Verifying weak memory concurrent data structure implementations

Version of May 6, 2024

Casper Henkes

Verifying weak memory concurrent data structure implementations

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Casper Henkes
born in Den Haag, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Verifying weak memory concurrent data structure implementations

Author: Casper Henkes
Student id: 4569555

Abstract

From formal hardware models to programming language implementations concurrency is everywhere. While there has been a lot of work done on verifying concurrent systems a large part of it is focused on sequential consistency (SC). In practice, it is more common to encounter weak memory models for which the techniques developed for SC do not work. There exists previous research on verifying weak memory concurrent programs, however, this work is often limited in scope and is often difficult to understand and apply to a broader context. This thesis gives a general approach for calculating higher-level relations between function calls in a weak memory context that work for weak memory concurrent mutex, stack, and queue data structures. These relations allow us to abstract away implementation details for easier reasoning about program behaviour. These function-level relations and data structure models defined using them are implemented in a stateless model checker and used to verify several existing mutex, stack, and queue implementations.

Thesis Committee:

Chair: Prof. dr. A. van Deursen, Faculty EEMCS, TU Delft
Committee Member: Dr. S. Chakraborty, Faculty EEMCS, TU Delft
Committee Member: Dr. C.B. Poulsen, Faculty EEMCS, TU Delft

Chapter 1

Introduction

Concurrency is one of the default ways to increase program performance in the current technological landscape. From servers handling many clients at the same time to local systems dividing tasks between different software or hardware threads. Local concurrency, which we will be focusing on in this thesis, operates under the shared memory model where multiple threads communicate through loading data from and storing data in memory accessible by both threads. A basic model for shared memory concurrency is interleaving. Interleaving allows for any possible execution that can be generated by arbitrarily interleaving the operations of all threads that constitute the program. The model that only allows interleaving executions is called sequential consistency (SC).

It turns out that SC is quite strict and, for performance reasons, many real-life applications use weaker models. These models collectively called weak-memory models allow for more behaviour than only having interleaving threads would allow. Because of this extra behaviour, previous research-proven under SC usually does not translate well to weak-memory contexts. This paper focuses on a weak-memory context as both modern hardware [2, 20, 22] and programming languages [3, 4, 9, 17] models usually operate under weak memory assumptions.

There is work on weak memory concurrency, but that work usually focuses on properties and relations found between instruction-level events, like *load* and *store* events. These relations are synthesised in graphs that detail the execution of the program called an execution graph. These execution graphs and low-level relations are powerful tools for proofs and verification, but reasoning with them can be difficult and counter-intuitive.

To simplify the reasoning about interactions between different parts of the code it helps to formulate the concurrent program as layers of abstractions. This approach is also used in larger non-concurrent systems. Abstracting away some implementation details helps make formulations about problems cleaner and easier to understand.

The main idea of this thesis is to show that such a layered approach works in practice for weak memory concurrent systems. We use lower instruction-level relations to create a higher abstraction layer of function-level relations that we can then use to verify properties of the code under test without worrying about the implementation details of the underlying code. To show that this approach works, we develop a stateless model checker for C and LLVM implementations of weak-memory concurrent programs and use it to calculate these function-level relations. Then these function-level relations are used to verify different implementations of weak memory concurrent data structures.

The proposed tool verifies implementations by using test programs. These test programs consist of two separate parts, the implementation of the data structure and a driver program. The implementation consists of the actual code implementation of the data structure. The driver program can be seen as the test scenario, it is a program that uses the data structure and defines how many threads there will be and what thread will call what functions with

what values. The stateless model checker then explores all possible executions of the implementation, and a custom verification algorithm verifies that the executions adhere to the chosen data structure model.

This research proposes a modified model checker to verify the behaviour of weak memory concurrent data structure implementations. This paper shows that the tool can correctly construct function-level relations and use them to verify concurrent mutex, stack, and queue data structure implementations.

The main contributions of the research are as follows. First, the thesis defines a practical way to translate instruction-level properties into function-level properties. Second, it provides an open-source tool that can verify several higher-level properties on multiple data structures.

The rest of the paper follows the following structure. The background goes into more detail on shared memory concurrency and memory models, before explaining how weak memory programs can be verified. The main ideas explain the contributions in more detail and give a broad overview of the techniques used to construct function-level relations and verify data structure models. The technical details chapter explains the inner workings of the algorithms used. The evaluation shows the performance of the proposed tool on a variety of implementations. The discussion goes over optimally, soundness, the precision of the algorithms, and any strong points and limitations of the research. Lastly, we sum up the most important parts and describe potential future work.

Chapter 2

Background

2.1 Shared memory concurrency

Shared memory concurrency is a popular way to model concurrent behaviour on programs running on local systems. In this model, multiple threads communicate by storing and loading data to and from a shared memory. The simplest way to model shared memory concurrency is with interleaving threads. The memory model you get when the only allowed executions are those that interleaving threads can generate is called sequential consistency (SC). However, there are weaker memory models often used in practice called weak memory models. The rest of this section first explains the idea behind weak memory concurrency before explaining weak memory models and execution graphs.

Weak-memory concurrency SC [15] has been covered widely in research, but it turns out that it is not always applicable in practice. In practice, it is common for programs to behave in ways impossible under SC. This behaviour is called weak memory behaviour. Weak memory behaviour typically stems from allowing some reordering of load and store operations that take place on different memory addresses. Allowing this reordering is interesting in practice as it enables compiler and hardware optimizations that can drastically speed up the performance of programs.

Weak-memory example See figure 2.1 for an example program. Under interleaving it is possible to get the following outcomes: either $a = 1 \wedge b = 1$, $a = 1 \wedge b = 0$, or $a = 0 \wedge b = 1$. Thus under SC, it is impossible to get the outcome $a = 0 \wedge b = 0$. To get $a = 0$ you need A2 to happen before B1. Interleaving requires that A1 happens before A2 and B1 happens before B2 and since you need A2 to happen before B1 this means that A1 happens before B2. Thus when $a = 0$, this means that $b = 1$ and vice versa. Under weak memory, the compiler can rearrange some instructions that operate on different memory locations. Because A1 and A2 operate on different memory locations we can rearrange them. This weak behaviour allows you to get the result $a = 0 \wedge b = 0$ as it is no longer necessary that A1 happens before A2, so an execution like 0;A2;B1;B2;A1 becomes possible.

Memory Models While enabling weak memory behaviour can increase the speed of programs, it also allows programs to fail in unexpected ways if the additional behaviours are not considered carefully. To have better control over the allowed and disallowed behaviours memory models were invented. Memory models, such as SC we have seen earlier, define rules which restrict the consistent executions. Stricter memory models like SC and total store ordering (TSO) [20] do not allow the behaviour observed in figure 2.1 called load buffering. There are also more relaxed models such as the Power [2] and Arm [22] memory models which do allow load buffering.

Figure 2.1: Weak Memory Example Program (left), and an Example Execution Graph (right). tA and tB represent two concurrent threads. 0, A1, A2, B1, and B2 represent events in the execution where 0 initializes X and Y to 0. The outcome of the program is defined in terms of the values a and b take after the execution. For the Execution Graph, the black arrows represent the order the events are executed.

