Enabling Automatic Verification of Concurrent Data Structures

Master's Thesis Project Description
Christof Leutenegger
Supervisor: Prof. Dr. Michalis Kokologiannakis
15.10.2024

1 – Introduction

Writing correct concurrent software remains notoriously difficult, with many bugs occurring due to improperly synchronized code. To make matters worse, modern platforms employ weak memory consistency, i.e., the instructions of each thread might be reordered by the compiler or the CPU.

One way to mitigate this is to use thread-safe data structures (for example queues and stacks) in concurrent libraries. This enables programmers to synchronize their threads without worrying about correctness. However, the libraries used often assume sequential consistency, and their correctness within weak memory models is usually argued informally.

This thesis aims to argue about this interaction in more formal terms by verifying well-known concurrent libraries like `ck` and `libcds` using model checking. To do this, we need to wrap the primitives of the libraries in code that is executable by the model checker.

2 – Background

In the following subsection, we provide further background about memory models, the concurrent libraries, the model checker, and how we intend to make the concurrent libraries interact with the model checker correctly.

2.1 Weak Memory Models

Memory models define the interaction between different threads of a program. For a programmer, a memory model specifies how memory accesses are ordered or what value is to be expected on a load operation. Memory models exist at each level between programmer and processor, from the high-level programming language down to the processor hardware.

Specifying memory models is difficult (e.g., Java needed multiple iterations [7], [8], [9]) and they are complicated to understand. The C/C++ memory model was first defined in C11 and C++11 (and fixed with RC11 [4]). The most intuitive memory model for programmers is sequential consistency (SC) [3], which ensures that all instructions follow a global total order. An access mode that provides SC ordering is also the default in C++11 but can be relaxed for increased performance. If we relax SC (e.g., to `relaxed` or `release-acquire`) we have to assure correctness with the help of different relations (synchronize-with, happens-before, program-order, modification-order) defined on the execution

graph without a global total order. Doing so correctly is only a viable option for experts.

2.2 Concurrent Libraries

To make writing concurrent code viable for non-experts there exist multiple concurrent libraries that provide thread-safe, standard data structures like stacks, queues, maps, and trees. To achieve better performance, many data structure implementations use several strategies such as fine-grained synchronization (e.g., optimistic traversals, lazy updates, helping), contention reduction (e.g., elimination), and efficient memory reclamation (e.g., hazard pointers or RCU). Ensuring these techniques and data structures are implemented correctly in weak memory models is an open problem we try to solve with GenMC.

Two such libraries are `ck` [5] and `libcds` [6].

`ck` is a C library that provides concurrency primitives, safe memory reclamation, data structures, and synchronization primitives. The library is written on C99 which did not include a memory model. Instead, it uses the memory model of the underlying hardware directly and provides assembly code for various supported ISAs.

`libcds` is a C++ library that is written on C++11. Therefore, we can use compilers like `gcc` and `clang` to port the library to different platforms.

2.3 GenMC

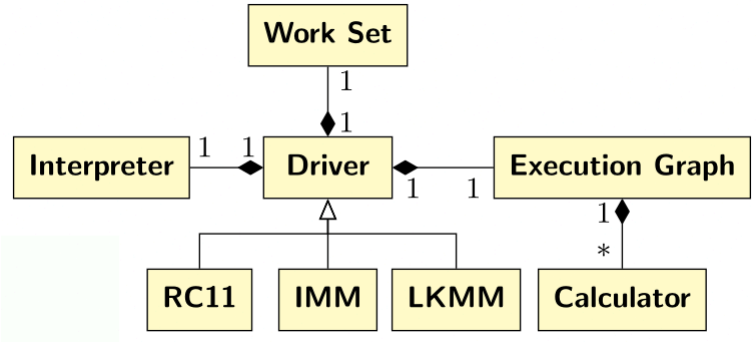

Fig. 1: components of GenMC (from [1])

GenMC [1] is a state-of-the-art stateless model checker for concurrent code. For this thesis, two properties of GenMC are especially important.

- *Generality:* GenMC is adaptable for different languages and memory models. This is achieved as the stateless model checker algorithm [2] has little requirements on the used memory models, allowing many different memory models to be used from SC [3] to RC11 [4]. Single components (see Fig. 1) can be swapped or adapted to different languages and memory models.
- *Extensibility:* GenMC is easily extensible for adding new models, synchronization primitives, and performance improvements. This enables us, for example, to add missing primitives that a library uses.

