# KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs

Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan

Fujitsu Labs of America, CA
{gli, ighosh, sree.rajan}@us.fujitsu.com

**Abstract.** We present the first symbolic execution and automatic test generation tool for C++ programs. First we describe our effort in extending an existing symbolic execution tool for C programs to handle C++ programs. We then show how we made this tool generic, efficient and usable to handle real-life industrial applications. Novel features include extended symbolic virtual machine, library optimization for C and C++, object-level execution and reasoning, interfacing with specific type of efficient solvers, and semi-automatic unit and component testing. This tool is being used to assist the validation and testing of industrial software as well as publicly available programs written using the C++ language.

## 1 Introduction

With the ubiquitous presence of software programs permeating almost all aspects of daily life, providing robust and reliable software has become a necessity. Traditionally, software quality has been assured through manual testing which is tedious, difficult, and often gives poor coverage of the source code especially when availing of random testing approaches. This has led to much recent work in the formal validation arena [4, 1]. One such formal technique is symbolic execution which can be used to automatically generate test inputs with high structural coverage for the program under test. The widely used symbolic execution engines currently are able to handle C or Java programs only. So far there has been no such formal tool designed specifically for the automatic validation and test generation for C++ programs. Currently C++ is the language of choice for most low-level scientific and performance critical applications in academia and industry. This paper describes our efforts in creating the first industrially usable symbolic execution engine for C++ programs.

Symbolic execution [4, 1] performs the execution of a program on symbolic (open) inputs. It characterizes each program path it explores with a path condition which denotes a series of branching decisions. The solutions to path conditions are the test inputs that will assure that the program under test runs along a particular concrete path during concrete execution. Typically a decision procedure such as a SMT (Satisfiability Modulo Theory) solver is used to find the solutions and prune out false paths. Exhaustive testing is achieved by exploring all true paths. Some sanity properties can also be checked such as memory out-of-bound access, divide-by-zero, and certain types of user-defined assertions.

Our tool is built on top of a symbolic execution engine KLEE [4] which is able to handle sequential C programs (mainly Unix utility programs). Our extended tool addresses the following questions and issues:

- How to extend a symbolic executor to handle C++ features?
- What optimizations are necessary to make the tool efficient and scalable?
- Can the tool work well on industrial applications and other important programs and beat manual testing with minimal manual effort?

## 2 Extended Executor for C++ Programs

As shown in Fig. 1, the tool's flow is similar to KLEE's. A C++ program is compiled into LLVM [7] bytecode, which is interpreted by KLOVER for symbolic execution. To handle the C++ library constructs we use a special C++ library which is described later. After the execution, statistics information and sanity check results are given. The other set of outputs are concrete test cases, which can be replayed in the real setting (*e.g.* compiled by GCC and run in a machine). After the replay, source program coverage is produced by gcov.
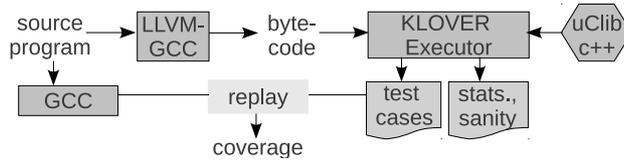


Fig. 1. Overall architecture of KLOVER.

**Virtual Machine State.** A symbolic state in KLOVER models a machine execution state. A register stores a concrete value or a symbolic expression. A memory is organized as components, each of which has a concrete address and an array of bytes recording the value. The fields of a C++ object are allocated consecutive memory blocks. In the following example, the two fields of object 1 (with runtime type $t_1$) satisfies $m_{1,2} = m_{1,1} + \texttt{size}(fd_{1,1})$. The memory blocks of different objects do not have to be consecutive, which can support automatic dynamic resizing. If a pointer can refer to multiple components, then a new state is generated for each possible reference (determined by SMT solving).

| object$_1$ : t$_1$ | | ... | object$_2$ : t$_2$ | | ... |
|---|---|---|---|---|---|
| $(m_{1,1},fd_{1,1})$ | $(m_{1,2},\ fd_{1,2})$ | ... | $(m_{2,1},\ fd_{2,1})$ | ... | ... |