Execution Graphs For axiomatic memory models the set of all potential outcomes of the program is described as a set of execution graphs, where each execution graph is matched to one of the potential outcomes of the program. An execution graph G is defined by events which are its nodes, and relations between those events which are its edges. Nodes are often on the level of individual instruction. Common edges are program order (po) and reads-from (rf). po describes the sequence of instructions listed from top to bottom. rf relates each load event L which reads a value from a store event S. All executions of a program P can then be seen as all execution graphs G consistent for the memory model the program operates under. For an example execution graph see 2.1. Here the execution 0;A1;A2;B1;B2 is shown. The outcome of this program is $a = 0 \wedge b = 1$. You can find this outcome by looking at the rf relation. For a the *store* event it reads from is 0, and thus its value is zero. For b it reads from A1 and since X is set to 1 there $b = 1$.

2.2 Program Verification under Weak Memory

Verifying concurrent programs can be done in various ways. One way is to run the program and trace the order of events that were executed. The resulting execution trace can then be used to see whether anything went wrong. This approach might catch the error, but it can also easily miss it if it does not show up in all executions of the program. Using a fuzzer [16, 24], or a model checker [2, 18, 10] gives a much better idea of the possible program behaviours. The idea of fuzzing is to run the program many times and see if an error shows up. This technique has the advantage that it can verify large and complex systems, but it is unable to verify all executions and thus might miss errors. The model checker can verify all possible behaviours of the program. The advantage of model checking is that it can give strong guarantees on the behaviour of the program, but it is currently infeasible to test large systems because any non-trivial concurrent program generates many different executions. The rest of this section gives an overview of how stateless model checkers work under a weak memory context and explains how GenMC, which is the stateless model checker we build on, operates.

Model Checkers Stateful model checkers have been proposed for weak memory models. Still, they are limited in their application by state space explosion and generally have a problem with comparing program states that produce the same outcome but that result from different execution graphs. Stateless model checkers are more promising in this area because they can utilize techniques [1, 25, 7] to reduce the number of executions needed to explore while still ensuring that all the possible behaviours of the program are analysed. Stateless

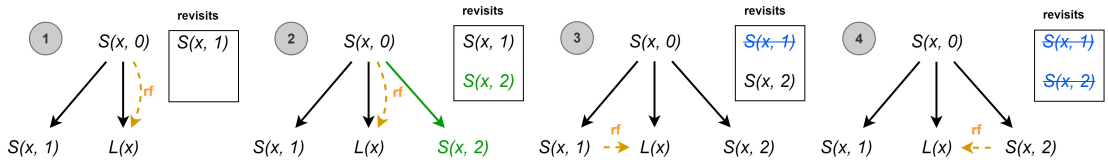

Figure 2.2: GenMC execution graph example. $S(x, 0)$ represents a *store* event writing 0 to variable x. $L(x)$ represents a *load* reading the value of the variable x. rf represents the reads-from relation. An event in green means that it is newly added

model checkers work by enumerating all executions of the program under test and verifying them individually. Stateless model checkers are closely related to the memory model the program operates under since the result of the program under test is determined by the memory model it operates under.

GenMC GenMC is a state-of-the-art LLVM-IR-based stateless model checker for weak memory concurrent C and C++ programs that can verify errors such as data races and safety violations [10]. The stateless model checking algorithm GenMC uses only distinguishes executions based on the po and rf relations which makes it suited to verification under multiple memory models since for these models only the po and rf relations are necessary to reconstruct all the required information.

GenMC Execution Graph Example GenMC creates executions by generating them step by step. Starting with a consistent execution G, it adds a new event to the graph and then verifies whether the new graph G' is still consistent. Once the first execution graph is complete it can backtrack to generate new executions. This is done by keeping track of which *load* operations can read from which *store* operations. It can then generate a new execution by revisiting a *load* operation that has not had all its potential read options explored. Consider the example given in figure 2.2. Let execution *1* be the initial execution. Here the *load* operation is reading from the initial *store0* operation. The *load* could also read from the *store1* and *store2* operations and those executions are given in *1* and *2* respectively.

2.3 Verifying Higher-Level structures

Larger software systems can be structured as layers of abstractions and weak memory concurrent systems are no different. Groups of instructions and atomic operations form concurrent function calls. Sets of these function calls together form concurrent data structures which again are used as a basis to define higher-level concurrent algorithms that use those data structures.

Higher-Level Properties In a similar way it is possible to create layers of abstractions in the properties of concurrent systems. Instruction-level relations can be used to construct relations between function calls. Those function-level relations are then used to define properties for the behaviour of concurrent data structures. Furthermore, the relations of multiple data structures working together can be used to model the behaviour of high-level algorithms. The tool presented implements existing data-structure models that are based on such function-level relations [23].

Chapter 3

Main Ideas

Our Research has three main ideas. First, it is possible to define higher function-level relations as paths of lower instruction-level relations. Second, using layers of abstractions we can separate finding the relations from verifying models based on said relations. Third, many data structure implementations give different guarantees and high-level concurrent data structure models can be used to verify these guarantees. These ideas are combined to extend an existing stateless model checker for weak memory concurrent programs [10] enabling it to verify weak memory concurrent data structure models [23] and variants of them. We then use these extensions to verify numerous concurrent mutex, queue, and stack implementations. The rest of the chapter first gives a high-level overview of the chosen approach before specifying the main contributions of the thesis.

3.1 Chosen Approach

We propose a stateless model-checker based on GenMC [10] that can find function-level relations and has implemented multiple data structure models using these relations. This tool serves as an example of how these layers of abstractions can be used, and as a practical program that can verify whether any C or LLVM-based implementation adheres to these models. A stateless model checker was chosen as the baseline for implementing the tool as it offers verification guarantees, can generate counterexamples whenever it finds an issue and can use techniques to reduce the space the checker needs to explore. These counterexamples are valuable in many practical scenarios, and verifying all possible states of the program guarantees we find any existing model violations.

3.2 Algorithm

Our verification Algorithm follows a similar approach to the verification algorithm used by GenMC. We construct execution graphs incrementally adding one event at a time and checking consistency at each step [10]. All function-level relations we consider have a reads-from (rf) edge as part of the relation. Any time a *load* or *store* event is added to the execution graph it might create a new reads-from (rf) which can allow us to find a new function-level relation. Thus we verify for every *load* and *store* event if any previously unknown relations can be found. If the tool finds new relations it looks up all function calls in the execution graph and calculates all function-level relations between them. These are then used to verify the state of the program.

A first high-level property To better explain the verification algorithm we explore how the tool verifies a mutex implementation. A mutex should enforce mutual exclusion for the data they protect. This means that there should be no two threads that can access that data

concurrently. To accomplish this the mutex enforces there to be a total order on all accesses to the shared variable.

To verify if this total order exists for a given mutex implementation we can look at the mutex *lock* and mutex *unlock* function calls. Under normal operation, each *lock* call will be directly followed by an *unlock* call in the same thread. And each *unlock* call is followed by exactly one *lock* call that next acquires the mutex.

This relation that relates *unlock* calls to the next *lock* call that acquires the mutex we call the communication order (com) relation. Utilising com we can simplify finding this total order assuming normal mutex usage. To verify if a total order of shared variable accesses exists we need only check that: The initialization event matches one *lock* event or that there are no events, each *lock* event has exactly one incoming com edge, and each *unlock* event has at most one outgoing edge to a *lock* event.

Using the approach specified above the tool can verify many such properties using execution graphs consisting solely of function calls and relations between them. This verification can then be applied to multiple data structures without changing the verification algorithm or the checks performed.

3.3 Contributions

The main contributions of this thesis are the practical translations from instruction-level relations to function-level relations, giving an example of how using layers of abstractions we can create concurrency models that do not rely on implementation details. Furthermore, the implemented tool allows for verification of weak memory concurrent mutex, stack, and queue implementations and is used to that end to verify existing implementations of these data structures.

