CCmutator: A Mutation Generator for Concurrency Constructs in Multithreaded C/C++ Applications

Markus Kusano and Chao Wang
Virginia Tech, Blacksburg, VA 24061, USA
{mukusano, chaowang}@vt.edu

***Abstract*—We introduce *CCmutator*, a mutation generation tool for multithreaded C/C++ programs written using POSIX threads and the recently standardized C++11 concurrency constructs. *CCmutator* is capable of performing partial mutations and generating higher order mutants, which allow for more focused and complex combinations of elementary mutation operators leading to higher quality mutants. We have implemented *CCmutator* based on the popular Clang/LLVM compiler framework, which allows *CCmutator* to be extremely scalable and robust in handling real-world C/C++ applications. *CCmutator* is also designed in such a way that all mutants of the same order can be generated in parallel, which allows the tool to be easily parallelized on commodity multicore hardware to improve performance.**

I. Introduction

Software testing has always been an expensive part of the software development process, typically taking up more than 50% of the total development cost [3]. Now the situation is exacerbated by the increasingly widespread use of multicore processors, whose computing power can only be unleashed by concurrent software. However, developing concurrent software is known to be difficult. Due to scheduling nondeterminism, multiple runs of the program may exhibit different behaviors even under the same input. Furthermore, the number of thread interleavings is often astronomically large. In addition, programmers today often think sequentially and therefore may overlook the crucial interleaving corner cases [21].

Mutation testing is a fault-based testing technique where small synthetic faults are systematically seeded into a program [17]. Mutation generation tools have been used for sequential software, e.g., to select test cases [9] and evaluate testing methods [5]. However, tools targeting multithreaded C/C++ programs, especially for POSIX threads and the newly standardized C++11 concurrency constructs, are still lacking. Our *CCmutator* tool will fill the gap.

To the best of our knowledge, *CCmutator*, shown in Fig. 1, is the first mutation generation tool for multithreaded C/C++ applications written using the PThreads and C++11 concurrency constructs. PThreads are available on a wide range of platforms spanning from embedded computing to distributed systems. C/C++ have widespread use in low-level systems code. The newly created C++11 standard also defines a *relaxed* memory model and a set of new concurrency constructs.

CCmutator can inject a broad spectrum of concurrency related software bugs into multithreaded C/C++ applications, to quickly generate hundreds or thousands of mutants, each of which may contain concurrency bugs that span across many lines of source code. There will be many applications for such *mutants*, including a direct use of them to provide a controlled way of evaluating software testing and verification tools. For example, with mutants generated from real world multithreaded applications, we can evaluate not only whether a tool is able to find particular types of concurrency bugs, but more importantly, whether its bug detection algorithm can *scale up* to applications of realistic size and complexity.

In the past, testing and verification tools for multithreaded C/C++ applications have been evaluated on two kinds of benchmarks: *small synthetic benchmarks* or *bugs in a few real applications*. Although bugs in real applications are essential for creating practically relevant tools, finding a sufficiently large number of bugs of a specific pattern can be challenging. Even more difficult is finding bug samples of enough variety to form a *comprehensive* benchmark set. For example, we noticed that most of the recent studies are using a small set bugs, e.g., from `Mozilla`, `Apache`, `MySQL`, `Aget`, and `Bzip`, just because they are well-documented and easy to find. This may threaten the validity of the evaluation metrics. In contrast, synthetic benchmarks, such as the security vulnerability related benchmark examples in NIST's Juliet suite [19], are programs created specifically for exhibiting certain bug patterns. They have the advantage of being rich in number and variety. However, their main problem is that the programs tend to be fairly small, and the bugs are *predictable* in the sense that tool developers already know *a priori* what to look for. *CCmutator* can be used to address the aforementioned problems.

We have implemented *CCmutator* based on the popular Clang/LLVM compiler platform. By leveraging the mature development infrastructure provided by LLVM, we are able to complete a robust implementation of *CCmutator* in a relatively short period of time. The LLVM platform also has a wide range of built-in program analysis methods, together with a large and rapidly growing user base. In addition to having a production quality C/C++ front-end (Clang), LLVM also has `llvm-gcc` and many other front-ends under development for languages such as Java and JavaScript. Since *CCmutator* is implemented on the LLVM intermediate representation (IR), it can potentially leverage the aforementioned front-ends to handle other languages. Therefore, we expect *CCmutator* to have a broad impact and be quickly adopted by researchers and developers in this community. The tool is currently available at http://github.com/markus-kusano/CCMutator.

