Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Rust Frontend for GenMC

Bachelor Thesis

Arbenit Kamberi

August 10, 2025

Advisors: Prof. Dr. Michalis Kokologiannakis
Department of Computer Science, ETH Zürich

Abstract

Stateless model checkers are widely used for verifying concurrent programs, but their application to Rust has thus far been limited. Existing model checkers lack full support for aspects of Rust's memory model, which prevents accurate verification of concurrent Rust programs. In this thesis, we extend GenMC, a state-of-the-art model checker for concurrent C/C++ programs that supports various weak memory models and is optimal in the number of executions explored, to also verify Rust programs. GenMC verifies the LLVM-IR produced during compilation of a C/C++ program. We extend it to support various constructs in LLVM-IR generated by the Rust compiler and develop approaches for overcoming the challenges we faced in doing so. We provide custom implementations of Rust's standard library concurrency APIs so they can be used with GenMC. We successfully ran and compared a collection of tests in Rust with the corresponding output for the same test in C. As a case study, we verified various concurrent data-structures of a popular, open-source Rust crate. Our work enables the automatic verification of Rust programs and provides a great foundation for future extension to more concurrency primitives in Rust.

Keywords: Model checking, Concurrency, Rust, LLVM

Contents

Chapter 1

Introduction

Concurrent programming is difficult to get right. The rise of weaker memory models which enable the compiler to apply more aggressive optimizations and even the CPU to optimize execution during runtime make the job of a software developer building concurrent programs even more difficult. These optimizations are usually designed to be sound in the single-threaded case. The use of a blocking lock on a critical section of the code to ensure only a single thread can enter it at a time would be a relatively easy option, but can lead to the loss of most performance gains one aspires by using multiple threads if used too much or in the wrong places, making lock-free alternatives often more desirable.

There are many things a software developer needs to consider when working in such a concurrent environment. What ordering to use on this atomic operation? Where is a fence needed? Is a compiler-level memory barrier sufficient? Each of them affects how multiple threads can interact (or negatively interfere) with each other. The range of possible interleavings of instructions during execution can grow enormous very quickly, making human reasoning about concurrent programs very challanging. Formal methods for verifying concurrent programs have therefore become of utmost importance in critical applications.

GenMC is a stateless model checker that can verify safety properties of concurrent C/C++ programs under a wide range of weak memory consistency models. It compiles the input to the LLVM Intermediate Representation (LLVM-IR) and performs the verification on the LLVM-IR level. For verification it repeatedly executes the LLVM-IR, until all possible program behaviors have been explored.

This foresight of basing verification on a popular intermediate representation like LLVM-IR enables GenMC to be extended to possibly all programming languages using it. In this thesis, we are making use of this by extending

GenMC so that it can verify Rust programs.

To be able to execute the program under test, GenMC expects that the produced LLVM-IR uses a concurrency API (e.g. for thread creation) that it can intercept during execution, and ensures that this is the case by providing custom headers for standard C libraries such as pthread.h and stdatomic.h. Unfortunately, however, expecting a particular API means that other languages that compile to LLVM-IR (e.g. Rust) are not immediately supported as the respective compilers do not produce GenMC-compliant LLVM-IR out of the box. We will provide custom implementations of commonly used Rust concurrency primitives and perform sound transformations on the LLVM-IR so that the resulting LLVM-IR has a structure consistent with what GenMC expects (as done for C/C++), making it ready for verification.

1.1 Our Contribution

We extend GenMC to Rust, adding a powerful tool to the Rust landscape, helping Rust developers achieve the memory safety the language strives for. By analyzing the LLVM-IR generated by Rust and supporting LLVM-IR transformations for various idioms characteristic to the Rust language, we also support a wider range of C-programs as well while adding resilience to these transformations. This includes the handling LLVM's memory intrinsics which now take all arguments into account, allow for handling of a much wider range of data-types and work on MacOS too. We can now also check if binary operations overflow and preemptively eliminate many more reads to uninitialized memory than before. This provides a great base for any other future extensions to GenMC. We evaluate our work by comparing results in Rust with results in C and also verify a popular, open-source library for concurrent programming as a case study.

1.2 Outline

We give some more detailed descriptions about GenMC, Rust and LLVM-IR in the background section. GenMC has three stages: compilation, transformation and verification. For each of those stages, we discuss the changes applied for that stage, which functionality those changes support for verifying programs, why we took the approach we did, and how these changes stand in relation to other stages. After, we evaluate our work first by translating many pre-existing C-tests into Rust and comparing the results and the number of executions explored, then by verifying implementations of data-structures used by a popular, open-source Rust concurrency library as a case study. We end the thesis with a conclusion of our work, reiterating achievements and ongoing issues, comparing with related work and presenting ideas for future expansion on the basis we delivered.

Chapter 2

Background

GenMC [14] was originally designed to verify C/C++ applications, operating on the produced LLVM-IR. The use of such a popular intermediate representation gives it the advantage that it can be extended to possibly every programming language using LLVM-IR in its compilation pipeline, like Rust.

GenMC has three main stages:

Compilation: The client program is compiled using Clang, stopping at the stage where LLVM-IR is generated. Custom headers are always included to reroute POSIX system-calls to GenMC's equivalent interfaces instead, e.g.:

`pthread_create` → `__VERIFIER_thread_create` and

`pthread_mutex_lock` → `__VERIFIER_mutex_lock`

This ensures that these calls can be intercepted. We will refer to these and more custom interfaces as GenMC's internal functions. Another highly relevant internal function is `__VERIFIER_assume`, which simply indicates that an execution trace should not be explored further after that point if a specified condition does not hold.

Transformation: The produced LLVM-IR goes through multiple transformation stages to help later verification such as replacing spin-loops or collecting debugging information.

Verification: The transformed LLVM-IR is verified using GenMC's execution engine that implements an interpreter for LLVM-IR and a driver. The driver schedules the executions to be ran by GenMC's interpreter while also interacting with it to detect actions that may be visible by other threads like atomic operations or writes to shared memory, ensuring that all possible executions can be discovered and explored. If an error is found, verification is halted and the specific execution causing the error is reported with an execution trace containing relevant event-labels (*start/join thread*, *malloc*, *free*, *atomic write/reads*, etc.) and relations (e.g. *reads-from*) between the labels.

The Rust language [30] has many built-in features to prevent common memory safety violations or concurrency issues already at compile time, making the language ever more appealing since its inception. Rust's data ownership and borrowing model aim to prevent many common safety violations. *Use-after-free* and *double-free* are prevented with Box and atomic reference counters, where Box ensures data is freed when the Box is dropped, thus making the reference to the data inaccessible, and atomic reference counters ensure shared data is only dropped when all threads do not use it anymore. Lifetimes ensure *references are always valid*. *Reads from uninitialized memory* must be explicitly applied. *Data races* are prevented by letting only a single thread hold a mutable reference to the data.

Not all memory safety violations and concurrency issues can be prevented at compile time. Many violations can arise with the use of the `unsafe` keyword, which can be necessary in certain instances. If we want to use non-blocking, lock-free data-structures for sharing memory over multiple threads, then the data held by that data-structure cannot be owned by any single thread by its very nature. Interactions with C-code through FFI-interfaces are inherently unsafe as Rust cannot enforce its restrictions across language boundaries. We may also need to ensure a specific condition always holds at certain points in the program to ensure correctness of the algorithm. These reasons make a model-checker still very much needed and viable for Rust.

Rust is in a first step lowered to MIR (Mid-Level IR), before being lowered to LLVM-IR, on which GenMC performs its verification. LLVM-IR can then be lowered to assembly for a wide range of target architectures. It therefore also provides certain intrinsics which are not meant to have direct equivalent LLVM-IR implementations per se as they are meant to be implemented directly in optimized assembly (e.g. [24, Arithmetic with Overflow Intrinsics]). Unlike C/C++, Rust extensively uses `invoke` instead of `call` [24, 'invoke' Instruction, 'call' Instruction] in the LLVM-IR to execute a function, mirroring the extensive use of a `Result`-object and the notion of unwrapping in the Rust language itself.

Chapter 3

Compilation

Behind many of Rust's STD interfaces, it uses Libc system calls which we need to replace with GenMC's equivalents (e.g. `pthread_create` → `__VERIFIER_thread_create`) such that these calls can be intercepted during execution. The goal of this chapter is for the Rust compiler to emit LLVM-IR code with calls to those GenMC's interfaces instead.

Replacing Libc: One option we explored was linking Rust's STD with a custom Libc implementation. This is the equivalent approach to what GenMC does in C, where it includes custom C-headers which overwrite the system default headers to reroute these Libc calls to GenMC's equivalents. This is not possible in Rust because for all built-in target-architectures, a pre-built copy of Rust's STD comes with it [32, 7.1 Built-in Targets]. Therefore, it already contains the Libc-calls it was originally built with. Linking a static C runtime (i.e. Libc implementation) is possible for certain target-architectures [30, 15 Linkage], but this does not help us as this linking still happens after the LLVM-IR is already generated and emitted.

We also explored the possibility of building Rust's STD from scratch, even with a self-defined target-architecture. This route was abandoned due to it, although possible, not being well supported by the Rust Project Team and being very cumbersome to maintain for many different Rust versions over a longer period of time.

The "genmc-std" wrapper: We instead build a wrapper around Rust's STD. A crate we name "genmc-std" that re-exports all types and interfaces in Rust's STD while overwriting just the modules we need to provide custom implementations for, e.g. managing threads or mutexes. The user can continue to use "std" as they would with Rust's STD without noticing the difference in their code. This is possible due to Rust's notion of preludes: "A *prelude* is a collection of names that are automatically brought into scope of every module in a crate." [30, 12.3 Preludes]. Rust's STD is linked into

every module as a prelude by default. We can specify our genmc-std crate as an external prelude with the alias "std", thus overwriting the "std"-entry that usually links to Rust's STD. When specifying an external prelude, it is applied to every module in the pipeline during compilation.

In the following sections we describe different parts of our genmc-std crate and how these are implemented to support different functionality we require. In the last section we present the frontend, where we describe the steps we take to compile Rust-inputs with our custom genmc-std crate to LLVM-IR.

3.1 Spawning & Joining Threads

This section describes the thread-module in our newly created genmc-std crate which provides the same interface with the same type-requirements as the threads-interface in Rust's STD [31, std::thread::spawn].

We modeled our implementation strongly after Rust's STD, leaving out everything not required for our purposes.

3.1.1 Handling Closures

Closures are central for multithreading in Rust, they're used for borrowing and transferring ownership. A closure can capture variables in its environment using the `move`-keyword, thus taking ownership of those variables, this is how data is passed to the new thread as the closure takes no arguments. These variables can then not be used anymore in the original thread as they are now owned by the new thread. For sharing data across threads safely, atomic reference counters [31, std::sync::Arc] are used.

Getting Box [31, std::boxed::Box] to work was fundamental for this to be possible, it handles fat pointers, preparing of arguments and dynamic invocation in this case. See Chapter 4 for the work we did to enable the use of Box.

We first examine how closures are compiled. When we pass a closure to `thread::spawn`, the closure body is compiled into its own separate function in LLVM-IR. A pointer to that function is held in a `vtable`. This function can take arguments even if the closure doesn't, those are the outside values that were captured by the closure using the `move`-keyword.

The closure-pointer in `thread::spawn` is then held in a Box, it is actually a fat pointer, i.e. two pointers. The first points to a struct which holds the arguments being passed to the closure, meaning again the values captured by it in our case. The second pointer points to the `vtable` which holds the function pointer to the closure body. We extend the lifetime of this Box to `'static` as the thread might run longer than the parent thread.

We cannot pass a fat pointer to `__VERIFIER_spawn`, it takes a single `ptr` and thus the second pointer would get lost. To handle this, we need double indirection. We put the fat pointer (represented by a Box), into another Box. We pass to `__VERIFIER_spawn` a pointer to a static function named `thread_start` and the pointer held by the outer Box.

The function `thread_start` handles the dynamic invocation of the closure, which is done through the `vtable`, and the exiting of the thread. To recap, `thread_start` receives a pointer named `main` as argument with the structure:

```
main -> struct { ptr1, ptr2 }
ptr1 -> struct { arguments to the closure }
ptr2 -> vtable -> function code of the closure
```

It again uses Box to build a fat pointer using the raw-pointer `main`, this fat-pointer is then callable. Box also handles the freeing of memory after executing the closure.

Current issues with dynamic dispatch: We face an issue in GenMC when using dynamic dispatching of a function through a `vtable` as is done here. A segmentation fault is signaled while executing the LLVM-IR in the verification phase. This happens because the function pointer GenMC retrieves from the `vtable` holds a wrong address in some instances. We meticulously analyzed the produced LLVM-IR and ensured that it is indeed correct. The exact reason as to why this happens is still an open question as of now. This is not something that can be evaded in the compilation stage since closures in Rust are inherently represented by their `vtables`. Because of this, when possible we used `thread::spawn` with closures and if that failed, we fell back to `thread::spawn_f`, a custom, non-STD function.

3.1.2 Spawning threads without closures: spawn_f

We provide a non-standard, light-weight interface to spawn threads using function-pointers instead of closures. Arguments, if present, are put on the heap and a pointer passed to the function, then freed from the heap at the end. Invocation of the function happens directly through the function pointer, not dynamically through a `vtable`. Like with closures, return-values are put on the heap so they can be retrieved by `.join()` in the exact same way.

This interface does not take ownership like a closure, we can therefore simply share a standard reference (&-syntax) to the same data across multiple threads as none take ownership of the data. This is useful as accessing the data through atomic reference counters will induce more synchronization, causing a higher number of executions. It therefore allows for multiple threads to still access the same data-structure of the client application to be verified, with maximal parallelism.

3.1.3 Joining threads and returning data

The return-value of a thread is put on the heap using Box, we then return a pointer to the value using `__VERIFIER_thread_exit`. That same pointer is retrieved when calling `__VERIFIER_thread_join`, we can again use Box to retrieve the value behind this raw pointer.

Using Box here is necessary as it allows returning non-sized elements, like the unit-type `"()"` for threads returning nothing. Such zero-byte sized types get instantiated by Box without reading from or writing to memory, therefore no unnecessary `malloc`-calls happen in that case either.

3.1.4 Scoped Threads

By utilizing scoped threads, the user can create multiple threads which are all joined at once. This is useful because after the scoped threads return, ownership of the data that was captured by the scoped threads is returned to the original thread, i.e. they borrow instead of taking full ownership. Therefore, all data captured by scoped threads can still be used outside that scope afterwards. This is safe since Rust guarantees that the scoped threads are joined and consequently the threads are finished before ownership is returned, which it cannot guarantee for normal threads.