Chapter 4

Technical Details

This chapter describes all changes to the existing code base that enable the model checker to verify the new concurrent data structure models. Before diving into these changes first an understanding of the code base is necessary. The rest of this chapter first explains the notation used in the rest of the chapter. Then it describes each newly added feature in its section that together allows the new models to be verified.

Existing code base GenMC works in three steps that are represented in figure 4.1. First, it invokes the Clang compiler to compile the C/C++ programs to LLVM-IR. The second step transforms the code to make verification more efficient. Then the third step explores the compiled and transformed code to verify the program under test. The verification step again consists of three main components, the Interpreter, the Driver, and the Execution Graph. The Driver decides which executions to explore and in what order that happens by managing the Work Set that the Interpreter operates on. It also calculates the po and rf relations between events using one of the memory model-specific drivers it has access to. Furthermore, It performs the actual verification that checks the program state using the Calculators that operate on the execution graph. Lastly, it notifies the Execution Graph any time the Interpreter finds an event that has to be added to the Graph. The Interpreter is responsible for correctly interpreting the program state and notifying the Driver whenever a potential event is found. The Execution Graph keeps track of the minimal amount of information on the current execution so that any necessary information can be restored later. It also contains calculators to restore information not directly stored whenever necessary.

New Features Three new features have been added to the code to enable the new verification. First, the inline-function pass that is part of the code transformation step is altered to keep the information of what instructions belong to which function call as that information is needed later to construct relations between function calls. Second, we alter the memory

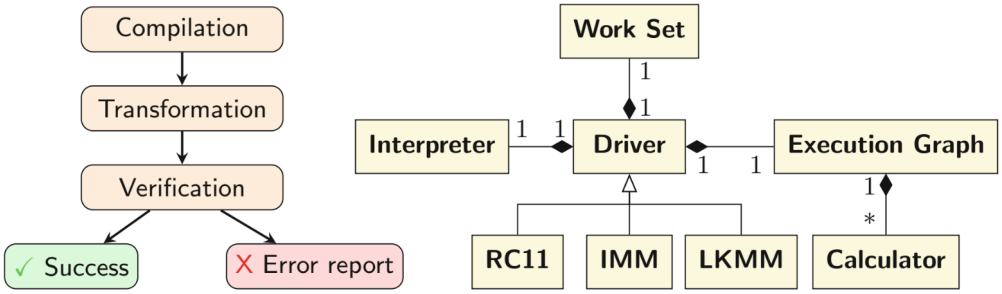

Figure 4.1: The architecture and dynamic components of GenMC [11]

model-specific drivers to find the function-level relations. Third, the driver itself is altered to support data structure model verification.

4.1 Execution Graph Semantics

This section gives the notation used for execution graphs G. This expands upon the notations given by GenMC [10] and adds function call events and function-level relations. Traces of programs are represented by a set of execution graphs. Each execution graph G consists of (i) a finite set of events E and (ii) several relations between those events.

Events An event can be either an instruction-level event eI or a function call event eF. An instruction-level event is a tuple in the form $\langle i, n, l_i \rangle$ where $i \in \mathbf{Z}$ is the thread identifier, $n \in \mathbf{Z}$ is the position of the event within its thread, and $l \in \text{Lab}_i$ is the label of the event. The label of an event can be given by the instruction event label function lab_i. A Function call event is a tuple $\langle l_f, c, f_e \rangle$ where $l_f \in \text{Lab}_f$ is the label of the function call, $c \in \mathbf{Z}$ describes its invocation count, and f_e is set of instruction-level events associated with this function call.

Relations An execution graph G knows two types of relations. One links to instruction-level events, and the other relates to function calls.

- The program order (po) relation, where $\text{po} \subseteq eI \times eI$ captures the order of events in the program's control flow for both function calls and instruction events.
- The Areads-from (rf) relation, where $\text{rf} \subseteq eI \times eI$ associates each *load* event with a set of *store* events that read from that *load*.
- The communication order (com) relation, where $\text{com} \subseteq eF \times eF$ matches those function calls that exchange information.
- The synchronization order (so) relation, where $\text{so} \subseteq eF \times eF$ denotes those $\text{po} \cup \text{com}^+$ paths that contribute to happens-before (hb).
- The local happens-before (lhb) relation, where $\text{lhb} \subseteq eF \times eF$ captures causality between different function calls. lhb is transitive and $\text{po} \cup \text{so} \subseteq \text{lhb}$.

Since both so and lhb are dependent on com it is enough to only keep track of po and com during execution to be able to find the other function level relations later. This results in an execution of G consisting of a tuple $\langle E, \text{rf}, \text{com} \rangle$.

Execution Graph Example An example of a GenMC execution graph can be found in figure 2.2. This graph shows instruction-level events and the relations between these events. However when verifying higher-level concurrent data structure models only function call events and relations between function calls are taken into account. See figure 4.2 for an example. The graph shows two concurrent threads. thread1 contains four function call events, $push0$, $push1$, $pop0$, $push2$, while thread2 contains one event $pop1$. The numbers you can see match the actual values that are pushed onto the stack and popped from the stack. The com edges are used to determine which values are pushed and popped. So each push whose value is not read gets a ? instead of a value as it is unknown what value has been pushed by that operation at this time.

Memory model The main memory model that the relation calculations between function calls are based on is the repaired C11 memory model (RC11) [14]. This is because this memory model is supported by default in GenMC and the data structure models [23] also support it. The RC11 memory model supports multiple modes for fences and accesses which are partially ordered by $\sqsubset$ as seen in figure 4.3 [14].

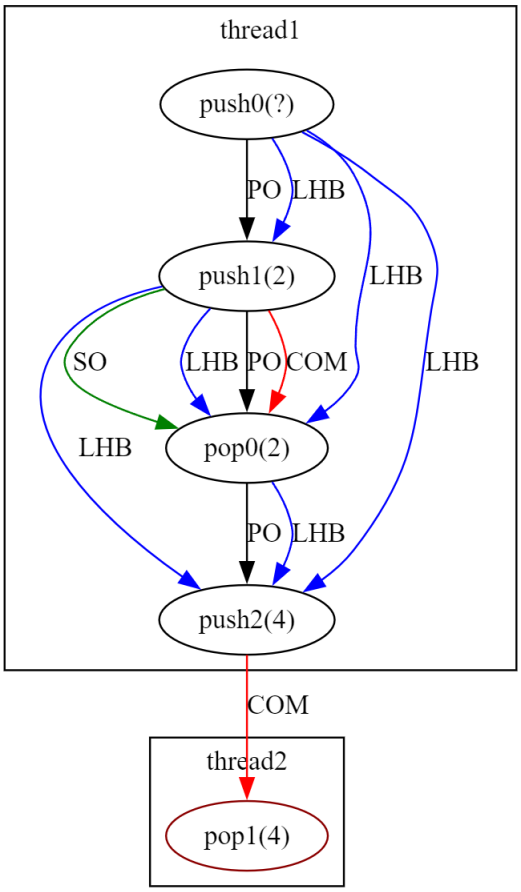

Figure 4.2: Example of an execution graph G of a stack implementation, showing only function call events and relations between function calls.