II. The Approach

The overall flow of *CCmutator* is shown in Fig. 1. Specifically, the C/C++ source code is first compiled by Clang into LLVM's intermediate representation (IR) (although any language with a LLVM front-end could be used). All the operations performed by *CCmutator* work on LLVM IR. The application of each mutation operator consists of two steps: the *enumeration* step and the *mutation* step. In the enumeration step, the tool automatically finds *mutation sites*, which are areas in the input code where a chosen mutation operator can be applied. At a bare minimum, *CCmutator* will output these mutation sites to the user, in the format of filenames and line

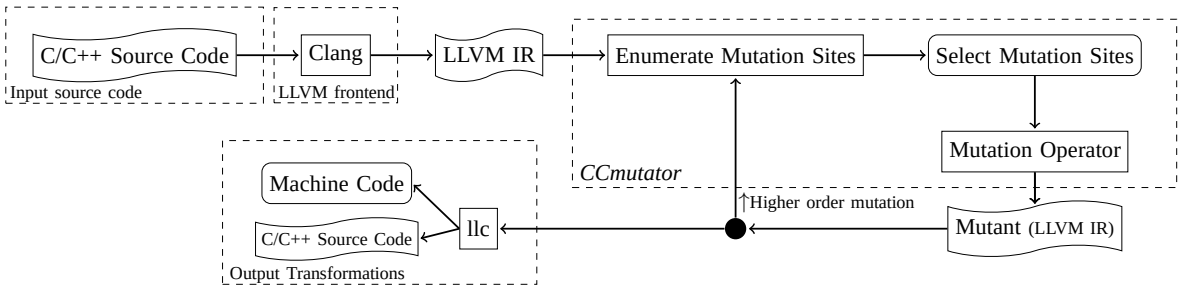

Fig. 1: Overall flow of *CCmutator*. C/C++ was used as the input source code but any language that has an LLVM front-end is supported. LLVM utility llc can convert the output mutant into various forms.

numbers in the original source, allowing the user to decide how to create the mutation. In the mutation step, *CCmutator* applies the mutation operator at the chosen mutation sites to produce the mutant. The output mutant can be used again as the input to *CCmutator* in order to generate higher-order mutants. *CCmutator* also provides a stripped down computer friendly output of the enumeration step for easy automation. The resulting LLVM IR mutant can be passed to other tools to transform it into another desired output format.

The language-independent IR of LLVM resembles a typed assembly language, but is fully readable, allowing for named variables and debug information to persist across mutation phases. Leveraging the mature development infrastructure in LLVM, we have largely avoided the difficulties experienced in implementing other mutation testing tools such as Csaw [10], for which parsing and transforming the C/C++ language in real world applications is a major challenge.

Both LLVM and its IR have been widely used not only in the research community, e.g., serving as foundation of tools such as KLEE [7] and Coredet [4], but also in real industry development systems such as Apple's LLVM compiler [1] and Google's Portable Native Client [12]. The LLVM IR can also be constructed by using various front-ends for C/C++, Java, JavaScript, Haskell and many other languages. Furthermore, tools such as RevGen [8] can convert x86 and ARM binary executables back to the LLVM IR. By leveraging these front-end tools, *CCmutator* will have even broader applications.

The architecture of *CCmutator* is designed to easily make use of multiprocessor hardware. Since each mutation operator is implemented in a separate LLVM *opt* pass, different operators can be applied in parallel. For example, every partial mutation applied to a file can run as its own OS process, allowing as much or as little hardware parallelism as desired by the user. There are no data dependencies between same order mutations as long as their input files are not the same as their output files.

III. Challenges and Our Solutions

In mutation testing, the set of code changes needed to produce a mutant P' from the original program P is called the mutation operator. The mutation operators in *CCmutator* are designed specifically for concurrency constructs. In contrast to mutation operators for sequential programs, which tend to be simple syntactic code changes, concurrency related mutation operators are significantly more involved.

As an example, consider the mutation of lock-based critical sections in Fig. 2 (left), defined by using mutex locks `lock1` and `lock2`. Suppose that we want to split, expand, shift, or shrink the critical sections. If *CCmutator* swaps the two mutex locks-unlock pairs in this code, for instance, it would produce the mutant in Fig. 2 (right), which introduces a deadlock.