To do this, scoped threads carry two lifetimes: `'scope` and `'env` representing the lifetime during which the scoped threads may still be running and the lifetime of everything borrowed by threads in the scope, respectively. The `'env` lifetime must outlast `'scope`.

We guide ourselves again by Rust's STD implementation. To join all threads at the end, Rust carries an atomic counter which holds the number of currently still running threads in the scope, spinning until that counter reaches zero. We do the same but use `__VERIFIER_assume` with the condition that the counter reached zero so our executions are finite in length.

We test the functionality of scoped threads by spawning multiple threads inside a scope and using the values that were captured by those threads again after the scope. In one of the tests, we also manually join one of the threads earlier already.

3.2 Atomics

Supporting atomic operations was relatively straight forward, as they are lowered into atomic RMW, store and load instructions [24, 'atomicrmw' Instruction, 'store' Instruction, 'load' Instruction], which GenMC's interpreter already understands and whose execution is already intercepted.

We present a quirk of how atomic operations are realized in Rust compared to Clang using atomic store as an example [28, library/core/src/sync/atomic.rs]:

```
unsafe fn atomic_store<T: Copy>(dst: *mut T, val: T,
  order: Ordering) {
  unsafe {
    match order {
      Relaxed =>
        intrinsics::atomic_store::<T, { AO::Relaxed }>(dst, val),
      Release =>
        intrinsics::atomic_store::<T, { AO::Release }>(dst, val),
      SeqCst =>
        intrinsics::atomic_store::<T, { AO::SeqCst }>(dst, val),
      Acquire => panic!("..."),
      AcqRel => panic!("..."),
    }
  }
}
```

We see that the verification of whether the ordering is valid is implemented to be done at runtime. Rust relies on the compiler optimizations it performs during a release-build (constant propagation / constant folding) to replace this switch statement with just the atomic store (or a panic) since the ordering is usually statically known. Compare that with Clang, which checks this directly during compilation. This handling of atomic operations makes our later changes in the transformation-phase crucial.

3.3 Mutex

The mutex is probably the most fundamental synchronization primitive in parallel computing. It ensures that only one thread can access a shared resource at a time, thus preventing race conditions. All other threads will perceive changes to that shared resource as being executed atomically.

3.3.1 Rust's Standard implementation

The Rust-standard implementation for mutexes does not use the POSIX-interfaces for mutexes like `pthread_mutex_lock`. Rust's mutex instead holds a futex with an atomic variable that can be in any of three states: `UNLOCKED`, `LOCKED` or `CONTENDED`.

We present the algorithm employed by Rust's STD-library for locking followed by some explanations about it [28, library/std/src/sys/sync/mutex/-futex.rs]:

Algorithm 1 Mutex implementation in Rust's STD

```
1:  procedure Mutex::lock(futex)
2:      if ¬futex.compareAndSwap(UNLOCKED, LOCKED, ∅) then
3:          state ← spin
4:          if state = UNLOCKED ∧ futex.compareAndSwap(UNLOCKED,
    LOCKED, state) then
5:              return
6:          end if
7:          while ⊤ do
8:              if state ≠ CONTENDED ∧ futex.swap(CONTENDED) = UN-
    LOCKED then
9:                  return
10:             end if
11:             if state = CONTENTDED then
12:                 futex.wait
13:             end if
14:             state ← spin
15:         end while
16:     end if
17:     return
18: end procedure
19: procedure spin
20:     i ← 100
21:     while i ≠ 0 do
22:         state ← futex.load
23:         if state ≠ LOCKED then return state
24:         end if
25:         i ← i − 1
26:     end while
27:     return state
28: end procedure
```

`futex.swap` and `futex.compareAndSwap` are the atomic-exchange and atomic CAS operations, respectively. The used orderings are `acquire` for `futex.swap` and (`acquire`, `relaxed`) for `futex.compareAndSwap`. In this description, `futex.compareAndSwap` returns a boolean indicating if the operation was successful and saves the current value of the atomic variable to the third argument only if the operation fails.

Note that we return on line 9 because we set a previously `UNLOCKED` state to `CONTENDED`, therefore it was just successfully locked by the current thread.

`SPIN` does a relaxed load, so this algorithm employs techniques of the TTAS-lock [10, Chapter 7] for performance gains on a contended lock. It also emits a hint to the processor (`_mm_pause` in X86_64, see [12, Section 11.4.4.4]) to inform the CPU that the program is in a spin-loop as to allow for further optimization by the CPU directly.

`futex.wait()` blocks the thread until another wakes it up using `FUTEX_WAIT` and `FUTEX_WAKE` system calls [16], no timeout is used.

3.3.2 Our custom implementation

It is not hard to see that Algorithm 1 is doing far too much for our application. It may employ techniques to prevent starvation on a contended lock, but such an implementation would slow down formal verification using GenMC due to the amount of interleavings it produces, when all we actually care about is which of the threads in the actual client program using the mutex is successful in acquiring the lock, and which is blocked. This algorithm could by itself be verified by GenMC for correctness,
which we will not do as we do not support futexes yet in this work.

For our application we assume correctness in Rust's STD-library, and want to prove correctness in the client application. We care about behaviors induced by the client application. Thus, we introduce a new, significantly simpler implementation for mutex in our genmc-std crate which uses GenMC's built-in system-primitives for mutexes, such that our mutex can induce all the same behavior in the client application as with the usual implementation and allow GenMC to explore all interleavings in their code. These system-primitives are meant to replicate the behavior of `pthread_mutex_*` primitives in POSIX and are called similarly: `__VERIFIER_mutex_*`.

POSIX/GenMC Primitive	**Description** [9]
*_mutex_t	Struct identifying the mutex.
*_mutexattr_t	Struct to set the mutex-type.
*_mutex_init	Initializing the *_mutex_t struct.
*_mutex_destroy	Uninitializing the mutex.
*_mutex_lock	Blocking acquire of the mutex-lock.
*_mutex_trylock	Non-blocking acquire of the mutex-lock.
*_mutex_unlock	Release the mutex-lock.

Table 3.1: POSIX Mutex primitives

We use the definitions for `__VERIFIER_mutex_*` in GenMC's C-header file "genmc.h" with bindgen [27], a Rust-crate that, given a C-header file, automatically generates Rust bindings to externally defined C-functions with FFI-compliant types.

When locking a mutex in Rust [31, std::sync::Mutex], the user receives a MutexGuard object that allows the user to access the value protected by the mutex. The mutex is automatically unlocked when the MutexGuard goes out of scope. Our Mutex & MutexGuard implement all the same traits as in Rust's STD-library, they can be hot-swapped in a client program as it fulfills the exact same type-requirements. We now describe the implementation of the different functionalities of the mutex.

Mutex Initialization

`Mutex::new` is a constant function in Rust, and we want to keep it that way such that we can initialize a static, global mutex. Thus, we cannot call `__VERIFIER_mutex_init` here, as it is an external function. Constant functions must be pre-computable. We realized that initialization of the mutex is not required as we are using the default mutex-type (same as normal mutex type in Linux-systems). Initialization to a default mutex is done by GenMC on the first lock as is also done in Linux if no initialization with attributes is done.

Acquiring the Mutex-Lock

Our `Mutex::lock` implementation simply calls `__VERIFIER_mutex_lock`. The status returned by this call is not used, as GenMC currently does not model failures. We can therefore assume that locking always succeeds and return a MutexGuard.

Releasing the Mutex-Lock

The MutexGuard implements Rust's Drop-trait. Thus, the mutex gets automatically unlocked when the MutexGuard returned by `Mutex::lock` goes out of scope. We achieve that by calling `__VERIFIER_mutex_unlock` in our drop-function.

Poisoning of the lock

A mutex in Rust is considered poisoned if a thread panics while holding the lock. This means the data protected by the mutex might be in an inconsistent or invalid state. The next thread trying to acquire the lock will get an error. If handled properly, the data could still be recovered using the IntoInner-trait [31, std::sync::Mutex]. Since GenMC does not model failures, a mutex in our implementation is never poisoned and clearing the poison-indicator is a no-op.

Mutex TryLock

Our implementation for `Mutex::try_lock` simply calls `__VERIFIER_mutex_trylock`, using its returned value this time to signal to the user if the lock was successful or not. However, during testing of our implementation in Rust, an error was found in GenMC's implementation of `__VERIFIER_mutex_trylock` which would make it not usable. Calling `__VERIFIER_mutex_trylock` on an already locked mutex caused a segmentation fault during the verification phase. This issue was resolved and will be presented in more detail in Section 5.1.

Verifying our implementation

In addition to the new C-tests we later provide for trylock in Section 5.1, we provide tests with stronger focus on the Rust-specific aspects of mutexes. This includes modification of the value that is protected by the mutex through its guard and automatic unlocking of the mutex when its guard goes out of scope. Our tests run as expected.

3.4 Memory Allocator

Rust's LLVM-IR output contains calls to externally defined functions:

1. `__rust_alloc`
2. `__rust_dealloc`
3. `__rust_realloc`
4. `__rust_alloc_zerored`

Which are Rust's equivalents to C's malloc, free, realloc and calloc, respectively. They perform the right system-calls for the OS we're compiling for. These functions are built into the compiler, they have no function-body and thus external function declarations in the LLVM-IR. Therefore, when they get called GenMC will throw an error: "Tried to execute an unknown external function: __rust_alloc".

These functions are defined in [28, alloc/src/alloc.rs]. All they do is forward their calls to Rust's standard global allocator [31, std::alloc::GlobalAlloc] defined by the system [31, std::alloc::System].

The Rust language provides the `#[gloabal_allocator]` attribute ([31, std::alloc::GlobalAlloc], [30, 20 The Rust runtime]) to overwrite the standard global allocator, calls to `__rust_alloc` & co. are then forwarded to that custom allocator instead of Rust's default system global allocator. We cannot use this as this forwarding is still done in the Rust compiler, therefore there is again no LLVM-IR implementation for `__rust_alloc`.

We need to provide implementations for `__rust_alloc` & co. ourselves. We define our own functions `genmc__rust_alloc` & co. with the exact same function signatures. Then, during the transformation phase, we overwrite all calls to `__rust_alloc` & co. to call `genmc__rust_alloc` & co. instead. In `genmc__rust_alloc` & co. we provide our custom implementation for memory allocation through GenMC's internal functions (`__VERIFIER_malloc` & co.), using Rust's global allocator for Linux systems as a reference.

3.5 The Rust Frontend

In this section we describe the compilation process for taking a Rust-file (*.rs) or a Cargo project-file (Cargo.toml) as an input and retrieving the produced LLVM-IR, which can then be used in the transformation and verification phases. We perform a debug-build instead of a release-build since debug-information is required in the LLVM-output for later presentation to the user. We also consider the release-build too heavily compiler-optimized for our purpose of exploring all executions, causing some problematic executions we want to find to possibly be "optimized away" in such a build.

Manpage-style synopsis for GenMC:

```
genmc [OPTION]... [FILE]
genmc [OPTION]... -- [CFLAGS]... [FILE]
```

CFLAGS are passed to Clang during compilation. We extend this to "compiler flags", meaning for Rust-input we forward those options to Rustc / Cargo.

To detect the input-language, we look at the input file name. For Cargo.toml-files, we compile as a cargo project. If the file extension is ".rs", we compile as a single Rust-file and for ".ll" we have a direct LLVM-IR input and thus move directly to the transformation phase, this replaces the old command-line option "-input-from-bitcode-file". If none of the above is the case, we treat it as a C/C++-input by default and invoke Clang.

3.5.1 Compiling Rust input

When using GenMC with a single *.rs input file, we compile in two steps. First, we '`cargo clean`' and '`RUSTFLAGS="--emit=llvm-bc" cargo build`' the genmc-std crate separately. This emits the LLVM-BC (as bitcode) of the build and produces a Rust library file named `libgenmc_std.rlib` that we can include in other builds [30, 15 Linkage]. This step can be skipped using the option "`-disable-genmc-std-rebuild`" if a build already exists.

We can now use `libgenmc_std.rlib` to include genmc-std into the external prelude of every *.rs file being compiled with rustc, replacing the Rust's

pre-built STD that's already in the prelude as described in the introduction of this chapter.

```
rustc --emit=llvm-bc -Copt-level=0 --extern std=libgenmc_std.rlib
 -o rustc_out.bc path/to/inputFile
```

After successfully compiling our input file with genmc-std as the used standard library, we can now iteratively parse and link all the produced LLVM-BC files from both the Cargo build of genmc-std as well as the Rustc build of our input-file into a single, complete module using LLVM's Linker.

3.5.2 Compiling Cargo input

When using GenMC with a Cargo project, the Cargo project must have genmc-std as a dependency in its Cargo.toml like this:

```
[dependencies]
std = { path = './path/to/genmc-std', package = 'genmc-std'}
```

This ensures that Cargo includes genmc-std in the external prelude of the crate just like we do when compiling Rust input. We enter the given Cargo projects directory, where we first execute `'cargo clean'` to delete all files from previous builds and then `'RUSTFLAGS="--emit=llvm-bc" cargo build'`. This runs a full build of the project while emitting the LLVM-BC (as bitcode) of each Rust-file in the project into the target-folder. It is not possible to stop a Cargo-build directly after the phase that generates the LLVM-BC like it is done in Clang.

To retrieve a single, complete LLVM module that contains the code for the entire Cargo-project, we collect all "*.bc" files in the target-directory then parse and link each of them iteratively into a single module using LLVM's Linker.

3.5.3 –link-with=[FILE] option

This argument we newly added allows to separately compile the file given in the argument and once it and the regular input-file are compiled, link them together in LLVM into a single module. As an example later used in the evaluation, we have an input-file in C "Most parallel Client", which invokes functions defined in the Rust-file queue.rs. Both mpc.c and queue.rs are compiled separately using Clang and Rustc, respectively.

```
genmc --link-with=queue.rs mpc.c
```

This option further allows to cover verification of programs spanning FFI-calls between C and Rust more generally, which can be a major cause of issues due to Rust not being able to enforce its strong type system across language boundaries. Allowing projects to do a secure, step-wise migration from C to Rust.