**C++ Language Features.** Most C++ features such as templates and class inheritance are handled by the LLVM-GCC compiler. However, since C++ is far more complicated than C, there may be extra LLVM instructions (mainly intrinsic functions) and external functions which KLEE doesn't handle. Presently KLOVER can handle most of the widely occurring C++ specific LLVM instructions and external functions, and we are extending the tool further to handle the complete set. The new instructions and issues include:

2

– Advanced Instructions. For example, the llvm.stacksave intrinsic is used to remember the current state of the function stack, which is to be restored by llvm.stackrestore. The implementation of these instructions follows their semantics and is quite straightforward.
– Exceptions. An important feature of C++ is to provide built-in support for exceptions. The several llvm.eh. instructions along with a few external functions need to be interpreted in the right exception semantics, *e.g.* propagate the exceptions up the stack. We introduce a specific data structure to represent exceptions, build the exception table, control the bytecode execution flow of exceptions, and interpret exception instructions.
– C++ RTTI. C++'s Run-time Type Information system keeps information about an objects type at runtime. Besides enabling RTTI in LLVM-GCC, we keep track of the the runtime types of objects of polymorphic classes so as to handle operations such as dynamic_cast. Class hierarchy information is inferred from the type definitions in LLVM and the control flow.
– Memory Model . C++'s memory model involves many atomic operations and synchronization intrinsics. The compilation to specific platforms may also involve them. For example, llvm.memory.barrier guarantees ordering between specific pairs of memory access types, and lvm.atomic .load.add performs the add and store atomically. We do not address concurrency in this paper; while in [5] we describe how to extend KLOVER for GPU programs and compare symbolic execution with other symbolic methods [6] for concurrent programs.

**C++ Library.** The C++ standard includes a library for all commonly used data structures and algorithms. We choose and optimize the uClibc++ library [8] so as to improve the performance of symbolic execution. We compile this library into LLVM bytecode and load it into the engine dynamically. We maintain two versions of the C++ library: one for symbolic execution, the other one for handling concrete values and the Just-In-Time compilation of external functions.

## 3 Optimizations

To scale up the tool we adopt a variety of optimizations which are essential to the tool's performance and usability. These optimization are in addition to the ones done in KLEE which KLOVER inherits. The new optimizations can have a huge impact on the quality of results and symbolic execution time, as shown by the following example. We compare the cases without any optimization, with our optimized library, and with our string solver, on the main benchmark program used in [2]. The first two cases could not achieve full branch coverage since the input string is of specific size. In this section we elaborate these optimizations.

| No Optimization | | | +Optimized Lib. | | | +FLA String Solver | | |
|---|---|---|---|---|---|---|---|---|
| #tests | bran. cov. | time | #tests | bran. cov | time | #tests | bran. cov. | time |
| >10,000 | 67% | >2 hr. | 6 | 67% | 6 sec. | 9 | 100% | 3 sec. |

The standard C++ library is designed for concrete execution. Efficient symbolic execution requires rewriting all the C and C++ class implementations to:

(1) avoid unnecessary conditional statements to reduce the number of generated paths; (2) convert expensive expressions into cheaper ones; and (3) build fast decision procedures into the library implementation. KLOVER has optimized a number of commonly-used classes and algorithms with a similar purpose as [3].

**Library Optimization (Operational Approach).** The first optimization technique modifies the body of a function, which will be executed directly. For example, the compare method of the String class is as follows. It will produce only one path regardless of the values of the two input strings (of concrete lengths).

```
_UCXXEXPORT int compare(const basic_string& str) const {
  size_type rlen = vector<Ch, A>::elements;
  if (rlen > str.elements) rlen = str.elements;
    int v = 0; // 1, 0 and -1 stand for gt, eq and lt respectively
    for (size_type i = 0; i < rlen; i++)
      v += (~(!v)+1) & ((operator[](i)>str[i]) - (operator[](i)<str[i]));
    v += (~(!v)+1) & ((vector<Ch, A>::elements > str.elements) -
                      (vector<Ch, A>::elements < str.elements));
    return v;
}
```

