UNIVERSITY OF **MOL**ISE
DEPARTMENT OF BIOSCIENCE AND TERRITORY

Project in Software Security

Verification of Lock-free Data Structure

Student: Daniele Albanese
Supervisor: Prof. Gennaro Parlato

MAY 2023

Contents

Listings

List of Tables

Abstract

Model checking is one of the most effective tools for verifying high-criticality programs, such as those used in security applications, automobiles, and medical devices. However, model checking can be computationally expensive, as it requires analyzing a large number of system states, so efficient algorithms and specialized tools are needed for the verification of large-scale systems. The goal of this project is to design, develop, and implement a new verification system for lock-free data structures for concurrent C programs. This system will be complemented by a BMC-based concurrent program analysis tool called Lazy-CSeq which has a more famous and robust tool called CBMC as its analysis backend. The search was conducted on the GitHub platform using Lock-Free Queue (LFQ) and Data Structures (Data Structures) as keywords. At the end of the analysis, the generated trace was found to be valid, indicating that the analyzed data structure does indeed contain a bug.

1 Introduction

1.1 Application Context

Multithreading is a widely used structuring technique for modern software. Programmers use multiple threads of control for a variety of reasons: (i) to build responsive servers that interact with multiple clients, (ii) to run computations in parallel on a multiprocessor for performance, and (iii) as a structuring mechanism for implementing rich user interfaces. In general, threads are useful whenever the software needs to manage a set of tasks with varying interaction latencies, exploit multiple physical resources, or execute largely independent tasks in response to multiple external events. This is because the clock speed of processors is no longer easily increased, but the same cannot be said of the number of cores a processor can hold. Therefore, to achieve higher performance software must necessarily be concurrent.

1.2 Motivation

For multithreaded programs, analysis can be a challenging task. The complication lies in characterizing the effect of interactions between threads. The obvious approach of analyzing all possible interleavings of statements from parallel threads fails because of the exponential analysis time involved. A central challenge is to develop efficient abstractions and analysis that capture the effect of each thread's actions on other parallel threads. This becomes even more complicated with non-blocking data structures. These structures allow several threads to access the same memory cell simultaneously, however, as long as read operations are involved there are no particular problems since the data is untouched; but, the moment write operations are to be implemented, things change. Because of these complex interactions, concurrent programs often contain bugs that are difficult to find, reproduce and fix. Existing automatic bug-finding techniques and tools are not effective when it comes to concurrent programs. For techniques that explicitly analyze executions, finding rare bugs is like looking for a needle in a haystack.

1.3 Objective

The goal of this project, is to implement a new verification system for lock-free data structures for concurrent C programs. This system will be complemented by a BMC-based concurrent program analysis tool called Lazy-CSeq which has a more famous and robust tool called CBMC as its analysis backend. Thus, the goal will be to design, develop and implement a new lock-free data structure verification system using the Lazy-CSeq tool by going to instrument a concurrent C program so that CBMC can create for us all the possible configurations that, the created C program could generate. This saves considerable time and resources as we will be going to generate all the possible real configurations that the program could generate.

2 Background

2.1 Non-determinism

Non-determinism is a property of some systems in which the execution of the system can produce different outcomes every time it is run, even with the same inputs and the same initial state. This occurs because the system does not have a predetermined sequence of actions or because some events can happen randomly or arbitrarily. For example, a non-deterministic program can have several paths of execution, each of which produces a different outcome, depending on random factors or external input. Non-determinism can make system verification more complex since all possible system executions, including those that produce anomalous or unexpected results, must be considered. However, in some cases, non-determinism can be used to the system's advantage, for instance, to increase the system's security or robustness.

2.2 Model Checking

Model checking is a formal technique for verifying the correctness of a system, such as hardware or software, by analyzing all possible system states against a specific property. In practice, model checking performs an exhaustive analysis of the system, checking whether it satisfies a given property specification such as never experiencing a deadlock or violating security. Model checking is used in various fields, including computer science, software engineering, automation, and electronics. However, model checking can be computationally expensive, as it requires analyzing a large number of system states, so efficient algorithms and specialized tools are needed for verifying large, complex systems.