3.5.4 Determine Rust versions

We link GenMC to a specific Rust-installation when first compiling GenMC with CMake. The user can indicate the installation path using the new "`-DRUST_BIN_PATH=[FILE]`" option. The Rust compiler uses its own distribution of LLVM. To ensure compatibility, we check if the Rust-installation the user specified uses the same LLVM major version as GenMC is being compiled with and reject the provided Rust-installation if the versions do not match. Which LLVM-version the Rust compiler uses can be checked with "`rustc -vV`".

3.5.5 GenMC's custom interfaces

The headers genmc.h and genmc_internal.h provide public and private interfaces for interacting with the verifier, respectively. All `__VERIFIER_*`-functions mentioned so far are a part of these headers.

We translated all functions, types constants and macros in these files into Rust and expose the public interfaces in our genmc-std crate while using the internal functions in our own implementations.

Interfaces to the verifier, i.e. functions in genmc_internal.h, are marked with the #[no_mangle]-attribute such that the verifier can recognize them during execution. These interfaces should be marked as external functions in the LLVM-IR, but when building with Cargo we cannot call an externally defined function without Cargo trying to link these symbols and failing since it cannot find them. This cannot be disabled in Cargo. We instead mark GenMC's internal functions as externally defined in the LLVM-IR during the transformation phase.

Chapter 4

Transformation

Each of the additions to the LLVM passes was made to solve specific problems that were faced while realizing the other sections. We describe the implementation details in this chapter separately here.

4.1 Lowering arithmetric with overflow intrinsics

We found that Rust will use LLVM's with overflow intrinsics [24, Arithmetric with Overflow Intrinsics] to perform the binary operations *addition*, *subtraction* and *multiplication*, even if we do not check for overflows. These intrinsics return both the result of the binary operation as well as a bit indicating whether or not the operation overflowed. Handling these intrinsics in GenMC is crucial, without them we would not be able to verify even very basic models if they use addition, subtraction or multiplication. One example would be the implementation of a loop-counter.

In this section we describe the lowering of the following LLVM intrinsics into standard LLVM instructions executable by GenMC's interpreter [24, Arithmetric with Overflow Intrinsics]:

1. `llvm.sadd.with.overflow.{ i8 | i16 | i32 | i64 }`
2. `llvm.uadd.with.overflow.{ i8 | i16 | i32 | i64 }`
3. `llvm.ssub.with.overflow.{ i8 | i16 | i32 | i64 }`
4. `llvm.usub.with.overflow.{ i8 | i16 | i32 | i64 }`
5. `llvm.smul.with.overflow.{ i8 | i16 | i32 | i64 }`
6. `llvm.umul.with.overflow.{ i8 | i16 | i32 | i64 }`

These represent the binary operators *addition*, *subtraction* and *multiplication*, each of which has a *signed* and *unsigned* variant for the data-types `i8`, `i16`, `i32`

and `i64`. The result is a `struct {i*, i1}` where the first field is the wrapped result of the binary operation while the second field indicates whether or not the operation overflowed during execution. In normal compilation, these intrinsics would be lowered into optimized assembly for the specific target architecture.

```
declare {i32, i1} @llvm.sadd.with.overflow.i32(32 %a, i32 %b)
```

We'll present the general strategy first, then we discuss how it can be used to lower all variants of the intrinsic. Note that some inspiration for our solution was taken from KLEE's *IntrinsicCleaner* [26], although it was significantly improved upon and the implementation heavily simplified. Unlike KLEE, we do not need to know the maximum or minimum values for any of the integer types or differentiate between *signed* and *unsigned* cases in order to determine overflows.

Computing the result: We can easily determine the value for the first field in the returned struct by simply performing the binary operation, e.g. a standard `add` [24, 'add' Instruction]. This gives us the result of the operation, wrapped to fit into the given bit-length in case an overflow occurred.

Determining if an overflow occurred: This is the trickier part. To this end, we perform the same arithmetic operation with the same values, but with an extended bit-length large enough such that any resulting value would fit in this extended bit-length without overflowing. Then we compare the values of those two results and if they are not the same, we know that an overflow occurred. The full strategy:

1. Perform the operation with normal operands $\rightarrow$ result
2. Extend the operands to twice their bitlengths
3. Perform the operation again on extended operands $\rightarrow$ result_from_ext
4. Extend result to twice its bit-length as well $\rightarrow$ result_ext
5. Set overflow $= 1$ iff result_from_ext $\neq$ result_ext
6. Return `struct {result, overflow}`

Extending our approach to the entire with.overflow.* family: We can follow the exact same strategy outlined above for all our cases, but we do need to be careful to perform the correct operations in each case.
In steps 1 and 3 we need to perform the correct binary operation (`add` / `sub` / `mul`), always without any `nsw` or `nuw` flags [24, Binary Operations].
In steps 2 and 4 we need to zero-extend [24, 'zext .. to' Instruction] in the *unsigned* case and signed-extend [24, 'sext .. to' Instruction] in the *signed* case in order for the result to semantically represent the same value as before.

To keep our code simple and understandable, we've written the complete strategy into a macro while passing the right operations for the steps 1-4 as parameters to the macro.

Optimization: In all cases we observed, the resulting struct as a whole is not directly used by the LLVM-IR generated from Rust, it is only used to extract the values inside of it. This means that the following standard LLVM-IR output from a simple addition:

```
%0 = call { i32, i1 } @llvm.sadd.with.overflow.i32(i32 %_argc,
  i32 123)
%_5.0 = extractvalue { i32, i1 } %0, 0
%_5.1 = extractvalue { i32, i1 } %0, 1
```

Will be transformed into:

```
%0 = add i32 %argc, 123
%1 = sext i32 %argc to i64
%2 = add i64 %1, 123
%3 = sext i32 %0 to i64
%4 = icmp ne i16 %2, %3
%5 = insertvalue { i32, i1 } undef, i32 %0, 0
%6 = insertvalue { i32, i1 } %5, i1 %4, 1
%_5.0 = extractvalue { i32, i1 } %6, 0
%_5.1 = extractvalue { i32, i1 } %6, 1
```

Therefore, we have two `insertvalue` instructions to build the resulting struct of the intrinsic, just for the struct to be decomposed again using `extractvalue` instructions. This gives us a great opportunity for optimization. We analyze all uses of the intrinsic instruction and replace all uses of the `extractvalue` instructions to use the result we computed directly instead of the value extracted from the struct. Once all those uses are replaced, we can safely remove the `extractvalue` instructions. If the intrinsic instruction was used only by those `extractvalue` instructions, we can omit the `insertvalue` instructions completely as no struct is needed. This eliminates four instructions in the default case by having to emit neither `insertvalue` instructions nor `extractvalue` instructions. In the above example, any use of `%_5.0` is replaced with `%0` and `%_5.1` with `%4`.

Verifying our implementation: We created tests for all variants of the intrinsics in all data-types. For each, we have a test where the computation barely overflows or barely does not, comparing with our expected results. A set of a representative examples for testing our implementation is deferred to the appendix in Section A.1.

4.2 Lowering LLVM's Memcpy Intrinsic

In this section we discuss changes made to the promotion of LLVM's `memcpy` intrinsic [24, 'llvm.memcpy' Intrinsic]:

```
declare void @llvm.memcpy.p0.p0.i32(ptr <dest>, ptr <src>,
                                    i32 <len>, i1 <isvolatile>)
declare void @llvm.memcpy.p0.p0.i64(ptr <dest>, ptr <src>,
                                    i64 <len>, i1 <isvolatile>)
```

This intrinsic copies the specified number of bytes in `len` from the `src` pointer to the `dest` pointer. GenMC will try to lower calls to this intrinsic into a series of `load` and `store` instructions instead. This lowering pass required significant extensions for it to work with Rust as many use-cases were not supported because they were not or very rarely applied by the Clang compiler.

It was of utmost importance for this to work correctly. Rust's Box [31, std::boxed::Box] is used for anything related to managing heap-allocated data and it requires this intrinsic. Boxes are heavily used in client applications and in many places in Rust's STD directly. For example, Box is used in the implementation of atomic reference counters [31, std::sync::Arc], which are used for sharing data between multiple threads. As we saw in Section 3.1, Box is also needed in our implementation for spawning and joining threads. Thus, Box must work for even some of our basic goals to be achievable.

How does the original implementation work?
Assuming the pointee-type of `src` and `dest` are the same, it would recursively index into the pointee-type of `dest` until it reaches a primitive (non-composite) type. Then, it generates `getelementptr` instructions [24, 'getelementptr' Instruction] using the computed indexes to reach said primitive type, possibly nested inside multiple composite types. Using the addresses retrieved by the `getelementptr` instructions, it will generate `load` instructions from the `src` and corresponding `store` instructions to `dest` in order to, one by one, copy all fields from `src` to `dest`. Here's a simple example:

```
%datatype1 = type { i32, i64 }
%datatype2 = type { i32, i64 }
define internal void @main() {
start:
  %_5.i = alloca %datatype1, align 8
  %_4.i = alloca %datatype2, align 8
  call void @llvm.memcpy.p0.p0.i64(ptr align 8 %_4.i,
    ptr align 8 %_5.i, i64 12, i1 false)
```

```
  ret void
}
```

Transformed output:

```
%datatype1 = type { i32, i64 }
%datatype2 = type { i32, i64 }
define internal void @main() {
start:
  %src_ptr = alloca %datatype1, align 8
  %dst_ptr = alloca %datatype2, align 8
  %memcpy.src.gep = getelementptr inbounds %datatype1,
    ptr %src_ptr, i64 0, i32 0
  %memcpy.dst.gep = getelementptr inbounds %datatype2,
    ptr %dst_ptr, i64 0, i32 0
  %memcpy.src.load = load i32, ptr %memcpy.src.gep, align 4
  store i32 %memcpy.src.load, ptr %memcpy.dst.gep, align 4
  %memcpy.src.gep1 = getelementptr inbounds %datatype1,
    ptr %src_ptr, i64 0, i32 1
  %memcpy.dst.gep2 = getelementptr inbounds %datatype2,
    ptr %dst_ptr, i64 0, i32 1
  %memcpy.src.load3 = load i64, ptr %memcpy.src.gep1, align 4
  store i64 %memcpy.src.load, ptr %memcpy.dst.gep2, align 4
  ret void
}
```

In each of the following subsections, we present the new use-case we need to support for Rust and how we extended the existing pass to support it.

Some of ideas are motivated by the fact that a lot of the code in this pass is shared with the implementation for lowering LLVM's `memset` intrinsic [24, 'llvm.memset.*' Intrinsic], whose behavior should not change by our extensions for `memcpy`.

4.2.1 Differently structured pointee-types in src and dest

Rust will create `memcpy`'s between a `src` and `dest` which are of different data-types, but are still compatible with each other.

Compatible: All primitive types nested inside the composite type have the same bit-lengths and the same offsets from the starting address, they are just wrapped into (possibly multiple) different composite types. Primitive types may be ptr, i8, i16, float, double and so forth.

This means that we can copy between `src` and `dest` without causing mixed-sized integer accesses, which is our intended goal. Here is a minimal LLVM-IR output from Rust as an example:

```
%"PtrComponents<u64>" = type { ptr, {} }
%"PtrRepr<u64>" = type { [1 x i64] }
define internal void @main() {
start:
  %_5.i = alloca %"PtrComponents<u64>", align 8
  %_4.i = alloca %"PtrRepr<u64>", align 8
  call void @llvm.memcpy.p0.p0.i64(ptr align 8 %_4.i,
    ptr align 8 %_5.i, i64 8, i1 false)
  ret void
}
```

The two types are compatible with one another as a pointer in this environment is also 64-bits wide, both the inner `ptr` in `PtrComponents<u64>` and the `i64` from `PtrRepr<u64>` have no offset from the starting address.

The original implementation assumed that the source and destination pointers will hold data of the exact same type, thus outputting the following faulty transformed LLVM-IR:

```
%"PtrComponents<u64>" = type { ptr, {} }
%"PtrRepr<u64>" = type { [1 x i64] }
define internal void @main() {
start:
  %_5.i = alloca %"PtrComponents<u64>", align 8
  %_4.i = alloca %"PtrRepr<u64>", align 8
  %memcpy.src.gep = getelementptr inbounds %"PtrComponents<u64>",
    ptr %_5.i, i64 0, i32 0, i32 0
  %memcpy.dst.gep = getelementptr inbounds %"PtrRepr<u64>",
    ptr %_4.i, i64 0, i32 0, i32 0
  %memcpy.src.load = load i64, ptr %memcpy.src.gep, align 4
  store i64 %memcpy.src.load, ptr %memcpy.dst.gep, align 4
  ret void
 }
```

The problem here is the indexing into `%_5.i` of type `PtrComponents<u64>`, i.e. the `src` pointer. The original implementation always used the pointee-type of `dest` to generate the indexing to use with `getelementptr`, which is correct if `src` and `dest` are of the same type. In this example, the `i64` from `dest` is inside an array, whereas the `ptr` from `src` is not. Therefore it will try

to index further into the `ptr` type as it assumes it is an array as well, leading to an indexing error as we cannot index further into a primitive type.

Since we are dealing with **compatible** data-types, all we require is that the `getelementptr` instructions on both `src` and `dest` computes the same offset from the starting address for each field we want to copy over. We make use of this by introducing a notion of "casting" for our case.

Casting: Refers to how this LLVM pass finds which pointee-types to use in the `getelementptr` instructions it inserts. Because LLVM uses opaque pointers (meaning their pointee-type is not defined for any pointer), the original implementation will look for the instruction defining the pointer-value to detect the appropriate pointee-type. If the pointer is defined by an `alloca` instruction, as is the case in the above example, it will use the allocated data-type. If the pointer is defined by another `getelementptr` instruction, it will use the result-element-type defined there.

We insert a `getelementptr` instruction to "cast" our `src` into the same data-type as `dest` with an index of just zero, this keeps the same address pointed to by `src`.

This way, the rest of the original implementation will use the same data-type in the `getelementptr` instructions it inserts for both `src` and `dest`, eliminating our issues with indexing. Here's the new transformed output after our changes:

```
%"PtrComponents<u64>" = type { ptr, {} }
%"PtrRepr<u64>" = type { [1 x i64] }
define internal void @main() {
start:
  %_5.i = alloca %"PtrComponents<u64>", align 8
  %_4.i = alloca %"PtrRepr<u64>", align 8
  %0 = getelementptr %"PtrRepr<u64>", ptr %_5.i, i32 0
  %memcpy.src.gep = getelementptr inbounds %"PtrRepr<u64>",
    ptr %0, i64 0, i32 0, i32 0
  %memcpy.dst.gep = getelementptr inbounds %"PtrRepr<u64>",
    ptr %_4.i, i64 0, i32 0, i32 0
  %memcpy.src.load = load i64, ptr %memcpy.src.gep, align 4
  store i64 %memcpy.src.load, ptr %memcpy.dst.gep, align 4
  ret void
 }
```

UID `%0` is our newly inserted `getelementptr` instruction which "casts" `%_5.i` from `PtrComponents<u64>` into `PtrRepr<u64>`.

