creating two branches, one in which the loop exits (equivalent to `assume ~g`), one in which the loop continues (equivalent to `assume` g; S; `while` g `do` S).

Bounding. Unless a loop invariant is specified, it is impossible to verify all possible execution paths of a program, since we cannot tell beforehand how often a loop will be executed and would have to verify an infinite number of executions. To avoid this problem, we cut any branches of the execution tree that are longer than K. Since the implementation language, Haskell, is lazy, this means that execution paths that are not verified are also not generated. This avoids the program getting into infinite recursion.

3.2 Optimisations

Incremental WLP calculations. Instead of fully calculating each WLP before using Z3 to verify it, it is calculated on the fly while the tree is traversed. The steps to do this are as follows:

- Whenever an expression needs to be used (for verifying assumptions and assertions, and at the end of an execution path), it is transformed as required by potential earlier statements, using a saved predicate transformer function.
- Any time an assumption is encountered, it is stored in the Z3 context and saved for the rest of the execution path.

[3]https://github.com/wooshrow/gclparser

[4]The full source code of the program can be found on GitHub: https://github.com/Tjalle-S/PSV-project/tree/main.

[5]And, of course, because the parser is only available in Haskell.

- Any time an (intermediate) assertion is encountered, it is immediately verified.
- Any time any other statement is encountered along the path, the saved predicate transformer is updated and passed on.
- Whenever the end of an execution path is reached, the calculated WLP is verified.

This approach reduces the number of calls to Z3, and allows easily checking for infeasible paths.

Infeasibility checks. Any time an assumption is encountered, the current list of assumptions (the branch condition) is checked for feasibility using Z3. If there is no assignment of variables satisfying the assumptions, the branch is immediately accepted. Otherwise, verification continues normally.

Verifying loop invariants[6]. For verifying loop invariants, we use the standard inference rule for while loops, in the partial correctness implementation. Given a Hoare triple $\{P\}$ `while g do S` $\{Q\}$ and an invariant I, we need to verify the following conditions:

(1) $P \implies I$
(2) $\{I \wedge \neg g\}\ S\ \{I\}$
(3) $(I \wedge \neg c) \implies Q$

In the verification, this corresponds with the following steps:

(1) Add an assertion that I holds before the loop.
(2) Create a branch. This contains the assumption that $I \wedge g$ holds, then S, then the assertion that I holds.
(3) Create another branch. Add an assumption that $I \wedge \neg g$ holds after the loop, and remove the loop itself, continuing with the rest of the program.

In both branches, the names of variables that are modified in S need to be replaced by fresh ones, since assumptions made before the loop about these variables might not hold within or after the loop.

Expression simplification. Before an expression is sent to Z3, it is simplified. This simplification is done on binary operators and negation. The rules used are common rules for simplifying binary operators with respect to boolean algebra and mathematical simplification.

4 Results

We ran GLEE on four benchmark programs. These are slightly modified versions of the programs `memberOf`, `divNyN` and `pullUp`, and `bsort`. The modification involved adding loop invariant to those programs that did not have it, to test the effect on those programs as well. For most benchmarks, we do not check invariants, as correct invariants mean that relatively few calculations are required.

There are statistics on the following data:

- Running time. The time it took for GLEE to complete the verification.
- Number of paths verified. This means all paths that are followed to the end, and not pruned.
- Number of branches pruned. This means the amount of times a branch has been pruned, regardless of the number of paths that are cut off by the prune.
- Formula size. This means the total number of leafs (literals, variables and `sizeof`) of formulae sent to Z3 for verification. Keep in mind that some of these formulae may be used multiple times, due to the incremental calculations.

There is an experiment parameter, N, which controlled the minimum length of arrays for most programs (and the divisor for `divByN`.

4.1 Invalid Programs

The invalid programs were run at a depth sufficient to detect the violation for al

In figure 1, it can be seen that the violation in the invalid variants of the benchmark programs is found in a very short amount of time. Figures 2 and 3 reveal that the reason is that they are found almost immediately, after which the program can immediately report the violation and stop. Figure 4 shows that not many paths needed to be pruned either.

In this set of benchmarks, `bsort` is an exception, since it is the only one that was run with invariant detection on. The reason for this is that this is the benchmark intended for this optimisation, and that GLEE was unable to verify `bsort` with sufficient depth to find the violation for high values of N. `memberOf` was run with $K = 66$, `divByN` with $K = 50$ and `pullUp` with $K = 58$. For `bsort`, depth is irrelevant, as long as all paths reach the end of the program. Any higher value of K has no influence, since loops are removed by the invariant detection.