2.3 Bounded Model Checking

Bounded model checking is a formal verification technique that performs an exhaustive analysis of a system but limits the analysis to only the first n states of the system. In other words, bounded model checking limits the search for errors to a fixed number of execution steps of the system. This approach can be useful for verifying large-scale systems, where exhaustive analysis of all states is impractical due to computational complexity. Bounded model checking is often used for verifying communication protocols, hardware, and software. However, using a limited search dimension ca lead to false negatives, where errors are overlooked because they are beyond the preset limit. Therefore, it is necessary to choose the search dimension wisely to ensure that relevant errors are still detected.

2.4 CBMC

CBMC is a formal verification tool used for verifying C programs based on model-checking technology. CBMC analyzes the program's source code and generates a formal representation, called a model, that represents all possible program executions. Model-checking is then used to verify that the program satisfies certain properties specified by the user, such as functional correctness or prevention of security vulnerabilities like buffer overflows or memory leaks. CBMC is a highly effective tool for verifying high-criticality programs, such as those used in security applications, automobiles, and medical devices.

2.5 Lazy-CSeq

Lazy-CSeq is an automated bug-finding tool based on Bounded Model Checking (BMC) and Sequentialization. Lazy-CSeq first translates a multi-threaded C program into a nondeterministic sequential C program (called *sequentialized* program) that preserves reachability for all round-robin schedules with a given bound on the number of rounds. It then re-uses existing high-performance BMC tools as backends, such as CBMC[6], for the sequential verification problem. This translation is carefully designed to introduce very small memory overheads and very few sources of non-determinism, so that it produces tight SAT/SMT formulae, and is thus very effective in practice.

2.6 Lock-free Data Structure

Lock-free data structures are data structures designed to allow concurrent access by multiple threads without the use of locks or other explicit synchronization mechanisms. This means that each thread can perform operations on the data structure independently of other threads, without the need to acquire a mutual exclusion lock that would limit other threads' access to the same data structure. Instead, lock-free data structures use techniques such as atomic pointers, atomic operations, and random store mechanisms to ensure that the operations performed by threads are coherent with each other and that no conflicts occur between threads. Lock-free data structures are used primarily in applications with a high degree of parallelism, such as data processing systems, multimedia processing systems, and games. However, designing and implementing lock-free data structures requires a good understanding of atomic operations, shared memory, and thread synchronization.

3 Approach

3.1 Library Pre-processing

The first phase of our approach involves the initial pre-processing of the library. This phase consists of replacing any assembly code[1] (Listing 1) with semantically equivalent C code (Listing 2) and removing unnecessary code for analysis. This phase is necessary because the Lazy-CSeq tool is still under development so some limitations must be compensated manually.

```
.section .data
hello_world:
    .ascii "Hello, World!\n"
    hello_world_len = . - hello_world
```

[1]Assembly code is a low-level language that is used to directly program computer hardware. It is used within C code to directly access computer hardware or to write particularly efficient code to perform complex operations quickly. Assembly code has no intermediate layers between the code and the computer hardware as is the case with code written in a high-level language such as C and is, therefore, faster and more efficient. However, assembly code is much more complex to write than C code and requires a thorough knowledge of the computer hardware on which it is to be run.

```

.section .text
.globl _start

_start:
    movl $4, %eax
    movl $1, %ebx
    movl $hello_world, %ecx
    movl $hello_world_len, %edx
    int $0x80

    movl $1, %eax
    xorl %ebx, %ebx
    int $0x80
```