The effect of this newly inserted instruction is that for `src` we now use the same type as for `dest`, notice how `%memcpy.src.gep` now uses `PtrRepr<u64>`

instead of `PtrComponents<u64>`. Consequently, it is now not a problem that the indexes for `getelementptr` are generated using only `dest`, because we are using the same type in the `getelementptr` to compute the offset from `src` too.

This approach generalizes superbly for all possible cases because we can treat compatible data-types as being the same. It is also much simpler than the alternative, which would be generating separate `getelementptr`-indexing for both `src` and `dest` for each field being copied.

There are cases where we cannot cast `src` $\rightarrow$ `dest`, for example when the `dest`-pointer was retrieved by a *malloc*-call and therefore has no known pointee-type. This can happen when using Box to allocate a struct onto the heap in Rust. Then, we try casting `dest` $\rightarrow$ `src` instead.

For full correctness, we use integers of the index size in the environment we're working in.

Supporting older LLVM-versions without opaque pointers: Older LLVM-versions (<15) do not use opaque pointers. By default, every pointer holds the pointee-type. This limits the operations we can perform on those pointers to use the same type as the pointee-type always. This makes "casting" using a `getelementptr` impossible owing to the requirement that the same type must be used in the instruction as the pointee-type already is.

To allow for the same approach as before, we use `ptrtoint` [24, 'ptrtoint .. to' Instruction] and `inttoptr` [24, 'inttoptr .. to' Instruction] instructions in LLVM <15 to get an opaque, non-restricted pointer, which can then be "casted" into the right type using a `getelementptr` instruction as before. We use an integer type of the same bit-size as a `ptr` in the given environment.

4.2.2 Copy smaller into larger data-type

The original implementation assumed that we're always copying the full type, it therefore did not take the `len`-argument into account at all.

To allow for copying only a part of the full type, i.e. when copying a smaller array into a larger array, we keep track of the sum of bit-sizes of all primitive types we already created `load` and `store` instructions for and stop when that sum equals the `len`-argument.

We abort early if the `len`-argument is larger that the store size of the type being copied from (this would be a buffer-overflow), or while processing if the byte-size of the next primitive-type being copied is larger than the remaining byte-length to be copied (this would cause mixed-sized integer accesses). Both these cases would happen if we pass a `len`-argument of seven in the earlier examples instead of eight. Both errors must have been

induced by the user using `memcpy` directly, we have not observed Clang or Rustc generating such obviously incorrect instructions.

4.2.3 Zero-len Memcpys

When using Box with a zero-sized type in Rust, e.g. the unit type `()` or PhantomData [31, std::marker::PhantomData], Rust will still create a `memcpy`-instruction with a `len`-argument of zero. This is a no-op and thus we remove those instructions completely.

Supporting this case is required for running threads that do not have a return-value, as they will return the unit type by default.

Please note that in this case of zero-sized types, Box would still not invoke `malloc` at all. Therefore, we do not have unnecessary memory allocation calls.

4.2.4 Constant-Expression Arguments

We observed that sometimes, `src` or `dest` would be passed as constant expression arguments. For example:

```
call void @llvm.memcpy.p0.p0.i64(
  ptr align 8 getelementptr inbounds (%struct.queue_t,
    ptr @queue, i64 0, i32 0, i32 1),
  ptr align 8 %_3, i64 8, i1 false)
```

To handle this, we first move those constant expression arguments into their own instructions, then proceed as normal.

4.2.5 Supporting Fortified Variants

On MacOS, we observed the use of the fortified variant of `memcpy`, which is enabled by default for Apple's Clang.

Specification [7]:

```
__memcpy_chk(void * dest, const void * src, size_t len,
  size_t destlen);
```

This variant checks if the given `len` is shorter than the `destlen` before performing the operation. We transform the call to `__memcpy_chk` to perform said check and the calling of LLVM's `memcpy` intrinsic before proceeding as normal, lowering that newly inserted `memcpy` intrinsic to `load` and `store` instructions as described before.

We use the following implementation from glibc implemented in C as a reference to insert equivalent LLVM-code directly in-place [6]:

```
if (destlen < len)
      __chk_fail();
return memcpy(dest, src, len);
```

Instead of `__chk_fail` we will call `__VERIFIER_assert_fail()` to also inform the user. The `memcpy`-call in the above reference implementation refers to the POSIX-compliant `memcpy` system call [9, memcpy]. In POSIX, `memcpy` will return the same value as the `dest` argument, whereas LLVM's `memcpy` intrinsic returns void. Because we are inserting LLVM's `memcpy` intrinsic instead, we replace all uses of the return value of `__memcpy_chk` to use `dest` directly.

In most cases we observed, `destlen` was retrieved by a call to LLVM's `objectsize` intrinsic [24, 'llvm.objectsize' Intrinsic]. Because of that, we also lower this call using LLVM's built-in memory tooling.

This implementation of the fortified variant directly in LLVM works great as the call to `llvm.memcpy` will still be lowered into `load` and `store` instructions, while also only being executed if the check of lengths succeeds, as intended.

4.3 FunctionInlinerPass

The *FunctionInlinerPass* attempts to inline all function calls where inlining is sound. This may increase overall code-size, but allows for better realization of later transformation passes like `Mem2Reg`, Section 4.4.

We first detail the original *FunctionInlinerPass*-implementation, then we discuss different reasons as to why it was problematic in the context of Rust and for each of these reasons, we present our solution for a simpler *FunctionInlinerPass* which achieves our intended goals.

4.3.1 Original Implementation

Below we present the basic working of the original *FunctionInlinerPass*. Here are some explainers:

The check on line 13 represents the base case for the recursive check in the "isInlinable" procedure. The check if the stack already contains the current function is there in order to detect recursive functions and mutually recursive cycles in the call-graph.

CalledF in F means any function CalledF that is being called in the function body of F.

IsInlinable is a recursive check, meaning for every function F in the module it iterates through all other functions that are reachable from F in the call graph for the module, and checks for each of them if they are inlinable as well.

Algorithm 2 FunctionInlinerPass

```
1:  procedure FunctionInlinerPass(Module M)
2:      for all Function F ∈ M do
3:          if isInlinable(F) then
4:              for all CallInstruction CI ∈ M do
5:                  if CI calls F then
6:                      InlineCall(CI, ∅)
7:                  end if
8:              end for
9:          end if
10:     end for
11: end procedure
12: procedure isInlinable(Function F, Stack L)
13:     if F contains indirect calls ∨ F contains calls to external functions ∨
        L.contains(F) then
14:         return ⊥
15:     end if
16:     L.push(F)
17:     for all CalledF in F do
18:         if ¬ isInlinable(CalledF, L) then
19:             return ⊥
20:         end if
21:     end for
22:     L.pop
23:     return ⊤
24: end procedure
```

4.3.2 Adapting it for Rust

We found that almost no function was being inlined in the LLVM-IR generated by Rust. This was due to two aspects:

Invoke Instructions

Rust extensively uses `invoke` instructions instead of `call` instructions. An `invoke` allows for unwinding to an error handling block incase an error occurs during the execution of a function. This is fundamental in Rust, it extensively uses so-called results [31, std::result::Result] and expects each of them to be handled by the user or unwinding further.

An invoke-instruction might look like this [24, 'invoke' Instruction] :

```
%res = invoke i1 my_func(i32 %my_arg1, i32 %my_arg2)
  to label %my_normal_dest unwind label %my_exception_dest
```

Where `my_func` takes two arguments of type `i32` and returns a value of type `i1`. If execution succeeds as expected, indicated by the callee returning with a `ret` instruction, we jump to the normal destination block. Otherwise, we jump to the unwind destination block of the latest `invoke` instruction in the call-stack. This means if `my_func` would call another function F with a `call` instruction, and F returns using a `resume` instruction, we would enter the `my_exception_dest` basic block in the above example.

LLVM's built-in tools already support inlining `invoke` instructions. So we used them on `invoke` like it was already done for `call`.

Very restrictive base-cases for the "isInlinable"-check

Looking at Algorithm 2, the base case on line 13 deems a function not-inlinable if, in its own function body, it performs an indirect call or a call to an external function.

An indirect call also includes calls to an inline assembly block. The following is an example of a compiler-level memory barrier used by Rust:

```
call void asm sideeffect "", "~{memory}"()
```

The check for calls to external functions includes calls to GenMC's internal functions (e.g. `__VERIFIER_thread_create`), as those are marked as externally defined functions in the produced LLVM-IR.

We deem those limitations too restrictive. Why should a function *F*, which includes indirect calls or calls to external functions, not be allowed to be inlined into a parent function *P*? Those same function calls can be performed in the parent function *P* as well. Those checks were removed after a consultation with the advisor as well as verifying that all preexisting tests still perform as expected after this change.

Back-propagation of the "not-inlinable" property in the call-graph

Looking again at the base case at line 13 in Algorithm 2, we see that the check for inlinability of a function F is performed recursively, checking all potential callees from F as well. This means that a function F is only deemed inlinable if every other function reachable from F is also considered inlinable.

We look at the following call-graph where f_n is a function and $\rightarrow$ indicates a call to another function:

$f_1 \rightarrow f_2 \rightarrow f_3 \rightarrow f_4 \rightarrow f_5 \rightarrow f_4 \rightarrow f_5$

Here, f_4 and f_5 are part of a mutually recursive cycle. Thus, f_3 and f_2 are also considered not inlinable as f_4 is reachable from them. So nothing is inlined.

This turned out to be problematic in Rust due to how the panicking-mechanism is implemented. We found that every use of the `panic!` macro [31] will generate the following LLVM-IR, note that the function-names are mangled:

```
call void @_ZN4core3fmt9Arguments6new_v117hbcce7c664e34cc69E(...)
call void @_ZN4core9panicking9panic_fmt17h1933f3dc606f9d8fE(...)
unreachable
```

The function `core::fmt::Arguments::new_v1` will use the `assert!` macro [31] to validate the arguments to the function. The `assert!` macro will panic if the assertion fails and generate another call to `core::fmt::Arguments::new_v1`. Therefore, `core::fmt::Arguments::new_v1` is actually a recursive function. See the function implementation below [28, library/core/src/fmt/rt.rs]:

```
pub fn new_v1<const P: usize, const A: usize>(
        pieces: &'a [&'static str; P],
        args: &'a [rt::Argument<'a>; A],
    ) -> Arguments<'a> {
        const { assert!(P >= A && P <= A + 1, "invalid args") }
        Arguments { pieces, fmt: None, args }
    }
```

Due to panicking and unwinding being such a central part of the Rust programming language and also very extensively used in Rust's STD-library, this rendered almost every function to be considered "non-inlinable" by the original *FunctionInlinerPass*.

This heavy use of panicking can be seen in Section 3.2, where the switch-statement generated by every atomic store operation contains a `panic!`.

Allowing more inlining: The main idea of the back-propagation of the "not-inlinable" property was to prevent inlining (mutually) recursive functions, this is the only base case at line 13 in Algorithm 2 that looks at the stack containing the already visited functions. Our goal is to allow the inlining of more function calls while not inlining recursive functions or functions which are part of a mutually recursive cycle. In our previous call-graph example, this would mean inlining f_2 and f_3 into f_1, but not inlining f_4 or f_5. It is

important to not inline these mutually recursive functions as this may lead to an infinite loop of inlining functions into one another.

To this end, we removed the original "isInlinable" check completely as the check for (mutually) recursive cycles is the only base case left after our previous changes. We also found that LLVM's built-in function-inliner already prevents infinite loops of inlining the same functions into one another. After a consultation with the advisor, it was decided to not rely on this mechanism due to it not being a well documented behavior.

Instead, we use LLVM's built-in tools for finding strongly connected components (SCCs) using Tarjan's Algorithm [22]. All functions which are not part of a SCC that contains a **cycle** are considered inlinable.

A SCC contains a cycle if it has a size greater than one (and therefore represents a mutually recursive cycle in the call-graph), or if the single function in the SCC is itself recursive [23, llvm::scc_iterator::hasCycle].

4.4 Mem2Reg Promotion & SROA

LLVM's *PromotePass* [25, mem2reg], commonly known as *mem2reg*, transforms stack-allocated memory (with `alloca` instructions) into SSA-value registers instead. It helps subsequent optimizations and code analysis, but also GenMC's performance during the verification phase as it needs to keep track of fewer reads and writes to memory.

In the Rust case, many `alloca` instructions were not being promoted. This caused significant problems, especially when using Box. With Box, we often observed `load` instructions from allocated memory without there being a `store` instruction to that memory region beforehand, therefore causing GenMC to find a "Reads to uninitialized memory" error during verification. However, the values of these reads from uninitialized memory were not even being used, meaning if the `alloca` instruction were to be successfully promoted by *mem2reg*, these loads would not exist.

Looking into LLVM's codebase we found that the reason so many `alloca` instructions were not promoted is because *mem2reg* will only promote `alloca` instructions that are exclusively used in `load` and `store` operations. In Rust's case, most `alloca` instructions are struct-types and it therefore requires using `getelementptr` instructions to access the individual fields inside.

Our solution is to use the *SROA* pass (Scalar Replacement of Aggregates) [25, sroa] before *mem2reg*. *SROA* dissects a struct into the individual fields of primitive types, and generates `alloca` instructions for each of these fields to replace the `alloca` of the entire struct. It then eliminates the `getelementptr` instructions on the original struct and replaces them with the use of the `alloca`'d primitives directly. After *SROA*, *mem2reg* can successfully promote

these stack-allocated values as they are of primitive types now, eliminating the reads to uninitialized memory.

This goes hand-in-hand with the FunctionInlinerPass in Section 4.3, since *mem2reg* is a function-level pass. Meaning if the pointer to a stack-allocated value is passed as a function argument, that stack-allocated value cannot be promoted, even if it is of primitive type after *SROA*. The new FunctionInlinerPass also ensures that maximally many such stack-allocated values can be promoted.