However, swapping the two locks in the above code is not a *local* operation. At the very least, one needs to identify whether two lock operations—for example, `mutex_lock()` and `mutex_unlock()`—access the same lock. In real world applications, lock pointers may be used instead of `lock1` and `lock2` to represent the same locks. But it becomes a non-trivial task to figure out whether the lock `p→lk` acquired in one program location is the same as the lock `tmp_lk` released in another program location. In general, one would need to perform an *alias analysis* to find out if `p→lk` and `tmp_lk` are the same lock. In our implementation of *CCmutator*, we have leveraged the availability of a flow-insensitive pointer analysis in LLVM to carry out the aforementioned analysis.

```
Original

1   mutex_lock(lock1);
2   //...
3   if (cond) {
4      mutex_unlock(lock1);
5      return;
6   }
7   mutex_lock(lock2);
8   //...
9   mutex_unlock(lock2);
10  mutex_unlock(lock1);
```

```
Mutant

1   mutex_lock(lock2);
2   //...
3   if (cond) {
4      mutex_unlock(lock2);
5      return;
6   }
7   mutex_lock(lock1);
8   //...
9   mutex_unlock(lock1);
10  mutex_unlock(lock2);
```

Fig. 2: Example: Swapping the mutexes of the lock-unlock pairs to reverse the locking order, thereby inserting a deadlock.

In the *enumeration* step, we implemented an algorithm for identifying the lock/unlock pairs that define critical sections. As shown in Fig. 2 (left), a lock acquire operation (in Line 1) may correspond to multiple lock release operations along different program paths (in Line 4 and Line 10). Although there is a `lock_guard` template in C++11, the use of PThreads style mutexes in C++ code is still allowed. In general, our implementation of *CCmutator* consists of a combination of static program analysis and mutation generation.

In the *mutation* step, we allow the user to select an appropriate set of the lock-unlock pairs (computed during the *enumeration* step) on which we apply the mutation operator. For mutations that split, shift, expand, and shrink critical sections, the mutation step involves inserting or removing lock and unlock calls in proper locations with correct mutex pointers as the parameters. For example, when splitting a critical section, a new unlock call to the same mutex pointer is inserted after the lock call of the pair; this is also the case

for a new lock call inserted after the new unlock call.

Another difficulty in mutating concurrent programs is that, naively making small syntactic code changes, although it has worked well for mutating sequential programs, often lead to low-quality mutants—mutants that can be *killed* by almost every test run of the concurrent program. To generate high-quality mutants, the application of *partial* [18] and *higher order* [13] mutations is important, because it allows for very specific combinations of mutations to be applied to create potentially subtle [15] concurrency bugs. We have implemented both approaches in *CCmutator*. Partial mutation allows the user to select parts of the given code where mutation operators are to be applied rather than the entire code. Higher order mutation is the repeated application of a mutation operator to an already generated mutant. Together, they can produce mutants with more focused and complex combinations of code changes relative to first order mutants, and therefore allow the user to rigorously target specific areas of a code base.

IV. The List of Mutation Operators

We have implemented a comprehensive set of concurrency mutation operators, which not only include the ones in the literature (e.g., [6] for Java) but also new operators that are specific to PThreads and C++11. Notice that some of the concurrent Java mutation operators defined in [6] had no C/C++ equivalent, e.g., the Java `synchronized` keyword.

Table I shows the list of implemented mutation operators. These operators are useful in that they have the possibility to create bugs in concurrent programs. For example, swapping the lock order of one thread can introduce a deadlock.

Operations such as splitting/shifting critical sections requires user input (shift direction, shift amount, etc.). *CCmutator* is naive in that it will not ensure a mutant is faulty. The authors of [14] dealt with this problem when using mutation testing by removing mutants from their test suite that did not fail any tests, were malformed code or were killed by every test. LLVM ensures that *CCmutator* at least generates syntactically correct LLVM IR.

TABLE I: The list of concurrency related mutation operators. Note that some operators are valid only for PThreads or for C++11.