Listing 1: Assembly - Hello World

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Listing 2: C - Hello World

By the term "*unnecessary code*", what is meant is conditioning code that depends on the characteristics of the system in which the program is run (Listing 3). A practical example is the presence of configurations to run a particular program on Windows systems when running on a Unix system. Another example of "*unnecessary code*" is any comment within the code[2].

```c
#if defined __GNUC__

...

#else
    #include <Windows.h>
    #define ATOMIC_SUB(x,y) InterlockedExchangeAddNoFence(x, -y)
    #define ATOMIC_SUB64(x,y) InterlockedExchangeAddNoFence64(x, -y)
    #define ATOMIC_ADD InterlockedExchangeAddNoFence
    #define ATOMIC_ADD64 InterlockedExchangeAddNoFence64
    #ifdef _WIN64
        #define mb() MemoryBarrier()
        #define lmb() LoadFence()
        #define smb() StoreFence()
        inline bool __CAS(LONG64 volatile *x, LONG64 y, LONG64 z) {
            return InterlockedCompareExchangeNoFence64(x, z, y) == y;
        }
```

[2]These changes were necessary because, in testing, they highlighted some problems related to the Lazy-CSeq tool.

```
        #define CAS(x,y,z)  __CAS((LONG64 volatile *)x, (LONG64)y, (LONG64)z)
    #else
        #define mb() asm mfence
        #define lmb() asm lfence
        #define smb() asm sfence
        inline bool __CAS(LONG volatile *x, LONG y, LONG z) {
            return InterlockedCompareExchangeNoFence(x, z, y) == y;
        }
        #define CAS(x,y,z)  __CAS((LONG volatile *)x, (LONG)y, (LONG)z)
    #endif

    // thread
    #include <windows.h>
    #define THREAD_WAIT(x) WaitForSingleObject(x, INFINITE);
    #define THREAD_ID GetCurrentThreadId
    #define THREAD_FN WORD WINAPI
    #define THREAD_YIELD SwitchToThread
    #define THREAD_TOKEN HANDLE
#endif
```

Listing 3: Unnecessary Code

3.2 Concurrent C Program

The second phase is to create a concurrent C program that makes use of a lock-free data structure[3]. Once created, this program will need to be instrumented in such a way that it can be analyzed using Lazy-CSeq. Instrumentation of the code involves adding the prefix **`__VERIFIER_atomic_`** to the functions that operate on the lock-free data structure which is used in CBMC to indicate that a function represents an atomic operation[4] (e.g. Listing 4).

```
void __VERIFIER_atomic_enq(int *num)
{
    lfq_enqueue(&ctx, (void *)num);
}

int *__VERIFIER_atomic_deq()
{
    return (int *)lfq_dequeue(&ctx);
}
```

Listing 4: Atomic Functions

Next, in order to extract the configuration generated by CBMC, it will be necessary to implement a function that prints the data structure by going to save the contents in a variable (e.g. Listing 5).

[3]As mentioned in Section 3.1 the Lazy-CSeq tool is still under development, so one thing to do during this second phase is to import the implementation of the data structure under analysis within the created program.

[4]Atomic operations are used to ensure that no interference occurs between concurrent processes accessing the same shared data so that all operations on that data are performed sequentially and uninterrupted.

```c
void __VERIFIER_atomic_print()
{
    int i = 0;

    while ((ret = __VERIFIER_atomic_deq()) != 0)
    {
        var_to_get[i] = *ret;

        i++;
    }
}
```

Listing 5: Print Function

Finally, the comment **`//CS_ASSUME`** and the call of the function **`assert()`**[5], having 0 as the input parameter, will have to be added at the end of the program main function, before the **`return 0`** (e.g. Listing 6).

```c
int main(int argc, char const *argv[])
{
    lfq_init(&ctx, 1);

    pthread_t thread_one, thread_two;

    pthread_create(&thread_one, NULL, thread_func_one, NULL);
    pthread_create(&thread_two, NULL, thread_func_two, NULL);

    pthread_join(thread_one, NULL);
    pthread_join(thread_two, NULL);

    __VERIFIER_atomic_print();

    //CS_ASSUME
    assert(0);

    return 0;
}
```

Listing 6: main.c

3.3 Instances Generation

Once the concurrent C program is completed, it should be analyzed by Lazy-CSeq. Lazy-CSeq is a tool that uses CBMC as its backend. CBMC is a model checker for sequential C programs so it is not suitable for analyzing concurrent C programs. Lazy-CSeq translates a concurrent C program into a nondeterministic sequential C program, called **`sequentialized`** (Listing 7), which preserves reachability.