Rust generally considers reads from uninitialized memory as undefined behavior [30, 17.2 Behavior considered undefined], it is explicitly allowed when using MaybeUninit [31, std::mem::MaybeUninit].

4.5 RustPrepPass

This pass was newly added to handle all the small transformations needed to prepare our Rust-generated LLVM-IR for further processing.

4.5.1 Inlining calls to lang_start

The main-function in the LLVM-IR is not the main-function the user wrote in Rust. Instead, it invokes `lang_start` [31, std::rt::lang_start], which in turn invokes the actual main-function the user wrote in Rust. As we do not need the functionality provided by `lang_start`, meaning capturing panic and unwinding, for our purposes, we simply replace the call to `lang_start` to call the user-written main-function directly. This cannot be done by the *FunctionInlinerPass* as there is no LLVM-IR implementation for `lang_start_internal`, so we are dealing with a call to an externally defined function.

4.5.2 Cleaning out GenMC-internal function bodies

We remove the function bodies of all of GenMC's internal functions and mark them as externally defined in the LLVM-IR. See Section 3.5.5 for more detail.

4.5.3 Forwarding calls to the memory allocator

We replace all calls to `rust__alloc`, `rust__dealloc`, `rust__realloc` and `rust__alloc_zeroed` to the versions of our custom memory allocator instead, namely `genmc_rust__alloc`, `genmc_rust__dealloc`, `genmc_rust__realloc` and `genmc_rust__alloc_zeroed`. See Section 3.4 for more detail.

4.5.4 Cleaning debug-output

All UIDs named "`*.dbg.spill`" as well as all instructions that depend on them, meaning users of them and users of those users and so on, are

generated purely for debugging purposes. We collect and remove all of those instructions completely. We do not disable debugging since we still need the other debug-output for later presentation to the user. It could be disabled by performing a release-build instead or using the flag "`debuginfo=0`".

4.5.5 Removing alloc-shim

In newer Rust versions, if the compiler detects that a different STD is being used (as is in our case), it will deliberately add an external global variable `__rust_no_alloc_shim_is_unstable` to force the user into manually linking and recognizing they may not have a memory allocator [28, Pull request #86844]. We do have a memory allocator and still use Rust's STD behind our wrapper, therefore we can safely remove this global variable fully.

Chapter 5

Verification

In this section we describe the changes we performed in GenMC's interpreter to support functionality we required in other chapters.

5.1 Supporting Mutex Try-Lock

We found that `__VERIFIER_mutex_trylock` caused a segmentation fault in GenMC when it was called on an already locked mutex, even if the lock is held by the same thread.

There were also no test-cases already present that covered try-lock in C. We wrote some test-cases in C which covered the different aspects:

1. Try-Lock on a free mutex
2. Try-Lock on a mutex held by another thread
3. Try-Lock on a mutex held by the same thread
4. Locking/Unlocking multiple times, mixing lock and try-lock

The C-tests also aborted with a segmentation fault when calling try-lock on an already held lock. So we concluded that the issue must be in GenMC's underlying implementation.

Expected Behavior

We must first understand what the expected behavior for `pthread_mutex_trylock` is in POSIX systems as the behavior for `__VERIFIER_mutex_trylock` is modelled after it.

The POSIX standard [9] states that "The pthread_mutex_trylock() function shall be equivalent to pthread_mutex_lock(), except that if the mutex object referenced by mutex is currently locked (by any thread, including the current

thread), the call shall return immediately." and "The pthread_mutex_trylock() function shall fail if: [EBUSY] The mutex could not be acquired because if was already locked.", where [EBUSY] is the corresponding error code.

Fixing it in GenMC

Calls to GenMC's internal functions like `__VERIFIER_mutex_trylock` are handled by GenMC's interpreter, which in turn calls GenMC's driver (see [14]). In this case `Interpreter::callMutexTryLock` calls `GenMCDriver::handleLoad`, which returns a `std::optional`. We found that the segmentation fault happens when `GenMCDriver::handleLoad` returned a `std::nullopt` (empty optional value) and provided an initial patch on which we continued work for this thesis. The issue was further taken up by the advisor and resolved outside this thesis.

To ensure that the new implementation as well as our own patch we initially used provide consistent results, we ran our C-tests 200 times with the option `--schedule-policy=arbitrary` to check if we always get the same number of executions, as was suggested by the advisor.

5.2 Supporting invoke on internal functions

Using certain internal functions (e.g. `__VERIFIER_assume`) with an `invoke` instruction [24, 'invoke' Instruction] instead of a `call` instruction [24, 'call' Instruction] caused a segmentation fault. It is important to support this case as Rust generally uses `invoke` over `call`.

Since `invoke` instructions are terminators, after executing such an instruction we need to jump to a new basic block (e.g. the normal destination). GenMC's interpreter supports this, but does it while handling the returning of a value from the function. Therefore, for internal functions which do not have a return value the interpreter did not have a next instruction to execute as no jump to a new basic block was performed. We solve this by calling the interpreter's `returnValueToCaller`-handler also for functions returning void. It can already handle a void return-type.

Chapter 6

Evaluation

In this chapter, we will reflect on our work and test our implementations in various ways. For every section, we describe our methodology and present our results, whereas the full results are deferred to the appendix.

Many tests from GenMC's pre-existing large test-harness were translated from C into Rust. These are very helpful because they allow us a direct comparison of number of executions as well as event traces between our Rust-version and a C-version with known expected results.

In a final case study, we analyze and try to verify several concurrent data-structures of a popular, open-source Rust-crate, namely Crossbeam [5].

Hazard Pointers: In the implementations for Treiber's stack and MS-queue we'll present later we must ensure that memory is not freed while another thread still uses a reference to it, this is to prevent *use-after-free* errors.

GenMC provides built-in tools for handling hazard pointers. We can mark a pointer being used by the current thread as hazardous, other threads may hold the same pointer. When such an hazardous pointer is freed by any thread, GenMC will ensure that this memory is only freed once all threads mark that pointer as not being in use anymore.

For setting up hazard pointers we must use `__VERIFIER_hp_alloc` to first allocate some memory. There is an issue unrelated to our work in GenMC where this memory is considered never freed even though a corresponding `__VERIFIER_hp_free` call is present and visible in the event-trace. This happens in C as well using the same tests and that warning was ignored for our purpose after notifying the advisor.

6.1 Litmus Tests

The litmus tests are a collection of 111 simple and small tests, usually compromising of only a few atomic RMW operations and fences over multiple threads. Most tests have multiple variants, which just spawn the threads in different orders.

The purpose of these is to ensure that the right amount of executions are explored under different memory models. Each variant should have the same number of executions.

We translated all litmus tests with all variants from C directly into Rust, using AI-powered code-completions (GitHub Copilot) to speed this very repetitive task up. The resulting code was ensured to be exactly as expected.

Them being so small and numerous makes it easy to verify that the basics of our Rust frontend work and if some fail, reason about their number of executions and compare outputs between the C and Rust version to ensure they are the same.

We ran the litmus tests on Rust 1.69 using LLVM-15.0.7 and Rust 1.78 using LLVM-18.1.2 inside a Debian-Linux Docker-container. All tests are successful, see the full console output in the appendix in Section A.2 and Section A.3, respectively. The first test taking longer as genmc-std is being built there.

6.2 Linearization Tests (Relinche)

GenMC implements the Relinche-Algorithm [8] for verifying a relaxed notion of bounded linearizability of a data-structure suited for weak memory models. Classical linearizability [11] describes the notion that every method on a data-structure behaves like a single atomic operation. Bounded linearizability in this context means the data-structure is correct for all clients that perform at most an upper-bounded number of operations on the data-structure, where that bound is a parameter. Correct is defined under a relaxed notion of classical linearizability to adequately capture the behaviors of weak memory models.

We use the pre-existing MPC *"Most parallel Client"* and translate the C implementations in GenMC's test harness for the MS-queue and Treiber's stack into Rust. Using the new `--link-with` option (See Section 3.5.3), we can link our Rust implementation of the queue/stack into the C-implementation for MPC. An example of the command being executed might look like this:

```
genmc --disable-estimation --rc11 -disable-mm-detector
    --check-lin-spec=queue_spec_v.in
    --link-with=/rust/tests/correct/data-structures
```

```
/ms-queue-dynamic/my_queue.rs -- -DRTN=2
-DWTN=2 -DMSQUEUE
-include /root/genmc/rust/tests/correct/relinche
/queue/empty_queue.h mpc.c
```

Full output of the relinche-tests using Rust 1.69 with LLVM-15.0.7 is added in the appendix at Section A.4.

We now describe the Treiber's Stack and Michael-Scott Queue these linearization tests were run on and their results.

6.2.1 Treiber's Stack

The Treiber's stack [33] is a classic in the introduction to concurrent, lock-free data-structures and formal verification thereof due to its simplicity.

The stack is represented using nodes and an atomic pointer to the `top/head` of the stack. Each `node` holds a `value` and a standard pointer to the `next` node in the stack.

To push and pop, it repeatedly tries modifying the `head` using a CAS-operation to replace it with a newly created `node` (push) or the current `head`'s `next` field (pop).

We must ensure that while popping, we do not free the memory of a `node` while another thread also still holds that same pointer. To achieve this, we use GenMC's hazard pointer tooling as described in the introduction to this chapter.

We get the same results as with the C-equivalent tests. GenMC recognizes that the Treiber's stack is not linearizable and prints the same events as being in conflict with each other as in C. Only difference we observed in the execution traces are additional events for `SPIN_START` and `LOOP_BEGIN`. These events are generated in the *SpinAssumePass* in the transformation phase when it cannot transform a spinloop into a `__VERIFIER_assume` call, like it is doing successfully in the C-case for this test, and instead falls back to a dynamic check. This does not further affect our results, but the *SpinAssumePass* can be extended to handle LLVM-IR generated by Rust better in future work.

6.2.2 Michael-Scott Queue

The MS-queue [15] is a lock-free, concurrent FIFO queue that is still widely in use. It is represented using a single linked list. The MS-queue is a helping data-structure, meaning if an operation fails because another thread is performing an operation on it also, the current thread will help the other thread to successfully finish its operation first.

Each `node` in the `queue` contains a `value` and an atomic pointer to the `next` node in the `queue`. The `queue` itself holds two atomic pointers: One to the `head` of the `queue` and the other to the `tail`. Initially, both point to a dummy node (sometimes called the sentinel node) to simplify the enqueue and dequeue logic when the queue is empty.

To enqueue a `value` (inserting at `tail`), allocate a `new` node holding the `value`. Atomically read the `tail` pointer and then the `next` field of the `tail`. We make sure the `tail` pointer has not changed at this point and retry if it has. If `next` is `null`, a CAS operation is used to link the `new` node as `next`. Otherwise, another thread is also enqueuing and we help move `tail` to point to `next`. We retry until we successfully linked our own `new` node to `next` of the `tail`. At the end, we move the `tail` to point to our `new` node.

To dequeue (remove from `head`), we atomically read `head`, `next` of that node and `tail` pointers. We make sure the `head` pointer has not changed at this point and retry if it has. If `head` and `tail` point to the same node, either `next` is `null`, meaning the queue is empty, or another thread is currently enqueuing and we help it move the `tail` to point to the `next`. If `head` and `tail` are not the same, we can simply read the value from the `next` node and use a CAS to move `head` to point to `next`.

We must ensure that while dequeuing, we do not free the memory of a node while another thread also still holds that same pointer. To achieve this, we use GenMC's hazard pointer tooling as described in the introduction to this chapter.

We get the same results as with the C-equivalent tests. GenMC considers the MS-queue to be linearizable.

6.3 Verifying Crossbeam Data-structures

We pulled the implementations of several concurrent data-structures from Crossbeam [5], a crate providing state-of-the-art tools for concurrent programming in Rust. The data-structures include three MPMC-queues (ArrayQueue, SegQueue, MS-Queue) and the classic Treiber's Stack.

For each data-structure, we will describe their working and the small adjustments we performed to Crossbeam's implementation for our purpose of model-checking before presenting the results of our tests.

A major difficulty we faced here was an issue when spawning threads using closures as described in Section 3.1.1. Consequently, where possible we used `thread::spawn` (with closures) and `thread::spawn_f` if that failed, for some tests we kept both versions in the repository for demonstration.

All tests in this section were executed on Rust 1.69 using LLVM-15.0.7.

6.3.1 MPMC-Queues

Following is the test set we used for ArrayQueue, SegQueue and MS-Queue.

Test 0: [18] Two threads, each saving their value to a global array, pushing that value onto the queue, popping from the queue and saving that popped value into a different global array. At the end, the sum of pushed values to the queue must match the sum of popped values from the queue. Both threads read and write to different indexes in the arrays.

Test 1: [17] Three threads, 1 producer (single push), 1 consumer, 1 producer (single push) + consumer.

Test 2: [21] Four threads, 2 producers each pushing two values and 2 consumers each popping 2 values.

Test 3: [21] Three threads, each having 1 push and 1 pop operation.

Test 4: [21] Single thread, pushing 100 elements then popping 100 elements to verify FIFO ordering.

Test 5: [3] Main thread spawning a thread that pushes a value, then the main thread pushes a value too. Main thread checks the length of the queue and that it is not empty. Only performed for ArrayQueue and SegQueue.

ArrayQueue

This is a bounded MPMC-Queue based on Dmitry Vyukov's algorithm [34], meaning the queue holds a buffer of a predefined maximal size N and cannot grow beyond it.

The `queue` is represented by a `buffer` (an array) with N `slots`, a `head` and a `tail` each represented by an atomic counter to index into the `buffer`. Each `slot` holds a `value` and a `stamp` that is an atomic counter. If `stamp` is the same as `tail`, then this is the `slot` to write into next. If it is one larger than `head`, than this is the `slot` to read from next. The `head`, `tail` and `stamp` each hold two smaller values in one larger atomic counter, a `lap` and a `index`.

To enqueue, we read the current `index` and `lap` from the `tail`. To compute the `new tail` after enqueuing, we increment the `index` and if the new index is larger than N, we set the `new tail` to the next `lap` with a `index` of zero instead. We check the `stamp` of the `slot` at the new `index` we computed. If `stamp` equals the original `tail`, we can try moving the `tail` forward to the `new tail` and if that succeeds, we can write the `value` into the `slot` (from old `tail`) and update the `slot`'s `stamp` to `tail + 1`, otherwise retry. If `tail` and `stamp` are not the same, the `head` may be one lap behind the `tail`, in which case the `queue` is full, or the `stamp` was not yet updated, in which case we try again.