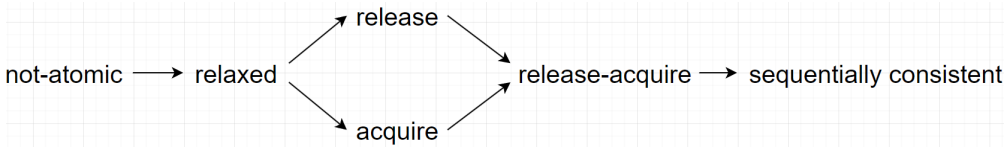

Figure 4.3: memory accesses under RC11

4.2 Tracking inlined functions

As stated above the GenMC framework uses the LLVM backend to inline all non-recursive function calls. This inlining makes it impossible to know what function an event would have belonged to. So it is necessary to keep track of it when the function is inlined.

Debug Information LLVM-IR allows programs to add debug information to instructions. The tool uses this debug information to convey what instruction belongs to which function call as it can be added without modifying the instructions. This debug information can then be used during interpretation to know whether a function call has started or ended. This can then be communicated to the Driver. Early in the process, it was noted that just marking the instructions of the function call was not enough as the marked instructions may get interrupted by unmarked events generated by something like **malloc**. During interpretation, there is no way to differentiate between marked instructions being interrupted and a function call ending.

Marker Instructions Another way to mark what instructions belong to a function call is to have special *function call start* and *function call end* markers that you place on certain instructions. During interpretation, it is then possible to assume any instruction between a *start* and an *end* marker is part of the same function call. The call instruction cannot be used to mark where the function *starts* or *ends* as it gets removed during the inlining process. In the same vein, it is difficult to use the instructions directly before and after the call instruction as a marker as there are cases where these instructions are modified or removed. To ensure that the marker instructions always exist the tool adds two special marker instructions right before the call instructions and right after. These marker instructions are never executed but only used to mark the *start* and *end* of a function call. It is not always enough to insert these instructions as when a call instruction is called from within the body of a while loop it might not immediately see the *end* marker as the function ends. To combat this the execution engine was modified to track which threads have a function that has started but has not ended yet. The tool then uses this knowledge to handle these edge cases.

New Events The tool adds two new events to the execution graph to help keep track of the functions, function start and function end. A function start event is added to a thread whenever the interpreter finds a new function start marker instruction. Similarly, an end event is added when an end marker instruction is found.

Invocation Only tracking when a function call starts or ends does not help differentiate separate calls to the same function. To track these separate calls each function call is assigned a unique integer for its specific function called its invocation. This invocation is zero when no calls to this function exist in the execution graph, or it is one higher than the highest invocation that exists for this function. Two function calls can only share an invocation if they belong to different functions.

4.3 Calculating Function-level Relations

There exist numerous relations between different function calls in literature. This research focuses on four specific ones, namely program order (po), communication order (com), synchronization order (so), local happens-before (lhb).

po The tool finds po by using the start and end events of the function calls. If there is a direct path consisting of event-level po edges between the end of one function call $F1$ and the start of a function call $F2$ without the path being interrupted by a different function call then $(F1, F2) \in \text{po}$.

com Finding com requires you to find out what function calls have exchanged information. For this to be the case you minimally need a *load* event belonging to function call $F2$ to read from (rf) a *store* event belonging to $F1$. A single rf event is not enough however as implementations of concurrent data structures will read values written by other calls without taking any data from the data structure in question. The com relation can only be found once the *load* event exists. The tool finds the inverse of com as it finds a connection between the current and a previous event. While com matches current and future events.

For a basic mutex, it is enough to check if the *load* l in $F1$ is part of a successful compare-and-swap (CAS) or read-modify-write (RMW) operation and if the value it reads from the *store* s in $F2$ is zero. Reading zero here means that the mutex is not currently locked in the tested mutex implementations and combining that with the information that the *load* is part of a successful CAS or RMW operation ensures that it is the only thread that reads that value. Thus in such a case $(F2, F1) \in \text{com}$ as the function call $F2$ successfully communicated that

the mutex was free to $F1$. If a different standard is used in other implementations then it is necessary to change the check on the value of the load.

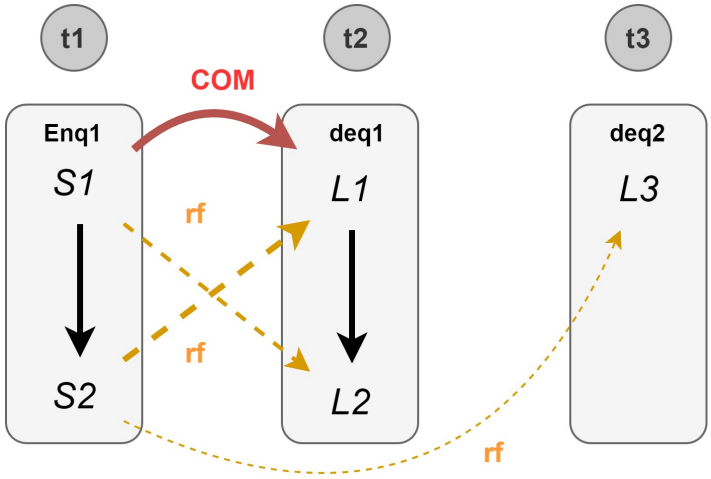

Figure 4.4: Example Execution Graph for showing how com is found. There are three threads, t1, t2, and t3. Each thread consists of one function call.

For the stack and queue finding com is more difficult. Queues and stacks work by having a shared list of nodes that can be added to using either a **push** or **enqueue** operation or removed from using a **pop** or **dequeue**. The value put is never directly read from or written to the data structure, instead it is transmitted using these nodes. Thus in the case where you have a **pop** operation that reads a value put on a stack by a **push** operation, you will first have a *store* $s1$ where the **push** writes the value to the node. Then another *store* $s2$ where the node is written to the stack. The **pop** will then use a *load* $l1$ to retrieve the node. Finally, it reads the value contained using another *load* $l2$. Thus if $\exists s1 \in F1, s2 \in F1, l1 \in F2, l2 \in F2$ such that $(s1, l2) \in \text{rf} \wedge (s1, s2) \in \text{po} \wedge (s2, l1) \in \text{rf} \wedge (l1, l2) \in \text{po}$ then $(F1, F2) \in \text{com}$. The tool does require that $s1$ uses a non-atomic memory access. This allows the tool to differentiate COM paths exchanging the actual values from other nodes that can be passed around due to implementation details. For an example see figure 4.3. Here the tool will find a com relation between t1 and t2 because the path S1 po S2 rf L1 po L2 is the exact path we are looking for. The tool will not find a com relation between t1 and t3 because there is just one rf edge between S2 and L3, which is not enough to construct the path the tool looks for.

so Finding so is much the same as finding com. so is slightly stricter than com in that in addition to the previous path it also requires that $(s1, l2) \in$ hb. Under the repaired C11 memory model (RC11) this means that either $(s1, l2) \in$ po, that $l2$ is of memory order acquire or sequential consistent and that the $s1$ is of memory order release or sequential consistent, or that some other event exists that allows us to infer that $(s1, l2) \in$ hb.

lhb $\text{po} \cup \text{so} \subset \text{lhb}$. Thus if $(F1, F2) \in$ lhb it means that there exists some path consisting of po or so edges connecting $F1$ and $F2$.

4.4 Model Verification

The full verification of the models consists of three different algorithms. The first algorithm decides when we want to verify the model. The second one fully calculates the relations between all the currently existing function calls. Lastly, the third algorithm verifies if the current execution adheres to the chosen model.