[5]The **`assert()`** function is a standard C library function that is used to check for assumptions in code during program execution. The function takes a Boolean expression as an argument and checks that it is true. If the expression is false, the **`assert()`** function stops program execution and generates an error message.

```
        ./lazycseq.py \
            -i {file_path} \
            --unwind {unwind} \
            --rounds {rounds} \
            --seq \
```

Listing 7: Instance Generation

In this way, a concurrent C program is made sequenced so that it can be analyzed by the CBMC backend. Then, once the sequenced program is obtained, it is analyzed by CBMC which returns a trace containing a counterexample i.e., a possible execution configuration of the program (Listing 8).

```
        ./cbmc \
            {file_path} \
            --unwind {unwind} \
            --stop-on-fail \
            --object-bits {object_bits} \
            --trace \
```

Listing 8: CBMC Analysis

The generated configuration will have to be inserted in the place of the **`//CS_ASSUME`** comment so as to indicate to CBMC that that particular configuration should be considered invalid in the next run (Listing 9)[6]. In this way, we will have a new concurrent C program that can no longer generate the previously found configuration. This process will have to be repeated until CBMC provides no more counterexamples, that is, until CBMC has generated all the possible configurations that that concurrent C program can generate.

```
    __VERIFIER_assume(!(var_to_get[0] == 3 && ... && var_to_get[7] == -1));
```

Listing 9: Example of __VERIFIER_assume()

3.4 Instances Validity

Having completed the generation of all possible instances, which the program can generate, it is time to check what happens by removing this constraint. For this reason, the last version of the generated program, that is, the program containing all possible configurations that the program can generate; must be stripped of the **`__VERIFIER_atomic_`** prefix. Doing so will result in a new version of the program that will no longer have the constraint on the atomicity of operations. What you want to check is that, by removing the constraint on the atomicity of operations, the program will not be able to generate a new configuration that is different from those previously generated. Then, having created this new version of the program, we will proceed with the same procedure as described in Section 3.3 i.e., the program will be analyzed by Lazy-CSeq, which will create a sequenced version of the program, which will be analyzed by CBMC; which can generate two types of traces: (i) a trace that will not contain a counterexample, this will mean that the program is unable to generate a new configuration, thus that the data structure is correct; (ii) a trace that will contain a counterexample, this

[6] The **`__VERIFIER_assume()`** function is a function used in CBMC to specify assumptions about the state of the system during model checking of a C program. The function takes a Boolean expression as an argument and assumes that this expression is true during model checking.

does not necessarily mean that the data structure is incorrect, but there may have also been a problem related to the instrumentation of the program or a problem related to Lazy-CSeq or related to CBMC. Therefore, in order to give an answer, it will be necessary to analyze the generated trace by going to check whether the counterexample is related to the data structure or not.

4 Experimental Design

To speed up the more iterative operations, such as generating instances (Section 3.3) and validating instances (Section 3.4), Python scripts were made that can be found in the GitHub repository of the project [2]. A machine with the following characteristics was used to run the experiments: (i) Arch Linux operating system having kernel version 6.1.26-1-lts, (ii) AMD Ryzen 9 5900x CPU @ 4.30GHz, (iii) 64GB RAM @ 3200MHz, (iv) Samsung 980 PRO SSD. Regarding lock-free data structures to be used, a search was made for existing open-source implementations. The search was conducted on the GitHub platform using lock-free and data structures as keywords. The search revealed two projects that implement an LFQ (Lock-Free Queue). Both projects were developed in C and were published on GitHub. The projects are as follows: (i) darkautism/lfqueue [5] and (ii) Taymindis/lfqueue [14]. Unfortunately, due to time constraints, only the first project could be analyzed.

5 Results

This section contains the results obtained using the approach proposed in Section 3 and using the resources described in Section 4.