To dequeue, we read the current `index` and `lap` from the `head` and check the `stamp` of the `slot` at `index`. If the `stamp` is one larger then the `head`, we can try to dequeue. We first compute the `new head` as for enqueue, increasing the `index` or the `lap` if the `index` would go beyond N. We try to move the `head` to the `new head` and if that succeeds, we can read the `value` from the `slot` (old head) and update the `stamp` for the next operation, otherwise retry. If the `head` and `stamp` are the same, the `head` and `tail` might also be the same, in which case the `queue` is empty. Otherwise, we retry.

For our purpose, we removed cache padding on the values and exponential backoff. Loops are bounded as otherwise infinite loops are possible. We can set a value for the loop-bound between one and five using a Rust cfg-parameter.

All five tests ran successfully, we deferred GenMC's output of these tests to the appendix at Section A.5.1.

SegQueue

Crossbeam does not cite any other works they used for the SegQueue. We assume it was developed by the Crossbeam project developers directly.

SegQueue is organized in a special way compared to the others. Each `value` is held in a `slot`, a `slot` holds the `value` and a `state`. We can set `WRITE`, `READ` and `DESTROY` bits for the `state`. A `block` holds an array of `slots` and an atomic pointer to the `next` block in the linked-list which represents the `queue`. The `queue` is represented by two `positions`: `head` and `tail`. A `position` holds an atomic pointer to a `block` and an atomic counter representing an `index` into the block.

To enqueue, we first atomically load the `tail`'s `position`. We try to insert into the next `slot` position in the block. If we are going to write into the last `slot` of the block, we will also need to allocate the `next block`. If the `tail`'s `position` points to one after the last `index` of a `block`, it means the last `slot` is already written to by another thread but the `next block` not yet linked, so the thread must retry. If we're doing the first enqueue, we must allocate the first block or retry if that fails, which can happen when another thread allocated it faster than us. At this point, our thread is ready to move the `tail` forwards. If we succeed and the new `tail`'s `position` is the last `slot` of the `block`, we allocate a `next block` and write our `value` into the `slot`, setting the `WRITE` bit in the `state`. If advancing the `tail` fails, we retry.

To dequeue, we first atomically load the `head`'s `position`. If the `head` points to a `slot` beyond the `block`, we must wait until a `next block` is installed by another thread that has moved the `head` forward and retry. We use a `HAS_NEXT` bit in the `position` to determine if the `block` has a `next block` and update this bit if needed. If `head` and `tail` point to the same `slot` in the

same `block`, then the `queue` is empty. Assuming the `queue` is not empty, we check if the `block` we will read from is not yet allocated, which can be the case if the first enqueue is still in progress, and retry in that case. We move the `head` to the next `slot`, retrying if that fails. If the new `head`'s `slot` lies beyond the bound of the `block`, we wait until the `next block` is allocated and to read the `value` from its first `slot`. To read the value, we must wait until the enqueuing thread set the `WRITE` bit for the `slot`. We set the `READ` bit after reading from the `slot`. If we are reading from the last `slot` in the `block`, we try to destroy the block by setting every `slot` that was not already `READ` to `DESTROY` and deallocate the `block` from memory only if all `slot`'s were already set to `READ`. If we're not reading from the last `slot` but `DESTROY` was set on our `slot` before we marked it as `READ`, then another thread read from the last `slot` and could not destroy the `block` because our thread was still reading from it, so we try destroying the `block` now.

For our purpose we removed cache padding on the values and exponential backoff. Loops are bounded as otherwise infinite loops are possible. We can set a value for the loop-bound between one and five using a Rust cfg-parameter.

All five tests ran successfully, we deferred GenMC's output of these tests to the appendix at Section A.5.2.

MS-Queue

Crossbeam's implementation of the Michael-Scott Queue [15] was removed from the collection of data-structures which are exposed to users in favor of SegQueue and ArrayQueue. We found it still being used deep behind the scenes of Crossbeam's global garbage collector, that is the implementation we verify here.

We already described how the MS-queue operates when presenting the Relinche tests in Section 6.2.

Crossbeam's MS-queue pins the current thread to a CPU-core at the start of a enqueue/dequeue operation and unpins it when the operation is finished. When performing a dequeue on a MS-queue, we must ensure that we do not deallocate a node from memory if another thread still holds a reference to it. Crossbeam ensures that using a time-deferred deallocation, where Crossbeam's custom garbage collector waits and only deallocates the node if all threads are currently not pinned to any core, i.e. the system is in a quiescent state in terms of operations on the MS-queue. To achieve the same goal we use GenMC's hazard-pointer tools instead as was done in our linearization tests.

For our purpose, we also removed cache padding on the values.

We could not successfully run test 3 due to GenMC's interpreter not yet supporting constant-expression aggregate types [24, Complex Constants]. Test 5 was not performed because it is not relevant as this implementation does not offer a method to check either the length of the queue or if it is empty. The other four tests ran successfully, we deferred GenMC's output of these tests to the appendix at Section A.5.3.

6.3.2 Treiber's Stack

Like the MS-Queue, the Treiber's stack was removed from Crossbeam's repo in favor of other alternatives. We found Crossbeam's implementation in the commit where they deleted it [5, Pull Request #301] and verify that version for our case study.

We already described how the Treiber's stack operates when presenting the Relinche tests in Section 6.2.

Crossbeam's Treiber's stack also uses thread-pinning and time-deferred deallocation as described for Crossbeam's MS-queue such that a node does not get deallocated from memory while another thread still holds a reference to it. We again use GenMC's hazard-pointer tools instead to achieve the same goals.

For our purpose we also removed cache padding on the values and use Rust's AtomicPtr [31, std::sync::atomic::AtomicPtr] directly instead of Crossbeam's custom "Atomic"-wrapper, making our model self-contained in one file.

Test 0: (Copied from GenMC's test harness) Two writers and one reader. The writers atomically store a value in an array before pushing a value onto the stack. The reader pops a value from the stack and atomically reads from the array. Each thread accesses a different index in the array.

Test 1: [1] We push four values (0 to 3) onto the stack, then spawn four threads. Each thread pops one value from the stack and asserts it is in range.

Both tests ran successfully, we deferred GenMC's output of these tests to the appendix at Section A.5.4.

Chapter 7

Conclusion

This thesis set out to extend the GenMC stateless model checker to verify Rust programs. Through various additions in all three stages of GenMC (compilation, transformation, verification) and the development of custom implementations for relevant concurrency and memory management APIs in Rust's standard library tailored for our purposes, we demonstrated that GenMC can be extended and used to formally verify the correctness of Rust programs under weak memory models. While the most important concurrency and memory management primitives are now fully supported and we can thus already verify a comprehensive range of models, there are still issues with dynamic dispatching of functions which affect the use of closures for spawning threads. Ultimately, this thesis lays significant groundwork to the broader goal of enabling automated verification for Rust programs, whose use in systems programming is ever so increasing in popularity.

7.1 Related Work

RustMC [20]: Coincidentally, around the same time our work commenced, RustMC was released. It tries to achieve the same goals as we had, namely verifying Rust programs using GenMC. Although they faced some of the same problems as we did, their methodology is completely different. RustMC compiles as normal using Rust's STD, then tries to identify and replace libc system calls in the LLVM-IR (e.g. `pthread_create`) with GenMC's corresponding equivalent (e.g. `__VERIFIER_thread_create`). We built a custom wrapper around Rust's STD instead which uses GenMC's tooling directly in the places where it is needed and link the client programs to our custom wrapper. This has several key advantages, namely easier maintenance over a long time since we control the implementations of Rust-interfaces like spawn [31, std::thread::spawn] and our ability to provide actual custom implementations of primitives independently of how these are implemented

in Rust's STD. We made good use of that when supporting mutexes, where our interface uses `__VERIFIER_mutex_lock` directly. As seen in our discussion in Section 3.3, Rust's mutex-implementation actually relies on futexes and spin-loops for better performance on a contended lock, making that implementation not well suited for our use case.

Loom [4]: Loom is currently the industry standard for model checking concurrent programs in Rust and is also heavily used by Crossbeam to test their implementations. Loom builds on CDSChecker [19], it therefore inherits the same drawbacks of CDSChecker in comparison to GenMC as described in [13]. Namely only supporting the C11 memory model as formalized in [2], not correctly handling sequentially consistent atomic operations and exploring additional executions that may not be possible. Our work provides a valuable extension to the current Rust landscape by enabling verification of Rust programs under a wide variety of weaker memory models while also handling the aforementioned edge-cases right.

Miri [29]: Finds undefined behavior [30, 17.2 Behavior considered undefined] in Rust by checking many possible executions, but unlike GenMC, it explores far from all executions. Miri operates as an interpreter on Rust's mid-level IR (MIR). It is maintained by Rust's Programming Language Team directly and used heavily throughout Rust's STD to detect undefined behavior and document all such bugs found in earlier Rust versions as Miri-tests.

GenMC [14]: Of course, all of our work is on top of GenMC, a generic model checker for weak memory models. GenMC verifies programs on the level of LLVM-IR, with the foresight that using a popular intermediate representation will allow it to be extended to all kinds of programming languages which use that in their compilation pipeline. We now made use of that foresight to extend GenMC to a new programming language that is conceptually very different from the C-languages GenMC was originally built on.

7.2 Future Work

We present some extensions to our work which can be seen as possible next steps in future projects.

Fixing dynamic dispatch: One major issue we still face are the segmentation faults happening with spawning threads using closures as described in detail in Section 3.1.1. The LLVM-IR that is produced was extensively analyzed and ensured to be correct. This must be fixed in GenMC's interpreter and driver.

Scoped threads with manual join: Right now we wait until all scoped threads are finished before returning from the scope like it is also done in Rust's STD. This can be significantly improved upon by collecting and actually joining all

threads in the scope one-by-one instead. See Crossbeam's [5] implementation of scoped threads as a reference of how this can be achieved.

Spawn symmetric: Although we do provide GenMC's verifier interface for spawning symmetric threads, it is not yet used in our threads-module in genmc-std and the user would need to handle it completely manually.

Supporting barriers: We focused on mutexes first in our work as the most important primitive, but more primitives like barriers can be added in the future.

Better event-traces and error messages: GenMC uses the debug-output generated by Clang to indicate in which file and line an event happened. This needs to be extended to better support the debug-output generated by Rust as these indicators are currently often missing. When panicking, GenMC as of now gives an error displaying the mangled function name: "Tried to execute unknown external function _ZN4core9panicking5panic17h64521674a956cb9aE".

Support Constant-Expression Aggregates: GenMC's interpreter currently does not support constant expression aggregates [24, Complex Constants]. Although rare, we have seen these being used more often with newer versions of Rust.

SpinAssumePass: This pass tries to replace spinloops with `__VERIFIER_assume` calls, making verification more efficient. It can probably be extended so that these optimizations can be applied at more places in Rust. An example where it is applied successfully for C but not for Rust is presented in Section 6.2.1.

Optimized implementation for Arc: Arc [31, std::sync::Arc] (atomic reference counters) enables us to share data between threads but induces more synchronization due to the use of two separate atomic variables, causing more executions to be explored. This may be optimized as Arc does not interfere with our client-applications data.

Thread-local storage: Supporting the `thread_local!` macro [31] could open up a whole new class of applications. For example, Crossbeam's garbage collector relies on it and therefore its implementation could be verified once this macro is supported.

Appendix A

Appendix

A.1 Verifying Arithmetic Intrinsics With Overflow

We create tests for all following combinations:

$$\{Signed, Unsigned\} \times \{Add, Sub, Mul\} \times \{Overflowing, Non\text{-}Overflowing\}$$
$$\times \{\texttt{i8}, \texttt{i16}, \texttt{i32}, \texttt{i64}\}$$

for a total of 48 tests. In each test, we choose values such that the binary operation barely overflows on the given data-type or just barely does not overflow. In what follows, we'll show some representative examples.

Signed, Overflowing Addition on `i8`:

```
define i1 @main() {
start:
  %0 = call { i8, i1 } @llvm.sadd.with.overflow.i8(i8 127, i8 1)
  %_5.0 = extractvalue { i8, i1 } %0, 0
  %_5.1 = extractvalue { i8, i1 } %0, 1
  br i1 %_5.1, label %overflowing, label %panic
overflowing:
  %result = icmp eq i8 %_5.0, -128
  br i1 %result, label %return, label %panic
return:
  ret i1 %result
panic:
  unreachable
}
```

Unsigned, Non-Overflowing Addition on `i8`:

```
define i1 @main() {
start:
  %0 = call { i8, i1 } @llvm.uadd.with.overflow.i8(i8 250, i8 5)
  %_5.0 = extractvalue { i8, i1 } %0, 0
  %_5.1 = extractvalue { i8, i1 } %0, 1
  br i1 %_5.1, label %panic, label %nonoverflowing
nonoverflowing:
  %result = icmp eq i8 %_5.0, 255
  br i1 %result, label %return, label %panic
return:
  ret i1 %result
panic:
  unreachable
}
```

Signed, Overflowing Subtraction on `i64`:

```
define i1 @main() {
start:
  %0 = call { i64, i1 } @llvm.ssub.with.overflow.i64(
    i64 -9223372036854775808, i64 1)
  %_5.0 = extractvalue { i64, i1 } %0, 0
  %_5.1 = extractvalue { i64, i1 } %0, 1
  br i1 %_5.1, label %overflowing, label %panic
overflowing:
  %result = icmp eq i64 %_5.0, 9223372036854775807
  br i1 %result, label %return, label %panic
return:
  ret i1 %result
panic:
  unreachable
}
```

Unsigned, Non-Overflowing Multiplication on `i32`:

```
define i1 @main() {
start:
  %0 = call { i32, i1 } @llvm.umul.with.overflow.i32(
    i32 2147483647, i32 2)
  %_5.0 = extractvalue { i32, i1 } %0, 0
  %_5.1 = extractvalue { i32, i1 } %0, 1
```

```
  br i1 %_5.1, label %panic, label %nonoverflowing
nonoverflowing:
  %result = icmp eq i32 %_5.0, 4294967294
  br i1 %result, label %return, label %panic
return:
  ret i1 %result
panic:
  unreachable
}
```