Name	Description	posix	c++
msem	Modify Permit Count in Semaphore	✓	X
mwait	Modify parameter in cond_timedwait()	✓	X
mcnt	Modify cond_timedwait() time value	✓	X
rmwait	Remove Call to cond_wait()	✓	✓
rmwait	Remove Call cond_timedwait()	✓	✓
swptw	Swap cond_timedwait() with cond_wait()	✓	X
rmsig	Remove Call cond_signal()	✓	X
rmsig	Remove Call to cond_broadcast()	✓	X
swpb	Swap cond_signal() with cond_broadcast()	✓	X
swps	Swap cond_broadcast() with cond_signal()	✓	X
rmjoin	Remove Call to join()	✓	✓
rmyld	Remove call to yield()	✓	X
repjn	Replace join() with sleep()	✓	✓
rmvol	Remove Volatile Keyword	✓	✓
swplck	Swap lock-unlock pairs	✓	✓
rmecs	Remove explicit critical section	✓	✓
shfecs	Shift explicit Critical Region	✓	✓
shkecs	Shrink explicit Critical Region	✓	✓
epdecs	Expand explicit Critical Region	✓	✓
spltecs	Split Critical Region	✓	✓
rmf	Remove memory fence	n/a	✓
mfe	Modify memory fence ordering constraint	n/a	✓
repsf	Replace single-thread sync fence with cross-thread	n/a	✓
repcf	Replace cross-thread sync fence with single-thread	n/a	✓
repal	Replace atomic load with non-atomic load	n/a	✓
mal	Modify atomic load ordering constraint	n/a	✓
repsl	Replace single-thread sync atomic load with cross-thread	n/a	✓
repcl	Replace cross-thread sync atomic load with single-thread	n/a	✓
relas	Replace atomic store with non-atomic store	n/a	✓
mas	Modify atomic store ordering constraint	n/a	✓
repss	Replace single-thread sync atomic store with cross-thread	n/a	✓
repcs	Replace cross-thread sync atomic store with single-thread	n/a	✓
mrmw	Modify atomic read-modify-write ordering constraint	n/a	✓
rsrm	Replace single-thread sync atomic read-modify-write with cross-thread	n/a	✓
rcrm	Replace cross-thread sync atomic read-modify-write with single-thread	n/a	✓
mcx	Modify compare exchange ordering constraint	n/a	✓
rscx	Replace single-thread sync compare exchange with cross-thread	n/a	✓
rccx	Replace cross-thread sync compare exchange with single-thread	n/a	✓

A. Mutex Locks

These operators work on explicit lock-unlock pairs, to remove, swap, shift, or split critical sections created by these pairs. For example, removing the lock-unlock operations from the code allows for the critical sections to be disregarded, potentially creating data-races or atomicity violations. Swapping allows for not only the wrong locks to be used, which creates data-races, but also the wrong order in acquiring locks, potentially causing deadlocks. Shifting the critical section allows for instructions to be added or removed from the top or bottom of a critical section. Splitting allows for a new lock-unlock pair to be inserted inside the original critical section. Both shifting and splitting may introduce data-races and atomicity violations on specific variables.

B. Condition Variables

Mutations on condition variables fall into two subcategories: the mutations of *signal/broadcast* calls and the mutations of *wait* calls. For the first subcategory, we can remove calls to *signal* and *broadcast*, or replace calls to *signal* with calls to *broadcast*, and vice versa. For the second subcategory, we can remove calls to *wait*, or replace them with calls to *timed wait*. We can also replace calls to *timed wait* with calls to *wait*, which may introduce deadlocks. These mutation operators work on individual calls to *wait*, *signal*, and *broadcast*. We can also generate mutants by simultaneously changing all operations over a given condition variable.

C. Thread Creation and Join

Replacing a call to *join* with a call to *sleep* allows for the program to appear to be working correctly—if the sleep time happens to be sufficient—since once the call to *sleep* returns, the thread that was being waited for will have finished. However, this may lead to intermittent program failures, for instance, when the computer has a heavy workload. Unlike POSIX threads C++11 threads need to be explicitly joined or detached; if a thread has not been joined or detached and its destructor is called then `std::terminate` is called, causing the entire program to crash. This lowers the effectiveness of simply removing or replacing a join-able C++11 thread, since the bug will be detected easily at runtime. To be more effective, we should replace the call to *join* with a call to *detach* and an optional call to *sleep*. This would create a similar effect as seen in POSIX threads.

D. C++11 Atomic Object

Mutations on C++11 specific *atomic* objects include removing the *atomic* keyword from the code, which effectively turns the atomic object into a regular object. The C++11 atomic keyword is typically used to implement high-performance *ad hoc* thread synchronizations, e.g., the ones that are frequently

used in systems code as well as high-performance concurrent data structures. Removing the atomic keyword will likely introduce subtle concurrency bugs that are specific to the newly defined *relaxed memory model*, which allows the compiler and runtime systems to reorder certain sequential instructions that are typically forbidden by the *sequential consistency* requirement. These mutation operators are important because they cover usages of the C++11 concurrency constructs for systems code.

E. Semaphores

Mutations to semaphores involve modifying the permit count which allows for counting semaphores to be converted to arbitrary resource counts, and binary semaphores to be converted to counting semaphores. This potentially inserts deadlocks and/or data races.