The following results were obtained by analyzing the darkautism/lfqueue [5] library that implements a lock-free queue in C. As described in Section 4 the step of generating instances (Section 3.3) and validating instances (Section 3.4) were automated via a Python script. This phase took more than 17 hours to generate 44 instances and 50 minutes to validate the instances (more details on individual instances can be viewed in Table 1 regarding the Instance Generation phase and Table 2 regarding the Instance Validity). As described in Section 3.4, the possible results can be are two i.e.; the CBMC trace does not contain a counterexample, so we could say that the data structure is correct or the trace contains a counterexample, so further verification is needed to understand the nature of the problem. The result obtained from our analysis is of the second type. At this point, a manual analysis of the CBMC trace containing the counterexample was performed to check its validity. At the end of the analysis, the generated trace was found to be valid, indicating that the analyzed data structure does indeed contain a bug.

Instance	Runtime Symex	Runtime Convert SSA	Runtime Solver	Runtime decision	Variables	Clauses
#1	37.316s	18.4215s	179.17s	197.646s	22956789	119453007
#2	37.4513s	18.3649s	263.678s	282.094s	22962506	119476356
#3	37.0331s	18.4639s	253.064s	271.579s	22962992	119478751
#4	36.3883s	18.3831s	316.652s	335.087s	22963612	119482202
#5	39.896s	19.106s	184.134s	203.293s	22964097	119484501
#6	42.0034s	19.0145s	298.72s	317.787s	22964577	119486785
#7	43.6328s	19.317s	335.834s	355.206s	22965055	119488943
#8	40.323s	18.4773s	116.612s	135.14s	22965547	119491503
#9	43.133s	19.6269s	713.662s	733.352s	22966028	119493670
#10	39.0637s	19.2422s	311.209s	330.506s	22966504	119496062
#11	38.2072s	18.4021s	354.417s	372.87s	22966978	119497947
#12	38.5549s	18.8739s	733.011s	751.937s	22967612	119501105
#13	39.1454s	18.9891s	594.139s	613.181s	22968116	119503828
#14	37.5005s	18.8218s	521.431s	540.306s	22968607	119506272
#15	38.9249s	18.3732s	516.822s	535.248s	22969092	119508458
#16	37.9425s	18.4895s	313.739s	332.28s	22969579	119510650
#17	37.6913s	18.8775s	350.343s	369.271s	22970067	119512845
#18	38.0609s	18.689s	262.712s	281.453s	22970550	119515145
#19	37.1418s	18.5445s	296.213s	314.809s	22971032	119517322
#20	42.5768s	19.5839s	369.349s	388.994s	22971519	119519514
#21	41.8351s	19.3899s	245.252s	264.698s	22972007	119521709
#22	40.6502s	19.0234s	755.93s	775.007s	22972490	119523889
#23	42.8538s	19.2361s	171.261s	190.56s	22972975	119526075
#24	38.7522s	19.1375s	800.327s	819.52s	22973462	119528267
#25	38.2171s	18.912s	150.552s	169.516s	22973950	119530462
#26	36.931s	18.4538s	855.263s	873.77s	22974421	119532606
#27	36.4661s	18.3608s	946.165s	964.585s	22974895	119534377
#28	41.3794s	19.3435s	406.665s	426.061s	22975643	119537994
#29	55.3501s	22.8374s	1061.59s	1084.48s	22976138	119540217
#30	37.7385s	18.483s	1297.32s	1315.85s	22976628	119542425
#31	38.1516s	18.8058s	1040.19s	1059.05s	22977120	119544639
#32	38.8218s	19.0347s	711.383s	730.471s	22977614	119546859
#33	38.0289s	18.9566s	1040.53s	1059.54s	22978109	119549082
#34	37.5874s	18.3521s	1258.72s	1277.13s	22978595	119551278
#35	36.547s	18.6193s	164.213s	182.884s	22979084	119553483
#36	36.8681s	18.5379s	690.274s	708.863s	22979578	119555703
#37	37.0454s	18.6009s	1110.19s	1128.84s	22980073	119557926
#38	36.6847s	18.5105s	522.96s	541.522s	22980563	119560134
#39	36.8937s	18.5726s	1680.55s	1699.18s	22981055	119562348
#40	37.2777s	18.6876s	1065.73s	1084.47s	22981549	119564568
#41	37.101s	18.3677s	1656.79s	1675.21s	22982044	119566791
#42	36.1969s	18.5384s	1784.64s	1803.23s	22982526	119568975
#43	36.1633s	18.3661s	1055.41s	1073.83s	22983012	119571171
#44	36.5275s	19.0609s	1746.64s	1765.76s	22983506	119573391