Our pattern is simple: We call the intrinsic and execute an `unreachable` instruction if the result of our intrinsic lowering is not what we expect. Note the maximum and minimum values for the *signed* / *unsigned* variants and how these are computed:

$$\text{MAX}(\texttt{i8}\ \textit{Signed}) = 2^{8-1} - 1 = 127$$

$$\text{MAX}(\texttt{i8}\ \textit{Unsigned}) = 2^{8} - 1 = 255$$

$$\text{MIN}(\texttt{i64}\ \textit{Signed}) = (-1) * 2^{64-1} = -9223372036854775808$$

$$\text{MAX}(\texttt{i64}\ \textit{Signed}) = 2^{64-1} - 1 = 9223372036854775807$$

$$\text{MAX}(\texttt{i32}\ \textit{Unsigned}) = 2^{32} - 1 = 4294967295$$

A.2 Litmus Tests LLVM-15 / Rust 1.69

```
-----------------------------------------------------------------------
--- Preparing to run testcases in LITMUS under RC11 with MO
-----------------------------------------------------------------------

-----------------------------------------------------------------------
| Testcase               | Result | Execs    | Blocked  | Avg.time |
-----------------------------------------------------------------------
| 2+2W/                  | SAFE   |        4 |          |     2.70 |
| 2+2W+2sc+scf/          | SAFE   |        3 |          |     1.21 |
| 2+2W+3sc+rel1/         | SAFE   |        4 |          |     1.25 |
| 2+2W+3sc+rel2/         | SAFE   |        4 |          |     1.36 |
| 2+2W+4sc/              | SAFE   |        3 |          |     1.28 |
| 2+2W+scfs/             | SAFE   |        3 |          |     1.25 |
| 2CoWR                  | SAFE   |        4 |          |     1.29 |
| CoRR                   | SAFE   |        6 |          |     1.38 |
| CoRR0                  | SAFE   |        4 |          |     1.27 |
| CoRR1/                 | SAFE   |       36 |          |     1.43 |
| CoRR2/                 | SAFE   |       72 |          |     1.29 |
```

CoRW/	SAFE	3		1.32
CoWR/	SAFE	3		1.30
IRIW-acq-sc	SAFE	16		1.24
IRIWish	SAFE	28		1.33
JLT/	SAFE	8		1.26
LB	SAFE	3		1.24
LB+acq	SAFE	3		1.28
LB+addr	SAFE	3		1.30
LB+addr-fun	SAFE	3		1.27
LB+addr-ind-fun	SAFE	3		1.32
LB+ctrl	SAFE	3		1.25
LB+ctrl+rel	SAFE	3		1.24
LB+dep	SAFE	3		1.29
LB+fr-fr-data+data/	SAFE	18		1.25
LB+incMPs	SAFE	15		1.29
LB+rel	SAFE	3		1.23
LB2	SAFE	3		1.26
LB3	SAFE	7		1.28
MCP-rrw/	SAFE	6		1.35
MP	SAFE	3		1.25
MP+incMP	SAFE	7		1.26
MP+rel+acq	SAFE	2		1.47
MP+rel+acqf	SAFE	2		1.37
MP+relf+acq	SAFE	2		1.29
MP+relf+acqf	SAFE	2		1.28
MP+rels+acq/	SAFE	4		1.36
MP+rels+acqf/	SAFE	4		1.28
MP2	SAFE	9		1.30
MPU+rel+acqf/	SAFE	6		1.28
MPU+relf+acq/	SAFE	6		1.31
MPU+relf+acqf/	SAFE	6		1.30
MPU+rels+acq/	SAFE	13		1.44
MPU+rels+acqf/	SAFE	13		1.29
MPU2+rel+acq	SAFE	14		1.44
MPU2+rel+acqf	SAFE	14		1.42
MPU2+relf+acq	SAFE	14		1.38
MPU2+relf+acqf	SAFE	14		1.43
MPU2+rels+acq/	SAFE	36		1.34
MPU2+rels+acqf/	SAFE	36		1.38
RMWFix	SAFE	4		1.32
RWC+syncs	SAFE	7		1.27
S/	SAFE	4		1.38
S+rel+acq	SAFE	2		1.26
S+rel+acqf	SAFE	2		1.32

S+relf+acq	SAFE	2		1.31
S+relf+acqf	SAFE	2		1.25
S+rels+acq/	SAFE	4		1.45
S+rels+acqf/	SAFE	4		1.30
SB	SAFE	4		1.37
SB+2sc+scf	SAFE	3		1.50
SB+3sc+acq	SAFE	4		1.34
SB+3sc+rel	SAFE	4		1.26
SB+4sc	SAFE	3		1.47
SB+assert	SAFE	3		1.51
SB+rfis	SAFE	4		1.35
SB+scfs	SAFE	3		1.24
TC1	SAFE	3		1.29
TC2	SAFE	4		1.52
W+JW	SAFE	1		1.41
W+RRR+W/	SAFE	20		1.32
W+RWC	SAFE	7		1.24
WRC+dep	SAFE	8		1.27
WWR+2WR/	SAFE	0	8	1.33
WWmerge1	SAFE	17		1.27
WWmerge2	SAFE	8		1.27
Z6+acq/	SAFE	7		1.31
Z6.U/	SAFE	22		1.26
assume-ctrl	SAFE	4		1.31
atomicpo/	SAFE	4		1.31
casdep	SAFE	2		1.30
ccr	SAFE	2		1.31
cii	SAFE	6		1.36
cumul-release	SAFE	8		1.34
default/	SAFE	12		1.31
detour/	SAFE	9		1.29
fr+w+w+w+reads/	SAFE	210		1.29
inc+inc+RR+W+RR/	SAFE	600		1.44
inc2w/	SAFE	6		1.30
isa2	SAFE	7		1.26
malloc-dep	SAFE	2		1.62
peterZ	SAFE	7		1.27
po-loc	SAFE	3		1.30
psc-ar	SAFE	1	1	1.33
psc-sbhbsb	SAFE	1	1	1.39
rel-B-cumul-acq	SAFE	16		1.84
rel-pporf/	SAFE	3		1.71
relacq-B-cumul	SAFE	8		1.29
relseq/	SAFE	20		1.41

```
| rfi-preserved          | SAFE   |        3 |          |     1.41 |
| rii                    | SAFE   |        3 |          |     1.39 |
| rinc+wr/               | SAFE   |        3 |          |     1.33 |
| riwi                   | SAFE   |        3 |          |     1.32 |
| small0/                | SAFE   |        9 |          |     1.35 |
| small1                 | SAFE   |        8 |          |     1.26 |
| viktor-relseq          | SAFE   |      180 |          |     1.28 |
| wcii/                  | SAFE   |       24 |          |     1.49 |
| wrw/                   | SAFE   |       22 |          |     1.42 |
| wrw/                   | SAFE   |      300 |          |     1.51 |
| wrw/                   | SAFE   |     6576 |          |     2.12 |
-------------------------------------------------------------------
--- Test time:  211.71
-------------------------------------------------------------------


-------------------------------------------------------------------
--- Preparing to run testcases in LITMUS under IMM with MO
-------------------------------------------------------------------

-------------------------------------------------------------------
| Testcase               | Result | Execs    | Blocked  | Avg.time |
-------------------------------------------------------------------
| 2+2W/                  | SAFE   |        4 |          |     2.73 |
| 2+2W+2sc+scf/          | SAFE   |        3 |          |     1.38 |
| 2+2W+3sc+rel1/         | SAFE   |        4 |          |     1.80 |
| 2+2W+3sc+rel2/         | SAFE   |        4 |          |     1.29 |
| 2+2W+4sc/              | SAFE   |        3 |          |     1.30 |
| 2+2W+scfs/             | SAFE   |        3 |          |     1.26 |
| 2CoWR                  | SAFE   |        4 |          |     1.38 |
| CoRR                   | SAFE   |        6 |          |     1.36 |
| CoRR0                  | SAFE   |        4 |          |     1.36 |
| CoRR1/                 | SAFE   |       36 |          |     1.32 |
| CoRR2/                 | SAFE   |       72 |          |     1.31 |
| CoRW/                  | SAFE   |        3 |          |     1.25 |
| CoWR/                  | SAFE   |        3 |          |     1.28 |
| IRIW-acq-sc            | SAFE   |       16 |          |     1.40 |
| IRIWish                | SAFE   |       28 |          |     1.78 |
| JLT/                   | SAFE   |        8 |          |     1.31 |
| LB                     | SAFE   |        3 |          |     1.25 |
| LB+acq                 | SAFE   |        3 |          |     1.27 |
| LB+addr                | SAFE   |        3 |          |     1.43 |
| LB+addr-fun            | SAFE   |        3 |          |     1.43 |
| LB+addr-ind-fun        | SAFE   |        3 |          |     1.30 |
```

LB+ctrl	SAFE	3		1.24
LB+ctrl+rel	SAFE	3		1.46
LB+dep	SAFE	3		1.27
LB+fr-fr-data+data/	SAFE	19		1.49
LB+incMPs	SAFE	15		1.38
LB+rel	SAFE	3		1.45
LB2	SAFE	4		1.29
LB3	SAFE	8		1.23
MCP-rrw/	SAFE	6		1.25
MP	SAFE	3		1.30
MP+incMP	SAFE	7		1.23
MP+rel+acq	SAFE	2		1.30
MP+rel+acqf	SAFE	2		1.26
MP+relf+acq	SAFE	2		1.29
MP+relf+acqf	SAFE	2		1.33
MP+rels+acq	SAFE	4		1.28
MP+rels+acqf	SAFE	4		1.44
MP2	SAFE	9		1.25
MPU+rel+acqf/	SAFE	6		1.49
MPU+relf+acq/	SAFE	6		1.41
MPU+relf+acqf/	SAFE	6		1.27
MPU+rels+acq/	SAFE	13		1.36
MPU2+rel+acq	SAFE	14		1.29
MPU2+rel+acqf	SAFE	14		1.40
MPU2+relf+acq	SAFE	14		1.43
MPU2+relf+acqf	SAFE	14		1.39
MPU2+rels+acq/	SAFE	36		1.33
RMWFix	SAFE	4		1.29
RWC+syncs	SAFE	7		1.23
S/	SAFE	4		1.25
S+rel+acq	SAFE	2		1.33
S+rel+acqf	SAFE	2		1.39
S+relf+acq	SAFE	2		1.27
S+relf+acqf	SAFE	2		1.26
S+rels+acq/	SAFE	4		1.43
SB	SAFE	4		1.42
SB+2sc+scf	SAFE	3		1.28
SB+3sc+acq	SAFE	4		1.26
SB+3sc+rel	SAFE	4		1.40
SB+4sc	SAFE	3		1.26
SB+assert	SAFE	3		1.32
SB+rfis	SAFE	4		1.25
SB+scfs	SAFE	3		1.24
TC1	SAFE	3		1.43

TC2	SAFE	4		1.28
W+JW	SAFE	1		1.35
W+RRR+W/	SAFE	20		1.32
W+RWC	SAFE	7		1.24
WRC+dep	SAFE	8		1.56
WWR+2WR/	SAFE	0	8	1.32
WWmerge1	SAFE	17		1.25
WWmerge2	SAFE	8		1.31
Z6+acq/	SAFE	7		1.30
Z6.U/	SAFE	22		1.52
assume-ctrl	SAFE	4		1.27
atomicpo/	SAFE	4		1.29
casdep	SAFE	2		1.30
ccr	SAFE	2		1.25
cii	SAFE	6		1.35
cumul-release	SAFE	8		1.44
default/	SAFE	12		1.27
detour/	SAFE	9		1.28
fr+w+w+w+reads/	SAFE	210		1.29
inc+inc+RR+W+RR/	SAFE	600		1.54
inc2w/	SAFE	6		1.70
isa2	SAFE	7		1.47
malloc-dep	SAFE	2		1.68
peterZ	SAFE	7		1.31
po-loc	SAFE	3		1.24
psc-ar	SAFE	0	2	1.33
psc-sbhbsb	SAFE	1	1	1.50
rel-B-cumul-acq	SAFE	16		1.25
rel-pporf/	SAFE	4		1.36
relacq-B-cumul	SAFE	8		1.29
relseq/	SAFE	20		1.27
rfi-preserved	SAFE	4		1.28
rii	SAFE	3		1.26
rinc+wr/	SAFE	3		1.38
riwi	SAFE	4		1.33
small0/	SAFE	9		1.35
small1	SAFE	8		1.25
viktor-relseq	SAFE	180		1.26
wcii/	SAFE	24		1.39
wrw/	SAFE	22		1.39
wrw/	SAFE	300		1.59
wrw/	SAFE	6576		2.31

--- Test time: 208.73

```
--------------------------------------------------------------------


--------------------------------------------------------------------
---  Testing proceeded as expected. Total time: 420.44
--------------------------------------------------------------------
```

A.3 Litmus Tests LLVM-18 / Rust 1.78

```
--------------------------------------------------------------------
--- Preparing to run testcases in LITMUS under RC11 with MO
--------------------------------------------------------------------

--------------------------------------------------------------------
| Testcase               | Result | Execs    | Blocked  | Avg.time |
--------------------------------------------------------------------
| 2+2W/                  | SAFE   |        4 |          |     3.07 |
| 2+2W+2sc+scf/          | SAFE   |        3 |          |     1.33 |
| 2+2W+3sc+rel1/         | SAFE   |        4 |          |     1.24 |
| 2+2W+3sc+rel2/         | SAFE   |        4 |          |     1.22 |
| 2+2W+4sc/              | SAFE   |        3 |          |     1.29 |
| 2+2W+scfs/             | SAFE   |        3 |          |     1.41 |
| 2CoWR                  | SAFE   |        4 |          |     1.23 |
| CoRR                   | SAFE   |        6 |          |     1.32 |
| CoRR0                  | SAFE   |        4 |          |     1.22 |
| CoRR1/                 | SAFE   |       36 |          |     1.37 |
| CoRR2/                 | SAFE   |       72 |          |     1.24 |
| CoRW/                  | SAFE   |        3 |          |     1.21 |
| CoWR/                  | SAFE   |        3 |          |     1.23 |
| IRIW-acq-sc            | SAFE   |       16 |          |     1.22 |
| IRIWish                | SAFE   |       28 |          |     1.35 |
| JLT/                   | SAFE   |        8 |          |     1.24 |
| LB                     | SAFE   |        3 |          |     1.29 |
| LB+acq                 | SAFE   |        3 |          |     1.24 |
| LB+addr                | SAFE   |        3 |          |     1.29 |
| LB+addr-fun            | SAFE   |        3 |          |     1.27 |
| LB+addr-ind-fun        | SAFE   |        3 |          |     1.28 |
| LB+ctrl                | SAFE   |        3 |          |     1.26 |
| LB+ctrl+rel            | SAFE   |        3 |          |     1.26 |
| LB+dep                 | SAFE   |        3 |          |     1.27 |
| LB+fr-fr-data+data/    | SAFE   |       18 |          |     1.27 |
| LB+incMPs              | SAFE   |       15 |          |     1.22 |
| LB+rel                 | SAFE   |        3 |          |     1.25 |
```