F. POSIX Yield

Mutations on *yield* can remove calls that yield the current thread to the operating system scheduler. These *yield* calls are not as explicit of synchronization methods such as semaphores but they can be used to prevent thread starvation. Therefore, these mutation operators can introduce performance bugs, which are a type of concurrency related defects that do not lead to program crash or hang, but may degrade the runtime performance.

V. Related Work

Argrawal et al. [2] implemented the first mutation testing systems for the C language. Subsequent tools such as *CSaw* [10] and *MiLu* [16] also provide mutation operators for *sequential* aspects of C programs. Bradbury et al. [6] proposed a set of concurrent Java mutation operators. Tools such as Paraμ [18] implemented these operators. However, we noticed a lack of similar operators or robust implementations for multithreaded C and C++ programs.

Early mutation generation tools were designed based on the *competent programmer* hypothesis, stating that buggy programs would only be a few keystrokes away from a correct program [9], [2], [17]. This is evident in their creation of operators such as convert logical AND to logical OR. However, Purushothaman and Perry [20] showed that in reality, 90% of post-release faults were complex and would require more than one change to the programs syntax to fix. This means that one should put more efforts on creating not a large number of first order mutants but high quality subtle faults [13].

CCmutator implements all the mutation operators as partial order mutations [18]. This allows the user to select specific combinations of occurrences of fault injection sites to produce a mutant. This partial order mutant can then have any number of other partial order mutation applied to it, allowing for the creation of very specific higher order mutants [13].

In [14] different concurrent coverage metrics were evaluated against each other using Java concurrent mutation testing to see which could find the most bugs among many randomly generated mutants. *CCmutator* has the potential to do the same for C/C++. MuTMuT [11] introduces techniques for speeding up mutant *execution* during mutation testing.

VI. Conclusion

We have presented the first concurrency related mutation generation tool for multithreaded C/C++ applications, targeting concurrency constructs in the popular POSIX threads and the newly standardized C++11 threads. Our LLVM based implementation of the tool, called *CCmutator*, is both scalable and robust in handling real C/C++ applications. We expect our tool to be quickly adopted by researchers and developers in the area of concurrent software testing and verification, which is a large and rapidly growing community.

Acknowledgments

This work is supported in part by the NSF grant CCF-1149454 and the ONR grant N00014-13-1-0527.

References

[1] Apple. LLVM compiler. https://developer.apple.com/technologies/tools/.
[2] Argrawal, DeMillo, Hathaway, Hsu, Krauser, Martin, Mathur, and Spafford. Design of mutant operators for the C programming language. Technical report, Purdue University, 1989.
[3] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.
[4] T. Bergan, O Anderson, J. Devietti, L. Ceze, and Grossman D. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.
[5] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. An empirical framework for comparing effectiveness of testing and property-based formal analysis. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 2–5, 2005.
[6] J.S. Bradbury, J.R. Cordy, and J. Dingel. Mutation operators for concurrent Java (J2SE 5.0). In *Workshop on Mutation Analysis*, 2006.
[7] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Design and Implementation*, pages 209–224, 2008.
[8] V. Chipounov and G. Candea. Enabling sophisticated analyses of x86 binaries with RevGen. In *Workshop on Dependable Systems and Networks*, 2011.
[9] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
[10] M. Ellims, D. Ince, and Marian Petre. The Csaw C mutation tool: Initial results. In *Testing: Academic and Industrial Conference Practice and Research Techniques*, pages 185–192, 2007.
[11] M. Gligoric, V. Jagannath, and D. Marinov. MuTMuT: Efficient exploration for mutation testing of multithreaded code. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 55–64, 2010.
[12] Google. Native client. https://developers.google.com/native-client/.
[13] M. Harman, Y. Jia, and W. Langdon. Strong higher order mutation-based test data generation. In *SIGSOFT FSE*, pages 212–222, 2011.
[14] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel. The impact of concurrent coverage metrics on testing effectiveness. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 232–241, 2013.
[15] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 249–258, 2008.
[16] Y. Jia and M. Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Academic Industrial Conference on Testing*, pages 94–98, 2008.
[17] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011.
[18] P. Madiraju and A.S. Namin. Paraμ - a partial and higher-order mutation tool with concurrency operators. In *Workshop on on Software Testing, Verification and Validation*, pages 351–356, 2011.
[19] NIST. Juliet Test Suites. http://samate.nist.gov/SRD/testsuite.php/.
[20] R. Purushothaman and D.E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, June 2005.
[21] Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.