It is impractical to test the program at every possible step. This would first be very time-consuming and inefficient as many program points lead to the same state of function calls and found relations. Most of the properties that the current models want to verify are dependent on the com relation. Thus the verification procedure is called whenever a change in the com relation is found. Furthermore, it is important to verify the execution whenever a function call ends. Certain properties need to be checked between fully executed functions. Thus we verify all properties of the selected model on a specific function call whenever one of the following holds: either the newest event in the function call changes com or that event marks the end of the function call. This combined with GenMC's guarantees ensures that we verify every interesting point at least once.

4.5 Implemented Models

The tool supports several concurrent data structure models based on the previously defined function-level relations. Multiple slight deviations of these models are also implemented to help test the implementation and be able to show more about the tested data structures. Currently, only the mutex, stack, and queue data structures are supported. The rest of the section will give an example of a data structure model for a stack and explain how such a base model can be used to define variants. The base models for mutex, stack, and queue are adapted from literature [23]. The full list of models used can be found in the appendices.

Base Stack This is the base model for a consistent stack data structure.

1. there is at most one constructor event.
2. com relates matching **push** and **pop** events.
3. every **push** event matches at most one **pop** and vice versa.
4. every unmatched **pop** returns empty.
5. a **pop** events with a previous unmatched **push** event cannot return empty.
6. every matching edge is synchronizing. com = so.
7. ordered pushed values cannot be popped out of order, events adhere to the last in first out (LiFo) property.

The above stack model ensures that each pushed value is only popped once, that ordered **push** and **pop** operations adhere to the LiFo property, and that **pop** operations cannot fail when there is still data on the stack. See figure 4.2. In this execution, an error is detected when verifying the $pop1$ function call. There exists a com edge between $push2$ and $pop1$. The model requires that com = so and since there is no so edge between $push2$ and $pop1$ an error is raised. There not being a so edge signals that while the **pop** operation is getting a value from the **push** there is no strong synchronization between the threads. So you cannot be sure that $push2$ happens before $pop1$.

Variant Models There are many ways to create variant models. One way is to strengthen the seventh LiFo property to require that all push and pop operations adhere to some total order. A way to relax the model is to remove the fifth property. The new model will allow **pop** operations to fail when there is data on the stack. It is also possible to obtain queue models from the stack model by replacing LiFo with first in first out (FiFo) and requiring com to match **enqueue** and **dequeue** functions instead of **push** and **pop** functions. These variant models have different purposes. Different data structure implementations can give other guarantees and using these variant models allows one to verify whether those guarantees hold in practice.

Chapter 5

Evaluation

This chapter evaluates the research performed and the tool created. The first section describes the experiments run and explains the results. The remaining sections aim to answer the following questions. Does calculating these new relations impact the performance and behaviour of GenMC? Does the proposed way to find function-level relations work in practice? And, what kinds of properties can we verify using these models?

5.1 Results

This section details the results of the performed experiments. First, an overview is given of what each part of the table stands for. Then, the results are explored in more detail.

5.1.1 Table Layout

This subsection describes the meaning of each column of the result tables.

Implementation The implementation refers to the actual code implementation of the tested data structure.

Model The model refers to the concurrency model the code implementation is verified against. The full list of models used and their descriptions can be found in the appendix.

Test The test describes one of three test scenarios. These test scenarios define the driver program that calls the functions of the implemented data structure. There are three test scenarios *A*, *B*, and *C*. Scenario *A* is only used for mutexes. It describes a scenario of N concurrent thread, where each thread looks as follows **lock**; critical section; **unlock**. Scenario B consists of three different types of threads, read threads which only call **pop** or **dequeue**, write threads which only call **push** or **enqueue**, and read-write threads which first call **enqueue** or **push** before calling **dequeue** or **pop**, this scenario B consists of two read threads, three write threads, and one read-write thread. Scenario C consists of three different concurrent threads. Thread1 consists of **push**; **push**; **pop**; **push** for stacks or **enqueue**; **enqueue**; **dequeue**; **enqueue** for queues. Thread2 consists of **pop**; **push** for stacks or **dequeue**; **enqueue** for queues. Thread3 consists of a single **pop** or **dequeue** call for stacks and queues respectively.

Error The error column describes the errors found by the adapted model checker, on the left, and GenMC, on the right. A '-' symbol means that no error was found. The first two errors, (i) and (ii), can be found by both GenMC and the adapted version. These errors are safety violation (i), and non-atomic data race (ii). The third error (iii) denotes a model

violation where an unmatched **lock** event is detected. Error (iv) denotes that an execution graph was found where a **dequeue** or **pop** operation could fail when there was still data on the data structure. Error (v) occurs when multiple **dequeue** or **pop** events are com matched to the same **enqueue** or **push** event. Last, error (vi) says an execution exists where not all com edges are synchronising. So com $\neq$ so.

Executions explored The column labelled execution explored describes the number of executions the adapted model checker, on the left, has fully explored and that were (blocked) and the same for GenMC, on the right. All cases where the executions differ are the result of the adapted model checker finding a data structure model violation and ending exploration early. If the executions explored are denoted as 'number, *' then it means that the number of executions fully explored and blocked were equal between the adapted model checker and GenMC.

Runtime The runtime is a measure of the total runtime of the system when verifying these implementations. The time on the left is the time taken by the adapted model checker and the time on the right is the time the original GenMC code needs. The runtime of the adapted model checker is longer than the runtime of GenMC on the same test. The only time this is not the case is if the adapted model checker stops exploration early due to finding a data structure model violation.

5.1.2 Result Exploration

This section describes the data structure model violations found in more detail. The first paragraph explains the violations found for the mutexes as they all stem from the same origin. The second paragraph describes all errors found for the queue data structure. The last paragraph goes into detail about one specific error found for the stack data structure and discusses the other ones briefly.

Mutex Results The table below shows the results obtained from verifying the tested mutex implementations.

Implementation	Model	Test	Error	Executions explored	Runtime
mutex	*mutex*	A	(iii), -	0(1), 7086(776)	0.07s, 0,49s
alt-mutex	*mutex*	A	-, -	7086(776), *	0.62s, 0.33s
spinlock	*mutex*	A	-, -	14400(199434), *	4.33s, 1.43s
ttaslock	*mutex*	A	-, -	14400(64255), *	5.93s, 2.11s
ttaslock-opt	*mutex*	A	-, -	120(5561), *	0.31s, 0.16s
ticketlock	*mutex*	A	(iii), -	0(1), 120(774)	0.05s, 0.09s
twalock	*mutex*	A	(iii), -	0(1), 798720(461396)	0.17s, 57.54s

Mutex Explanation For the mutex data structure all violations found were the same. All violating graphs had a **lock** call that was not matched by a previous **unlock** call. This is a violation of the mutex data structure model since the mutex model requires all lock events to be matched. For one of the three cases, the mutex implementation, this violation came from the fact that this implementation does not call an init function before creating the threads. While the code still operates fine, as can be seen in the mutex-alt implementation which is the same implementation that does call some init function, this causes there to be no event before the first **lock** which ensures the first **lock** can never be matched by a previous function event thus causing a model violation. The other two violations are caused by something completely different. The ticketlock and twalock implementations use a different system from the rest

of the mutex implementations to determine what thread can next enter the critical section. This causes the com calculation to fail which causes unmatched **lock** events to exist.

Queue Results The table below shows the results obtained from verifying the tested queue implementations.