**Library Optimization (Relational Approach).** We can build a solver in the source code without "executing" the implementation. For example, the find_last_of method is as follows, where Vr denotes the return value. Function assume informs the executor to record the constraint. This implementation relates the inputs and outputs using logical formulas. It also produces only one path. Building solvers through source code definitions is a core feature of KLOVER.

```
find_last_of (const char c) {
  size_type rlen = vector<Ch, A>::elements;
  assume(Vr >= -1 && Vr < rlen);
  for (size_type i = 0; i != rlen ; i++)
    assume(i <= Vr || operator[](i) != c);
  assume(Vr == -1 || operator[](Vr) == c);
}
```

**Object-level Execution and Reasoning.** One of the main features of C++ is class and object. KLOVER's intermediate language (IL) is extended to model them directly. During the execution, a method call is not immediately expanded to its implementation when it is first encountered. Instead, a "lazy evaluation" approach is adopted to delay the evaluation until needed. Consider the following code. When the condition is encountered, KLOVER builds the expression str.substr(str.find_last_of('/') + 1) = "EasyChair", which can be simplified to str = s1 + "/EasyChair" for a free string variable s1. KLOVER builds in such simplifications and decision procedures (see next section) for common classes. We may simply use the library definitions of the methods to interpret this expression — now the interpretation is delayed to the condition point. We believe that object level abstractions is crucial for a Object-oriented language like C++.

```
    int k = str.find\_last_of('/');
    string rest = str.substr(k + 1);
    if (rest == "EasyChair")  ...
```

**Specific Solvers.** To further improve the performance of object-level reasoning, we implement off-the-shelf solvers for some common data structures. For instance we've implemented a string solver based on SMT solving which is similar to that in [2]. Consider the above example. Our solver creates the following expression constraining the values and lengths of the string variables. The constraints on the lengths are first extracted and solved to obtain a (minimal) instance for each length. Then the length of each string is set and the string constraint is solved. With such built-in solvers KLOVER not only improves the performance, but also allows more feasible inputs (*e.g.* having variable lengths).

$$\begin{aligned}
&\wedge\ (k = -1\ \wedge\ \forall i \in [0, \mathtt{len}(str)) : str[i] \neq \text{'/'}\ \vee \\
&\quad\ k \in [0, \mathtt{len}(str))\ \wedge\ str[k] = \text{'/'}\ \wedge\ \forall i \in [k+1, \mathtt{len}(str)) : str[i] \neq \text{'/'}) \\
&\wedge\ rest = str[k+1, \mathtt{len}(str))\ \wedge\ \mathtt{len}(rest) = \mathtt{len}(str) - k - 1 \\
&\wedge\ rest = \text{``EasyChair''}\ \wedge\ \mathtt{len}(rest) = 9
\end{aligned}$$

## 4    Experimental Results

KLOVER requires a driver to invoke a C++ program with symbolic inputs. A user is free to mark any input symbolic, but should ensure that the relationship between the inputs is appropriate. Since C++ program encapsulate the members of a class, the driver calls a class's public methods to make an object "symbolic".

KLOVER supports the declaration of an array to have a symbolic length. When an access to such an array is out of bounds; we increase dynamically the array's size to accommodate this access (up to some ceiling). KLOVER also supports the declaration of a possibly null pointer. Through such extensions, KLOVER is able to reduce the manual testing burden significantly over KLEE.

We run KLOVER on some real-life applications developed in Fujitsu on a laptop with two 1.60GHz processors and 2GB memory. Table 1 compares KLOVER with the manual method for unit testing. The size of these classes are about 5,500 lines of code in total. The (semi-automated) drivers for KLOVER are much more succinct and apprehensive than the manually written ones. For each class, KLOVER achieves much higher coverage by producing only a small number of test cases ($<20$). KLOVER's unit testing is able to beat manual testing in all cases, both in line coverage and branch coverage. Similar results have been obtained on other industrial instances, *e.g.* a real application with around 130,000 and 51,000 lines in the *.hh and *cc files respectively.