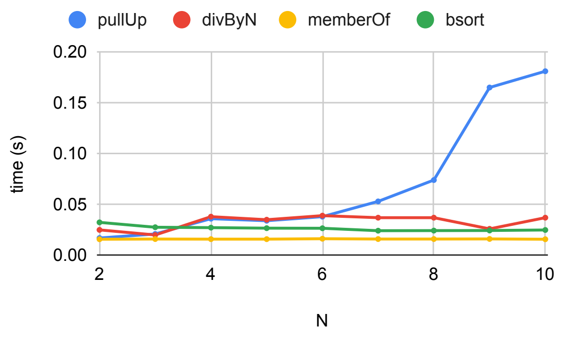

Figure 1: Runtime of the invalid versions of the programs.

4.2 Valid Programs

All benchmarks on the valid variants of the programs are run with $N = 10$.

Incremental WLP Calculations. This was an optimisation that was made early during the development of GLEE, and is deeply ingrained into the program. It is therefore not a heuristic that can be enabled or disabled, and as such, no benchmarks are available. However, during development, we saw that with $K = 60$, verifying the programs `memberOf` and `divByN` was roughly 60% faster than before.

[6]This is the implementation for the loop-invariant optional.

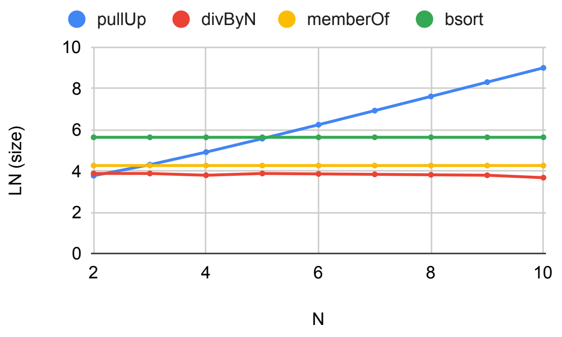

Figure 2: Formulae size of the invalid versions of the programs (where size is transformed with the natural logarithm).

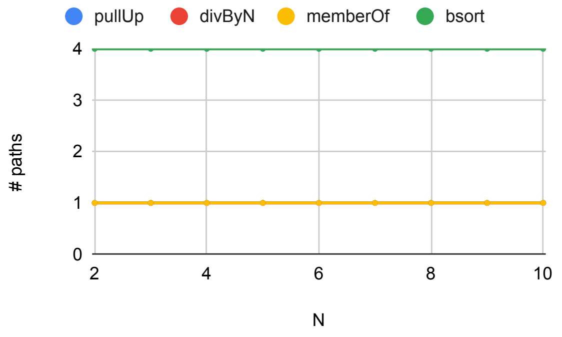

Figure 3: Amount of paths of the invalid versions of the programs.

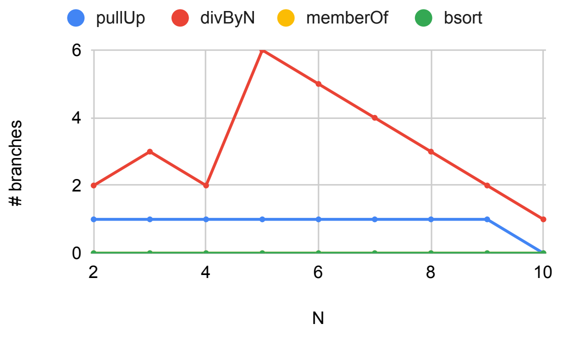

Figure 4: Amount of branched pruned of the invalid versions of the programs.

When running the program with no heuristics enabled, the runtime of the programs drives up exponentially for most programs as K is increased, as can be seen in figure 5. We will now compare the different heuristics with the results seen in this figure.

Infeasibility Checks. In figure 6 we can see that depending on the program, there may or may not be an increase or decrease in runtime. We see that with pruning, `memberOf` is roughly 40% slower, while `pullUp` is 30% faster, `bsort` is 70% faster and `divByN` is more than 90% faster. This wide range of increase and decrease can be related to the simple fact that `memberOf` does not have many of its paths pruned, thus the checking for infeasible paths takes extra time, while the other programs have most of its paths pruned.

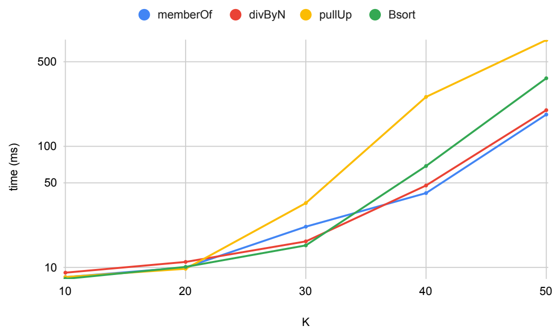

Figure 5: Runtime of the valid versions of the programs.