Table 1: Atomic Instances Analysis

Instance	Runtime Symex	Runtime Convert SSA	Runtime Solver	Runtime decision	Variables	Clauses
#1	669.266s	25.9946s	1088.52s	1114.58s	27658777	143988075

Table 2: No Atomic Instances Analysis

6 Threats to Validity

The potential threats to the validity of our results fall into the internal and external categories.

6.1 Internal Validity

The analysis of the concurrent program is a very complex activity and the possible errors during this activity are many; first and foremost is human error.

6.2 External Validity

The proposed test case was effective in that it showed that the proposed approach is capable of finding bugs within concurrent programs. However, not enough tests were conducted to be able to say that the proposed approach is a consistently good approach for analyzing concurrent programs.

7 Conclusion

The analysis of concurrent programs is a difficult problem. The creation of automatic approaches is as important as it is difficult. In this paper, a semi-automatic approach for the analysis of concurrent programs was presented. The approach is based on a symbolic model-checking technique, which allows the analysis of concurrent programs with an arbitrary number of threads. The use of Lazy-CSeq allows the analysis of concurrent programs using a robust tool such as CBMC. Symbolic model-checking techniques are exploited to generate a set of execution traces covering all possible behaviors of the program using functions. Once all traces are generated, one moves on to checking the concurrent program by checking that it is no longer able to generate new execution traces. The approach has shown good results on some examples of concurrent programs. In the future, to test the robustness of the approach, it is necessary to test the approach on more examples of concurrent programs as well as concurrent libraries. In addition, the approach needs to be tested on concurrent programs that use more complex data structures, such as linked lists.

References

[1] Michael Pradel Adrian Nistor and Darko Marinov. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. 2012.

[2] Daniele Albanese. Verification of lock-free data structure, 2022. https://github.com/dj-d/VoLFDS.

[3] Shan Lu Ankit Choudhary and Michael Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. 2017.

[4] Salvatore La Torre Bernd Fischer and Gennaro Parlato. Verismart 2.0: Swarm-based bug-finding for multi-threaded programs with lazy-cseq. 2019.

[5] darkautism. lfqueue. https://github.com/darkautism/lfqueue.

[6] D Kroening E Clarke and F Lerda. A tool for checking ansi-c programs. *TACAS 2004: 168-176*.

[7] Michael Emmi and Constantin Enea. Violat: Generating tests of observational refinement for concurrent objects. 2019.

[8] Murali Krishana Ramanathan Malavika Samak and Suresh Jagannathan. Synthesizing racy tests. 2015.

[9] Murali Krishana Ramanathan Malavika Samak and Omer Tripp. Directed synthesis of failing concurrent executions. 2016.

[10] Michael Pradel and Thomas R. Gross. Fully automatic and precise detection of thread safety violations. 2012.

[11] Malavika Samak and Murali Krishna Ramanathan. Multithreaded test synthesis for deadlock detection. 2014.

[12] Malavika Samak and Murali Krishna Ramanathan. Synthesizing tests for detecting atomicity violations. 2015.

[13] Sebastian Steenbuck and Gordon Fraser. Generating unit tests for concurrent classes. 2013.

[14] Taymindis. lfqueue. https://github.com/Taymindis/lfqueue.

[15] Valerio Terragni and Shing-Chi Cheung. Coverage-driven test code generation for concurrent classes. 2016.