We also run KLOVER on some C++ applications which are publicly available (*e.g.* at www.sourceforge.com). Table 2 shows the results for SHA-1 (a cryptographic hash function), Balancing AVL tree, a Regular Expression package (ported from java.util.regex), and a URI package for analyzing URLs. For these widely-used algorithms, there exist no prior effort on checking and testing their C++ versions using symbolic execution or other formal methods. Reg.Exp. and URI contain intensive string operations. KLOVER reveals several bugs (infinite loops) in "Reg.Exp." which are missed by the manual testing. The replaying in real settings shows that these bugs are real in 5 test cases.

It is possible that some run-time exception cases (*e.g.* running out of memory) are not covered by KLOVER such that the coverage may not reach 100%.

| Class | Driver LOC (Manual) | Coverage (Unit, Manual) | Driver LOC (KLOVER) | Coverage (Unit, KLOVER) | Exec. Time |
|-------|------|------|------|------|------|
| Class 1 | 547 | 50.6%/27.65% | 73 | 96.4%/78.8% | 2.4s |
| Class 2 | 726 | 96.21%/59.65% | 45 | 100%/75.9% | 0.3s |
| Class 3 | 537 | 93.51%/73.33% | 58 | 97.4%/73.6% | 0.5s |
| Class 4 | 337 | 94.16%/86.11% | 52 | 100%/87.2% | 0.5s |
| Class 5 | 286 | 87.68%/70.27% | 95 | 100%/77% | 2.6s |

**Table 1.** Experimental results of unit testing an industrial application. We compare manual unit testing with KLOVER in terms of the driver's size and the coverage (of format line coverage / branch coverage).

| Prog. | Source LOC | Sanity | Coverage (Manual) | #tests (Manual) | Coverage (KLOVER) | #tests (KLOVER) | Exec. Time |
|-------|------|------|------|------|------|------|------|
| SHA-1 | 450 | Y | — | — | 97.50%/91.67% | 22 | 30s |
| AVLTree | 700 | Y | 81.17%/42.04% | 150 | 92.86%/41.53% | 13 | 10m |
| Reg.Exp. | 3,100 | N* | 58.88%/59.12% | 12 | 87.87%/89.19% | 999(5*) | 37s |
| URI | 2,200 | Y | 86.5%/62.0% | 180 | 89.8%/64.4% | 134 | 2.5m |

**Table 2.** Experimental results on the C++ versions of some publicly available programs. KLOVER checks their sanity and produces test cases.

The coverage with KLOVER can be improved further with refined drivers. We intentionally keep the drivers simple such that they can be written quickly by the users knowing little about the applications. Yet such drivers (with appropriate constraints on the symbolic inputs) are able to achieve high coverage.

**Concluding Remarks.** Our tool is the first symbolic executor and test generation designed and tuned particularly for industrial C++ programs. We plan to further extend the tool and scale it up for larger C++ programs.

# References

1. ANAND, S., PASAREANU, C. S., AND VISSER, W. JPF-SE: A symbolic execution extension to Java pathfinder. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2007).
2. BJØRNER, N., TILLMANN, N., AND VORONKOV, A. Path feasibility analysis for string-manipulating programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2009).
3. BLANC, N., GROCE, A., AND KROENING, D. Verifying C++ with STL containers via predicate abstraction. In *Automated Software Engineering (ASE)* (2007).
4. CADAR, C., DUNBAR, D., AND ENGLER, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).
5. LI, G., GHOSH, I., RAJAN, S. P., AND GOPALAKRISHNAN, G. GKLEE : A symbolic execution and automatic test generation tool for GPU programs, 2011. Draft.
6. LI, G., AND GOPALAKRISHNAN, G. Scalable SMT-based verification of GPU kernel functions. In *Foundations of Software Engineering (SIGSOFT FSE)* (2010).
7. The LLVM compiler infrastructure . http://www.llvm.org/.
8. uClibc++: An embedded C++ library. http://cxx.uclibc.org.