The interpreter shipped within GenMC takes LLVM IR as input and interacts with the driver to verify the code. With `clang` we can compile `libcds` to LLVM IR directly, simplifying the porting of the library. Further, we have to extend the interpreter to accommodate missing LLVM IR operations. Then, we will provide alternative definitions for functions that the libraries use that are supported by the model checker (e.g., thread creation, locking, memory reclamation). Finally, we will prove functional correctness for the various data structures offered by these libraries using the model checker, and perhaps tinker with the model checker to enhance its scalability.

For `ck`, we either need to add additional scaffolding for the environment the library assumes or port it in its entirety to C11.

3 – Goals

The goal of this thesis is to verify one of the libraries `ck` or `libcds` with GenMC. This should enable this thesis to serve as a standardized benchmark for similar verification attempts in the future. Additionally, the interpreter may not support all operations the libraries use, in which case it will need to be extended to accommodate for that.

Other goals include the extension of GenMC for performance and support of non-determinism.

3.1 Core Goals

1. **Verify `TreiberStack` [10] and `MSQueue` [11]**
   In the first step, we will verify the correctness of two well-known concurrent data structures: the Treiber stack and the Michael-Scott queue. For that, we have to decide on the library we verify and familiarize ourselves with GenMC.
2. **Add missing LLVM IR operations and scaffolding**
   To verify all data structures, some parts are most likely missing in the GenMC interpreter (e.g., LLVM IR operations). Further, we need to add scaffolding for the libraries such that they can use the functions provided by the model checker. This is especially important for the `ck` library, which we would have to port to C11.
3. **Verify the concurrent data structures of one library**
   Apply what we did in the core goals 1 & 2 to the rest of the library. We will add data structures one at a time and then finally prove functional correctness for the entire library.
4. **Evaluation**
   Finally, we evaluate our verification of the library. This includes collecting and evaluating verification time measurements, reasoning about the introduced scaffolding, and our experience using GenMC for validating the library.

3.2 Extension Goals

1. **Speedup of GenMC verification**
   One way to improve GenMC is to shorten the verification time. One possibility is to add annotation that marks linearization points in the code, this would ensure that unnecessary orderings can be skipped.
2. **Add support for non-determinism to GenMC**
   Everything in GenMC is deterministic so far. To support the verification of as many programs as possible without introducing changes, it makes sense to add non-determinism or to simulate it through symbolic execution.

References

1. Kokologiannakis, Michalis, and Viktor Vafeiadis. "GenMC: A Model Checker for Weak Memory Models." In *Computer Aided Verification*, edited by Alexandra Silva and K. Rustan M. Leino, 12759:427–40. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021. https://doi.org/10.1007/978-3-030-81685-8_20.
2. Kokologiannakis, Michalis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. "Truly Stateless, Optimal Dynamic Partial Order Reduction." *Replication Package for "Truly Stateless, Optimal Dynamic Partial Order Reduction"* 6, no. POPL (January 12, 2022): 49:1-49:28. https://doi.org/10.1145/3498711.
3. Lamport, Leslie. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs." *IEEE Transactions on Computers* C–28, no. 9 (September 1979): 690–91. https://doi.org/10.1109/TC.1979.1675439.
4. Lahav, Ori, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. "Repairing Sequential Consistency in C/C++11." In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 618–32. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017. https://doi.org/10.1145/3062341.3062352.
5. ck. "Concurrrency Kit." Accessed October 13, 2024. http://concurrencykit.org/.
6. "Khizmax/Libcds: A C++ Library of Concurrent Data Structures." Accessed October 13, 2024. https://github.com/khizmax/libcds.
7. Pugh, William. "The Java Memory Model Is Fatally Flawed." *Concurrency: Practice and Experience* 12, no. 6 (May 2000): 445–55. https://doi.org/10.1002/1096-9128(200005)12:6%3C445::AID-CPE484%3E3.0.CO;2-A.
8. Manson, Jeremy, William Pugh, and Sarita V. Adve. "The Java Memory Model." In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 378–91. POPL '05. New York, NY, USA: Association for Computing Machinery, 2005. https://doi.org/10.1145/1040305.1040336.
9. Aspinall, David, and Jaroslav Ševčík. "Java Memory Model Examples: Good, Bad and Ugly." In *Proceedings of Verification and Analysis of Multi-Threaded Java-Like Programs (VAMP 2007)*, 2007. https://www.research.ed.ac.uk/en/publications/java-memory-model-examples-good-bad-and-ugly.

10. Treiber, R Kent. “Systems Programming: Coping with Parallelism.” *Technical Report RJ 5118, IBM Almaden Research Center*, April 1986.

11. Scott, Michael, and Maged Michael. “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms,” 1995. https://urresearch.rochester.edu/institutionalPublicationPublicView.action?institutionalItemId=344.