Implementation	Model	Test	Error	Executions explored	Runtime
ms-queue	*weak-q*	B	(i), (i)	16(58), *	0.09s, 0.05s
ms-queue-d	*weak-q*	B	-, -	2442(37630), *	25.01s, 18.86s
ms-queue-d	*hw-q*	B	-, -	2442(37630), *	24.94s, 18.86s
ms-queue-d	*hw-q*	C	-, -	100(1325), *	0.12s, 0.09s
qu	*weak-q*	C	-, -	101254(0), *	4.80s, 2.83s
qu	*hw-q*	C	(iv), -	2856(1), 101254(0)	0.32s, 2.83s
qu-opt	*weak-q*	C	(ii), (ii)	1(1), *	0.11s, 0.09s

Queue Explanation The queue implementations do all work with the com definition, so any violation caught by the modified model checker is caused due to the implementation not guaranteeing what the tested model requires. Both the qu and the ms-queue-dynamic implementations are interesting to note. The qu runs into the issue that dequeue operations might fail when there is still data on the stack. The ms-queue-dynamic implementation is the only data structure that does not run into any problems when run on all tests with the stricter model.

Stack Results The table below shows the results obtained from verifying the tested stack implementations.

Implementation	Model	Test	Error	Executions explored	Runtime
dq	*weak-s*	C	(ii), (ii)	7(1), *	0.08s, 0.05s
dq-opt	*weak-s*	C	(ii), (ii)	1(1), *	0.07s, 0.05s
stc	*weak-s*	C	(v), -	1403(1), 23230(0)	0.20s, 1.18s
stc-opt	*weak-s*	C	(vi), -	0(1), 1740(0)	0.07s, 0.14s
treiber-stack	*weak-s*	B	(iv), (iv)	7(2), *	0.14s, 0.07s
treiber-stack-d	*weak-s*	B	-, -	14056(134400), *	16.65s, 12.40s
treiber-stack-d	*c-s*	B	-, -	14056(134400), *	16.48s, 12.40s
treiber-stack-d	*c-s*	C	(iv), -	22(50), 168(454)	0.07s, 0.07s

Stack Explanation For the stack data structures it is interesting to look at the stc, stc-opt, and treiber-stack-dynamic implementations. For the stc implementation look at figure 5.1. Here a **push** event matches two **pop** events. The cause of this is that in this implementation the **pop** function always reads the value of the current node that is at the head of the stack. It afterwards checks if any collisions occurred by trying to remove the current head of the stack with a CAS operation. If this CAS fails then the function returns an error code, if it does not then it returns a success code. The problem is that in the case that it returns an error code the data that was on the head of the stack is still returned to the calling thread. A programmer might check this error code to know whether it can use the value or not, but in any case, the value is effectively duplicated which can often be undesirable behaviour. The stc-opt implementation relaxes its accesses to an amount that the com edges are no longer synchronising as can be seen in 4.2. The treiber-stack-dynamic implementation is interesting as it shows that different test scenarios are necessary to be able to catch all errors.

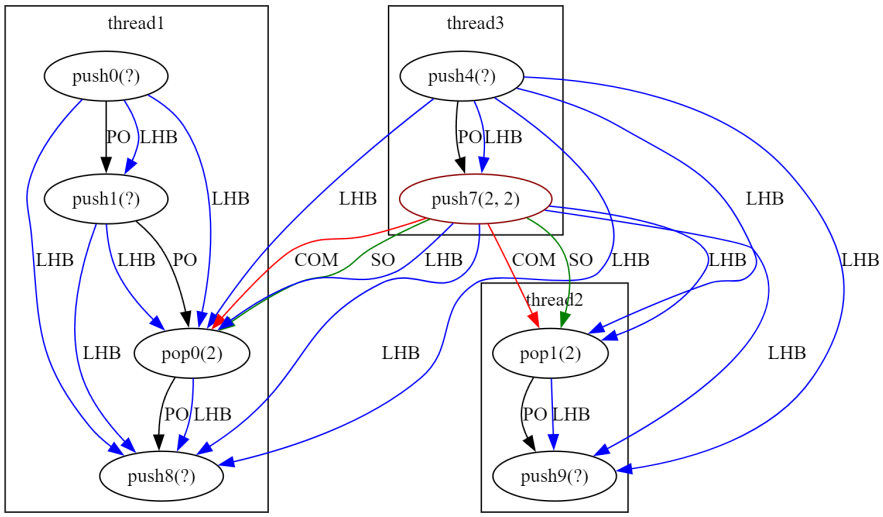

Figure 5.1: Stc: Two pop operations matched by the same push. This execution shows three concurrent threads. Event $push7$ has a com edge to both $pop0$ and $pop1$ which is not allowed by the weak-stack model.

5.2 Correctness of Implementation

Implementing these models into GenMC required many changes to the original code. These changes resulted in adding three new features to the model checker. These features are the tracking of inlined functions, calculations on high-level relations, and the verification of the models themselves. Most of these features are extensively tested by hand in addition to testing if the behaviour of GenMC stayed the same. The first paragraph explains the testing performed on the function tracking. The second paragraph elaborates on how the function-level relation calculation and the data structure model verification were tested. Third, we elaborate how we verified the underlying behaviour of GenMC has not changed due to the added code. Lastly, the fourth paragraph explains how seeded bugs were used to further test the model checker.

Tracking Inlined Functions Tracking the inlined functions is tested by adding and removing lines of code from the tests in predictable ways and then verifying if the number and type of events in the found function correctly correlated with the line of code added or removed.

Calculating Function-level Relations & Data Structure Models The calculation of high-level relations was tested in combination with the models as they are so intertwined that it is difficult to test them separately. They were tested using end-to-end tests similar to the implementation tests. These tests work by modifying axioms and verifying if everything is calculated correctly. For example, for the *mutex* model, a test was to see what the checker would find if you require each Unlock to be matched by two Lock events. That is to say, two different **lock** events should directly follow the **unlock** event. In a correct mutex implementation, this should not happen. It would allow two threads to enter the critical section simultaneously. The checker would then correctly state that it cannot find a second **lock** event that the **unlock** matches with, which is what we expect.

Behaviour of the Model Checker It is important that the underlying behaviour of the model checker is not affected by any of the code changes performed. This allows statements of soundness and completeness for the GenMC stateless model checker algorithm to be used

as a basis for discussing the same topics for the verification algorithms used in this thesis. The experiments ran give us reasonable certainty that the underlying GenMC behaviour is not changed by any of the code modifications. This is because whenever the adapted model checker did not find an error and had its execution stopped early both GenMC and the adapted model checker ran the same number of executions and found the same errors with the same execution graphs giving these errors.

Seeding bugs Another way of testing the model checker is to seed bugs in the implementations and see if the tool can find those bugs. We tested two main ways of seeding bugs. First, the memory accesses in the implementation were relaxed and second, the CAS operations were broken up into two separate operations. Relaxing the memory accesses did lead to errors, but in all cases, GenMC would raise an error before the data structure models failed. When splitting the CAS operations we can find an error for the ms-queue-dynamic where the data structure would allow multiple **dequeue** operations to read from the same **enqueue**.

Chapter 6

Discussion

This chapter discusses the contribution of the research and the tool created. It uses the results obtained from the evaluation chapter and discusses those findings in more detail. First, this chapter discusses the implemented algorithm's soundness, completeness, and efficiency. Then the chapter names some strong points of the system and its findings, then it continues debating this research and its limitations.

6.1 Performance

This section explains the performance of the system. It also details the soundness and precision of the implemented verification algorithm.