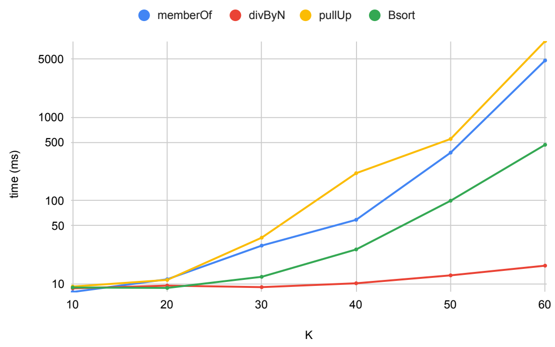

Figure 6: Runtime of the valid versions of the programs with infeasibility checks.

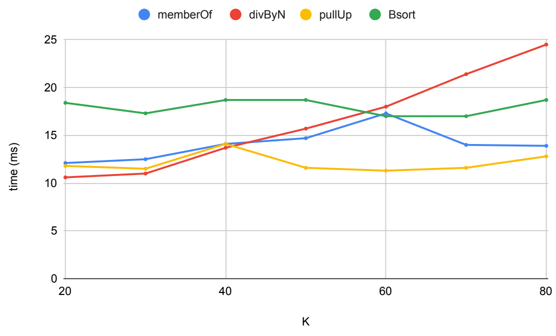

Figure 7: Runtime of the valid versions of the programs with loop invariants.

Verifying Loop Invariants. In figure 7 we are able to see the biggest speed-up of all heuristics when our tool uses loop invariants. All programs take around 15 ms to complete, even when the depth is increased. From this we can conclude that the speedup grows while the depth grows, as in other cases the time increases with larger values of K while here it stays roughly constant. If the invariant are correctly annotated, we can conclude that the tool is sound and complete.

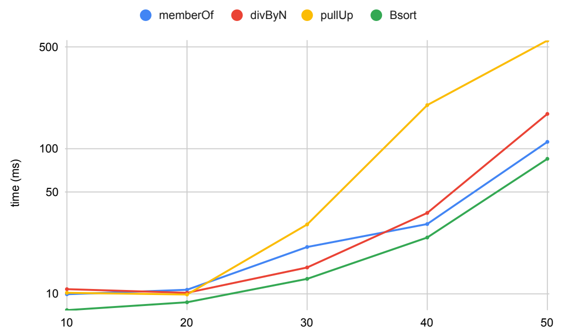

Figure 8: Runtime of the valid versions of the programs with expression simplification.

Expression Simplification. In figure 8 we see almost no decrease in runtime with expression simplification compared to the runtime without it as seen in figure 5. All programs do receive some speed-up, although not significantly much, except `bsort`. `bsort` appears to profit from having the expression solved before solving the WLP.

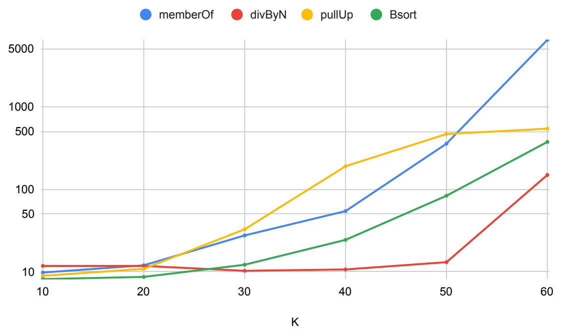

Figure 9: Runtime of the valid versions of the programs with expression simplification and infeasibility checking.

Heuristics combined. In figure 9 we have enabled all heuristics except for the using loop invariants. We can see mostly the same trends found in the results from checking for infeasibility. The result are exactly as expected, just slightly faster runtimes than the times found in figure 6.

5 Conclusion

In this project we set out to write a verifier for GCL programs. We did this by creating a WLP for every execution path of our program. These are verified on the fly with incremental WLP verification. To reduce the amount of the paths we also implemented a pruner and to reduce the size of the expressions we implemented a front-end simplifier. As a bonus we have made it possible for the verifier to make use of user-specified invariants, reducing the size of the execution tree and the WLP's. In the end we saw that all our heuristics improved the performance of our program in most cases.

5.1 Future Work

There are some optimisations which are not yet implemented in GLEE, but would be good to have. Some of those are:

- There are two different ways to create quantifiers in Z3. We found that the one we currently use (using global variables) seems significantly slower than the alternative (using De Bruijn indices). However, we have not been able to get this to work, as it lead to Z3 diverging and never finishing in some cases.
- Infeasibility checking leads to significantly worse performance on the benchmark program `memberOf`. Ideally, there would be some heuristic to detect when pruning is unlikely to have an effect, so the checks could automatically be skipped.

References

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (2018).
[2] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, Vol. 8. 209–224.
[3] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.
[4] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.
[5] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77.
[6] Edsger W Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457.