Performance The currently implemented verification algorithm is not optimal. The underlying GenMC revisit algorithm does optimally explore each interesting execution exactly once. However certain things hold the current implementation back from being optimal. First, it is currently required for the algorithm to verify the model state whenever a com edge is added and whenever a function call ends. A function call ending might change some verification properties, but often it does not, so the same state of the execution graph gets verified more than once.

soundness & completeness Completeness requires the algorithm to not have false positives, so any correct implementation should pass the models. Soundness requires the algorithm to not have any false negatives, any implementation that passes should be correct. The GenMC revisit algorithm guarantees that if a certain behaviour does not show up during its exploration then it cannot show up. This combined with the verification algorithm executing every time a com edge is added or whenever a function call ends ensures that any interesting state for function events is explored at least once. So any behaviour that does not show up cannot show up. This ensures that the algorithm is complete, as a correct implementation will not generate an erroneous graph. The algorithm is not fully sound for all properties. This is because some properties rely on lhb to order events. lhb is not a strong property. There are many executions in which parts of the graph are disconnected or that no clear way exists to order the events using just lhb. Because of this properties based on lhb cannot always be verified. The decision was made to only report an error when a clear error is found and not whenever there might be a potential model violation. Because of this, all properties based on lhb are not fully sound as an erroneous graph might pass as correct since not enough information is available to prove otherwise.

6.2 Strong points

The tool and the other outcomes of this Thesis serve as a strong foundation for any future work that aims to formulate layers of abstractions between different levels of concurrent systems. The tool can verify many different data structure implementations, can verify multiple different models, is quite general, and is easily extensible.

Practical The outcome of this Thesis has many practical applications as it can verify many different models on a wide variety of implementations. It can run on so many implementations because GenMC accepts most code written in C and other languages that can compile to LLVM with some limitations.

General The tool is not dependent on any specific form of implementation and testing a completely new implementation can thus be done without having to change any code. This also means that the verification and high-level-relation calculation can be implemented in other checkers, be it a model checker or an execution trace analysis.

Easily extensible The tool is very easily extensible as adding new models to the code or adding new relations has no impact on any existing ones and thus minimal changes to the code are needed to get the new model to work.

6.3 Limitations

This Thesis is not without its limitations. It is difficult to determine what guarantees the model checker provides with how tests are currently used. Some styles of implementations are not yet supported. The current way that com is calculated does not work for the other mutex model. Lastly, The revisit algorithm used by GenMC is not proven optimal for exploring execution graphs based on function calls and relations between them.

Tests For a test to run you have to provide a driver program that creates all the threads and calls that will be in the program. Creation of good or interesting driver programs is a difficult problem as without more information it is difficult to know if adding some more calls or changing the order of operations in a thread can lead to different behavior. Thus these driver programs are created with the main idea that usually more calls and threads should lead to more possible executions which could lead to new behavior.

Unsupported Implementations Some concurrent data structures are implemented using what can be called a prepare publish style. This uses two different function calls, for example, Write-prepare and Write-publish, instead of one Enqueue call. These implementations can be useful as you can for instance make different threads to prepare a batch of writes while the main thread can do something else until all those threads are finished. Afterwards, the main thread can write the batch in one go. This style of implementation is currently not supported in the tool as it would require a new way to match these separate functions onto a single call.

COM Currently COM is found as a restriction on read-from with a specific set of qualities. As can be seen from the ticketlock implementation the current formulation of COM does not work for every mutex implementation. There might exist a formulation which is completely independent of the underlying implementation, but that is something future work can elaborate upon.

Model Checker Algorithm The algorithm used by GenMC for the verification revisits reads based on which writes the read can still read from. This done well allows for optimal exploration of all execution graphs under the original assumptions. However, there might be a better revisit algorithm that works based on the additional information that the tracked function calls provide since currently as we previously discussed there are scenarios in which the same execution graph, from the perspective of function calls and their relations, is explored multiple times.

Chapter 7

Related work

This chapter describes related works to the research done in this Thesis. First, other tools like fuzzers can verify properties on weak memory concurrent systems. Then, other model checker algorithms are used to verify the properties of these systems. Lastly, other research delves into formulating higher-level models.

Fuzzers Fuzzers like C11tester [16] and Callfuzzer [24] are able to verify properties on concurrent systems. The advantage such systems hold is speed, allowing them to verify larger and more complex systems than model checkers like ours can. However, this results in a lack of soundness and completeness as not all possible execution graphs of the program under test can be verified.

Model checkers Many different model checkers have been proposed for verifying concurrent systems. Some stateful model checkers exist for verifying concurrent systems [8, 13, 21], but these are often limited by state space explosion. Stateless model checkers, such as Herd [2], CDSchecker [19], Nitpick [5], and Cppmem [4], using some techniques to reduce the number of executions to explore are more promising. These model checkers can verify a wide variety of properties, but they do not allow defining of the higher-level relations and models that this work is based on. Both works are orthogonal to each other as both verification and error checking types are valuable.

Lincheck Lincheck [12] combines stress testing with bounded model checking to verify concurrent algorithms on the Java Virtual Machine. This combination lets it verify complex systems with better certainty than a fuzzer. It also allows for easy creation of concurrent tests for complicated systems. It is close to this work since its user-friendly concurrency test creation enables users to write tests for concurrent systems without understanding all the intricate implementation details. The higher-level relations and models this research works on would allow for similar easier ways to create concurrent tests by defining higher-level models that verify those properties.

Data structure models There exists research on developing higher-level formal models for concurrent data structures [3, 23, 6]. Showing that more of these properties can be implemented in a similar version to the ones implemented in this paper would be a great addition. These data structure models are otherwise mostly orthogonal to this research as there are no tools that implement verification for such models that this work can be compared against.

Chapter 8

Conclusion

This thesis presents a stateless model checker to verify the behaviour of weak memory concurrent data structure implementations. The tool is used to verify real-life mutex, queue, and stack implementations and can verify important properties of these concurrent data structures. The approach proposed to find function-level relations works for several concurrent data-structure implementations. This approach and the concurrent data structure models generalise to multiple implementations of the same data structure, and the tool created can easily be extended with new models and data structures.

Future Work Several things could be improved upon in the future. First, implementing more memory and data structure models and adding implementations improves the tool's usability. Many different models and data structures can be studied in the future. An interesting example is the Linux RCU that operates under the Linux Kernel memory model. Another interesting option is to look at garbage collection algorithms and see how those can be modelled. Second, combining a test generator with the tool to create more interesting driver programs. Automatically generating driver programs is a good step to improve the guarantees the tool provides. This allows the tool to test multiple driver variations without user input. It is already possible for a user to create many different driver programs, but this is a tedious task and having an automatic way to generate and test these variations increases the usability. The tool currently only supports the RC11 memory model implemented within GenMC. Other memory models are used in practice and being able to test the programs on them would be a great addition to the tool.

Bibliography

[1] Parosh Abdulla et al. "Optimal dynamic partial order reduction". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: Association for Computing Machinery, 2014, pp. 373–384. ISBN: 9781450325448. DOI: 10.1145/2535838.2535845. URL: https://doi.org/10.1145/2535838.2535845.

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory". In: *ACM Trans. Program. Lang. Syst.* 36.2 (July 2014). ISSN: 0164-0925. DOI: 10.1145/2627752. URL: https://doi.org/10.1145/2627752.

[3] Jade Alglave et al. "Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel". In: ASPLOS '18. Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 405–418. ISBN: 9781450349116. DOI: 10.1145/3173162.3177156. URL: https://doi.org/10.1145/3173162.3177156.

[4] Mark Batty et al. "Mathematizing C++ concurrency". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 55–66. ISBN: 9781450304900. DOI: 10.1145/1926385.1926394. URL: https://doi.org/10.1145/1926385.1926394.

[5] Jasmin Christian Blanchette et al. "Nitpicking c++ concurrency". In: *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*. PPDP '11. Odense, Denmark: Association for Computing Machinery, 2011, pp. 113–124. ISBN: 9781450307765. DOI: 10.1145/2003476.2003493. URL: https://doi.org/10.1145/2003476.2003493.

[6] Thomas A. Henzinger et al. "Quantitative relaxation of concurrent data structures". In: *SIGPLAN Not.* 48.1 (Jan. 2013), pp. 317–328. ISSN: 0362-1340. DOI: 10.1145/2480359.2429109. URL: https://doi.org/10.1145/2480359.2429109.

[7] Jeff Huang. "Stateless model checking concurrent programs with maximal causality reduction". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 165–174. ISBN: 9781450334686. DOI: 10.1145/2737924.2737975. URL: https://doi.org/10.1145/2737924.2737975.

[8] Bengt Jonsson. "State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version)". In: *SIGARCH Comput. Archit. News* 36.5 (June 2009), pp. 65–71. ISSN: 0163-5964. DOI: 10.1145/1556444.1556453. URL: https://doi.org/10.1145/1556444.1556453.

[9] Jan-Oliver Kaiser et al. "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris". In: *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Ed. by Peter Muller. Vol. 74. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum fur Informatik, 2017, pp. 17.1–17.29. ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPIcs.ECOOP.2017.17. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2017.17.

[10] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. "Model checking for weakly consistent libraries". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 96–110. ISBN: 9781450367127. DOI: 10.1145/3314221.3314609. URL: https://doi.org/10.1145/3314221.3314609.

[11] Michalis Kokologiannakis and Viktor Vafeiadis. "GenMC: A Model Checker for Weak Memory Models". In: *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 427–440. ISBN: 978-3-030-81684-1. DOI: 10.1007/978-3-030-81685-8_20. URL: https://doi.org/10.1007/978-3-030-81685-8_20.

[12] Nikita Koval et al. "Lincheck: A Practical Framework for Testing Concurrent Data Structures on JVM". In: *Computer Aided Verification*. Ed. by Constantin Enea and Akash Lal. Cham: Springer Nature Switzerland, 2023, pp. 156–169. ISBN: 978-3-031-37706-8.

[13] Michael Kuperstein, Martin Vechev, and Eran Yahav. "Automatic inference of memory fences". In: *SIGACT News* 43.2 (June 2012), pp. 108–123. ISSN: 0163-5700. DOI: 10.1145/2261417.2261438. URL: https://doi.org/10.1145/2261417.2261438.

[14] Ori Lahav et al. "Repairing sequential consistency in C/C++11". In: *SIGPLAN Not.* 52.6 (June 2017), pp. 618–632. ISSN: 0362-1340. DOI: 10.1145/3140587.3062352. URL: https://doi.org/10.1145/3140587.3062352.

[15] Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers* C-28.9 (Sept. 1979), pp. 690–691. ISSN: 1557-9956. DOI: 10.1109/TC.1979.1675439.

[16] Weiyu Luo and Brian Demsky. "C11Tester: a race detector for C/C++ atomics". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 630–646. ISBN: 9781450383172. DOI: 10.1145/3445814.3446711. URL: https://doi.org/10.1145/3445814.3446711.

[17] Jeremy Manson, William Pugh, and Sarita V. Adve. "The Java memory model". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 05. Long Beach, California, USA: Association for Computing Machinery, 2005, pp. 378–391. ISBN: 158113830X. DOI: 10.1145/1040305.1040336. URL: https://doi.org/10.1145/1040305.1040336.

[18] Madanlal Musuvathi et al. "Finding and Reproducing Heisenbugs in Concurrent Programs." In: *OSDI*. Vol. 8. 2008. 2008.

[19] Brian Norris and Brian Demsky. "CDSchecker: checking concurrent data structures written with C/C++ atomics". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 131–150. ISBN: 9781450323741. DOI: 10.1145/2509136.2509514. URL: https://doi.org/10.1145/2509136.2509514.

[20] Scott Owens, Susmit Sarkar, and Peter Sewell. "A Better x86 Memory Model: x86-TSO". In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 391–407. ISBN: 978-3-642-03359-9.

[21] Seungjoon Park and D.L. Dill. "An executable specification and verifier for relaxed memory order". In: *IEEE Transactions on Computers* 48.2 (1999), pp. 227–235. DOI: 10.1109/12.752664.

[22] Christopher Pulte et al. "Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158107. URL: https://doi.org/10.1145/3158107.

[23] Azalea Raad et al. "On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290381. URL: https://doi.org/10.1145/3290381.

[24] Koushik Sen. "Effective random testing of concurrent programs". In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. Atlanta, Georgia, USA: Association for Computing Machinery, 2007, pp. 323–332. ISBN: 9781595938824. DOI: 10.1145/1321631.1321679. URL: https://doi.org/10.1145/1321631.1321679.

[25] Naling Zhang, Markus Kusano, and Chao Wang. "Dynamic partial order reduction for relaxed memory models". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 250–259. ISBN: 9781450334686. DOI: 10.1145/2737924.2737956. URL: https://doi.org/10.1145/2737924.2737956.

Acronyms

SC sequential consistency

RC11 repaired C11 memory model

TSO total store ordering

po program order

rf reads-from

rmw read-modify-write pairs

com communication order

so synchronization order

lhb local happens-before

hb happens-before

RMW read-modify-write

CAS compare-and-swap

FiFo first in first out

LiFo last in first out

Appendix 9

A

9.1 Data Structure Models

mutex

1. there is at most one constructor event.
2. com matches mutex **unlock** and **lock** events.
3. each **lock** event is matched by at most one event and vice versa.
4. all lock events are matched.
5. every matching edge is synchronizing. com = so.

weak-queue

1. there is at most one constructor event.
2. com relates matching **enqueue** and **dequeue** events.
3. every **enqueue** event matches at most one **dequeue** and vice versa.
4. every unmatched **dequeue** returns empty.
5. every matching edge is synchronizing. com = so.
6. ordered enqueued values cannot be popped out of order, events adhere to the FiFo property.

hw-queue

1. there is at most one constructor event.
2. com relates matching **enqueue** and **dequeue** events.
3. every **enqueue** event matches at most one **dequeue** and vice versa.
4. every unmatched **dequeue** returns empty.
5. a **dequeue** events with a previous unmatched **enqueue** event cannot return empty.
6. every matching edge is synchronizing. com = so.
7. ordered enqueued values cannot be popped out of order, events adhere to the FiFo property.

weak-stack

1. there is at most one constructor event.
2. com relates matching **push** and **pop** events.
3. every **push** event matches at most one **pop** and vice versa.
4. every unmatched **pop** returns empty.
5. every matching edge is synchronizing. com = so.
6. ordered pushed values cannot be popped out of order, events adhere to the LiFo property.

c-stack

1. there is at most one constructor event.
2. com relates matching **push** and **pop** events.
3. every **push** event matches at most one **pop** and vice versa.
4. every unmatched **pop** returns empty.
5. a **pop** events with a previous unmatched **push** event cannot return empty.
6. every matching edge is synchronizing. com = so.
7. ordered pushed values cannot be popped out of order, events adhere to the LiFo property.