LB2	SAFE	3		1.23
LB3	SAFE	7		1.22
MCP-rrw/	SAFE	6		1.23
MP	SAFE	3		1.24
MP+incMP	SAFE	7		1.22
MP+rel+acq	SAFE	2		1.31
MP+rel+acqf	SAFE	2		1.33
MP+relf+acq	SAFE	2		1.46
MP+relf+acqf	SAFE	2		1.33
MP+rels+acq/	SAFE	4		1.35
MP+rels+acqf/	SAFE	4		1.26
MP2	SAFE	9		1.24
MPU+rel+acqf/	SAFE	6		1.27
MPU+relf+acq/	SAFE	6		1.25
MPU+relf+acqf/	SAFE	6		1.25
MPU+rels+acq/	SAFE	13		1.28
MPU+rels+acqf/	SAFE	13		1.24
MPU2+rel+acq	SAFE	14		1.29
MPU2+rel+acqf	SAFE	14		1.43
MPU2+relf+acq	SAFE	14		1.39
MPU2+relf+acqf	SAFE	14		1.41
MPU2+rels+acq/	SAFE	36		1.29
MPU2+rels+acqf/	SAFE	36		1.52
RMWFix	SAFE	4		1.28
RWC+syncs	SAFE	7		1.23
S/	SAFE	4		1.23
S+rel+acq	SAFE	2		1.25
S+rel+acqf	SAFE	2		1.26
S+relf+acq	SAFE	2		1.25
S+relf+acqf	SAFE	2		1.24
S+rels+acq/	SAFE	4		1.26
S+rels+acqf/	SAFE	4		1.24
SB	SAFE	4		1.35
SB+2sc+scf	SAFE	3		1.22
SB+3sc+acq	SAFE	4		1.25
SB+3sc+rel	SAFE	4		1.23
SB+4sc	SAFE	3		1.23
SB+assert	SAFE	3		1.30
SB+rfis	SAFE	4		1.23
SB+scfs	SAFE	3		1.29
TC1	SAFE	3		1.43
TC2	SAFE	4		1.38
W+JW	SAFE	1		1.35
W+RRR+W/	SAFE	20		1.35

```
| W+RWC                  | SAFE   |        7 |          |     1.36 |
| WRC+dep                | SAFE   |        8 |          |     1.24 |
| WWR+2WR/               | SAFE   |        0 |        8 |     1.30 |
| WWmerge1               | SAFE   |       17 |          |     1.24 |
| WWmerge2               | SAFE   |        8 |          |     1.24 |
| Z6+acq/                | SAFE   |        7 |          |     1.24 |
| Z6.U/                  | SAFE   |       22 |          |     1.23 |
| assume-ctrl            | SAFE   |        4 |          |     1.40 |
| atomicpo/              | SAFE   |        4 |          |     1.27 |
| casdep                 | SAFE   |        2 |          |     1.24 |
| ccr                    | SAFE   |        2 |          |     1.26 |
| cii                    | SAFE   |        6 |          |     1.32 |
| cumul-release          | SAFE   |        8 |          |     1.32 |
| default/               | SAFE   |       12 |          |     1.22 |
| detour/                | SAFE   |        9 |          |     1.25 |
| fr+w+w+w+reads/        | SAFE   |      210 |          |     1.24 |
| inc+inc+RR+W+RR/       | SAFE   |      600 |          |     1.40 |
| inc2w/                 | SAFE   |        6 |          |     1.22 |
| isa2                   | SAFE   |        7 |          |     1.23 |
| malloc-dep             | SAFE   |        2 |          |     1.62 |
| peterZ                 | SAFE   |        7 |          |     1.23 |
| po-loc                 | SAFE   |        3 |          |     1.26 |
| psc-ar                 | SAFE   |        1 |        1 |     1.31 |
| psc-sbhbsb             | SAFE   |        1 |        1 |     1.28 |
| rel-B-cumul-acq        | SAFE   |       16 |          |     1.25 |
| rel-pporf/             | SAFE   |        3 |          |     1.30 |
| relacq-B-cumul         | SAFE   |        8 |          |     1.23 |
| relseq/                | SAFE   |       20 |          |     1.24 |
| rfi-preserved          | SAFE   |        3 |          |     1.23 |
| rii                    | SAFE   |        3 |          |     1.24 |
| rinc+wr/               | SAFE   |        3 |          |     1.35 |
| riwi                   | SAFE   |        3 |          |     1.46 |
| small0/                | SAFE   |        9 |          |     1.26 |
| small1                 | SAFE   |        8 |          |     1.29 |
| viktor-relseq          | SAFE   |      180 |          |     1.25 |
| wcii/                  | SAFE   |       24 |          |     1.25 |
| wrw/                   | SAFE   |       22 |          |     1.37 |
| wrw/                   | SAFE   |      300 |          |     1.47 |
| wrw/                   | SAFE   |     6576 |          |     1.97 |
-------------------------------------------------------------------------
--- Test time:  204.35
-------------------------------------------------------------------------
```

```
-------------------------------------------------------------------
--- Preparing to run testcases in LITMUS under IMM with MO
-------------------------------------------------------------------

-------------------------------------------------------------------
| Testcase                | Result | Execs    | Blocked  | Avg.time |
-------------------------------------------------------------------
| 2+2W/                   | SAFE   |        4 |          |     2.81 |
| 2+2W+2sc+scf/           | SAFE   |        3 |          |     1.23 |
| 2+2W+3sc+rel1/          | SAFE   |        4 |          |     1.45 |
| 2+2W+3sc+rel2/          | SAFE   |        4 |          |     1.24 |
| 2+2W+4sc/               | SAFE   |        3 |          |     1.29 |
| 2+2W+scfs/              | SAFE   |        3 |          |     1.25 |
| 2CoWR                   | SAFE   |        4 |          |     1.24 |
| CoRR                    | SAFE   |        6 |          |     1.25 |
| CoRR0                   | SAFE   |        4 |          |     1.26 |
| CoRR1/                  | SAFE   |       36 |          |     1.34 |
| CoRR2/                  | SAFE   |       72 |          |     1.31 |
| CoRW/                   | SAFE   |        3 |          |     1.31 |
| CoWR/                   | SAFE   |        3 |          |     1.27 |
| IRIW-acq-sc             | SAFE   |       16 |          |     1.34 |
| IRIWish                 | SAFE   |       28 |          |     1.36 |
| JLT/                    | SAFE   |        8 |          |     1.23 |
| LB                      | SAFE   |        3 |          |     1.24 |
| LB+acq                  | SAFE   |        3 |          |     1.24 |
| LB+addr                 | SAFE   |        3 |          |     1.34 |
| LB+addr-fun             | SAFE   |        3 |          |     1.30 |
| LB+addr-ind-fun         | SAFE   |        3 |          |     1.27 |
| LB+ctrl                 | SAFE   |        3 |          |     1.26 |
| LB+ctrl+rel             | SAFE   |        3 |          |     1.27 |
| LB+dep                  | SAFE   |        3 |          |     1.30 |
| LB+fr-fr-data+data/     | SAFE   |       19 |          |     1.26 |
| LB+incMPs               | SAFE   |       15 |          |     1.26 |
| LB+rel                  | SAFE   |        3 |          |     1.29 |
| LB2                     | SAFE   |        4 |          |     1.26 |
| LB3                     | SAFE   |        8 |          |     1.36 |
| MCP-rrw/                | SAFE   |        6 |          |     1.25 |
| MP                      | SAFE   |        3 |          |     1.24 |
| MP+incMP                | SAFE   |        7 |          |     1.22 |
| MP+rel+acq              | SAFE   |        2 |          |     1.27 |
| MP+rel+acqf             | SAFE   |        2 |          |     1.37 |
| MP+relf+acq             | SAFE   |        2 |          |     1.52 |
| MP+relf+acqf            | SAFE   |        2 |          |     1.52 |
| MP+rels+acq             | SAFE   |        4 |          |     1.46 |
```

MP+rels+acqf	SAFE	4		1.25
MP2	SAFE	9		1.29
MPU+rel+acqf/	SAFE	6		1.27
MPU+relf+acq/	SAFE	6		1.26
MPU+relf+acqf/	SAFE	6		1.31
MPU+rels+acq/	SAFE	13		1.26
MPU2+rel+acq	SAFE	14		1.27
MPU2+rel+acqf	SAFE	14		1.39
MPU2+relf+acq	SAFE	14		1.48
MPU2+relf+acqf	SAFE	14		1.40
MPU2+rels+acq/	SAFE	36		1.29
RMWFix	SAFE	4		1.25
RWC+syncs	SAFE	7		1.28
S/	SAFE	4		1.23
S+rel+acq	SAFE	2		1.27
S+rel+acqf	SAFE	2		1.24
S+relf+acq	SAFE	2		1.26
S+relf+acqf	SAFE	2		1.26
S+rels+acq/	SAFE	4		1.26
SB	SAFE	4		1.31
SB+2sc+scf	SAFE	3		1.24
SB+3sc+acq	SAFE	4		1.24
SB+3sc+rel	SAFE	4		1.35
SB+4sc	SAFE	3		1.25
SB+assert	SAFE	3		1.34
SB+rfis	SAFE	4		1.34
SB+scfs	SAFE	3		1.23
TC1	SAFE	3		1.24
TC2	SAFE	4		1.26
W+JW	SAFE	1		1.32
W+RRR+W/	SAFE	20		1.44
W+RWC	SAFE	7		1.45
WRC+dep	SAFE	8		1.29
WWR+2WR/	SAFE	0	8	1.29
WWmerge1	SAFE	17		1.23
WWmerge2	SAFE	8		1.25
Z6+acq/	SAFE	7		1.24
Z6.U/	SAFE	22		1.42
assume-ctrl	SAFE	4		1.29
atomicpo/	SAFE	4		1.39
casdep	SAFE	2		1.28
ccr	SAFE	2		1.37
cii	SAFE	6		1.37
cumul-release	SAFE	8		1.41

```
| default/                | SAFE   |       12 |          |     1.23 |
| detour/                 | SAFE   |        9 |          |     1.26 |
| fr+w+w+w+reads/         | SAFE   |      210 |          |     1.25 |
| inc+inc+RR+W+RR/        | SAFE   |      600 |          |     1.43 |
| inc2w/                  | SAFE   |        6 |          |     1.25 |
| isa2                    | SAFE   |        7 |          |     1.23 |
| malloc-dep              | SAFE   |        2 |          |     1.73 |
| peterZ                  | SAFE   |        7 |          |     1.27 |
| po-loc                  | SAFE   |        3 |          |     1.24 |
| psc-ar                  | SAFE   |        0 |        2 |     1.30 |
| psc-sbhbsb              | SAFE   |        1 |        1 |     1.30 |
| rel-B-cumul-acq         | SAFE   |       16 |          |     1.23 |
| rel-pporf/              | SAFE   |        4 |          |     1.31 |
| relacq-B-cumul          | SAFE   |        8 |          |     1.40 |
| relseq/                 | SAFE   |       20 |          |     1.24 |
| rfi-preserved           | SAFE   |        4 |          |     1.25 |
| rii                     | SAFE   |        3 |          |     1.24 |
| rinc+wr/                | SAFE   |        3 |          |     1.29 |
| riwi                    | SAFE   |        4 |          |     1.31 |
| small0/                 | SAFE   |        9 |          |     1.27 |
| small1                  | SAFE   |        8 |          |     1.32 |
| viktor-relseq           | SAFE   |      180 |          |     1.25 |
| wcii/                   | SAFE   |       24 |          |     1.27 |
| wrw/                    | SAFE   |       22 |          |     1.35 |
| wrw/                    | SAFE   |      300 |          |     1.58 |
| wrw/                    | SAFE   |     6576 |          |     2.32 |
-------------------------------------------------------------------------
--- Test time:  202.66
-------------------------------------------------------------------------


-------------------------------------------------------------------------
---  Testing proceeded as expected. Total time: 407.01
-------------------------------------------------------------------------
```

A.4 Relinche Tests LLVM-15 / Rust 1.69

```
-------------------------------------------------------------------------
--- Preparing to check conformance in RELINCHE under RC11 with MO
-------------------------------------------------------------------------

-------------------------------------------------------------------------
| Testcase                  | Spec | Result | Execs    | Hints    |
```

ms-queue-dyn2x2	v	SAFE	12	29
ms-queue-dyn2x2 (2)	vi	SAFE	12	24
ms-queue-dyn2x2 (3)	vr	SAFE	12	33
ms-queue-dyn3x3	v	SAFE	120	906
ms-queue-dyn3x3 (2)	vi	SAFE	120	762
ms-queue-dyn3x3 (3)	vr	SAFE	120	1050
ms-queue-dyn3x3 (4)	vir	SAFE	120	750
ms-queue-dynNxN	v	SAFE	3	3
ms-queue-dynNxN (2)	v	SAFE	12	29
ms-queue-dynNxN (3)	v	SAFE	20	83
ms-queue-dynNxN (4)	v	SAFE	120	906
ms-queue-dynNxN (5)	v	SAFE	210	2754
ms-queue-dynNxN (6)	v	SAFE	1680	42288
ms-queue-dynNxN (7)	vi	SAFE	3	3
ms-queue-dynNxN (8)	vi	SAFE	12	24
ms-queue-dynNxN (9)	vi	SAFE	20	74
ms-queue-dynNxN (10)	vi	SAFE	120	762
ms-queue-dynNxN (11)	vi	SAFE	210	2418
ms-queue-dynNxN (12)	vi	SAFE	1680	31488
ms-queue-dynNxN (13)	vr	SAFE	3	3
ms-queue-dynNxN (14)	vr	SAFE	12	33
ms-queue-dynNxN (15)	vr	SAFE	20	93
ms-queue-dynNxN (16)	vr	SAFE	120	1050
ms-queue-dynNxN (17)	vr	SAFE	210	3072
ms-queue-dynNxN (18)	vr	SAFE	1680	40044
tb-stack-dyn.sync_fail	vi	SAFE	1680	40044
tb-stack-dyn.sync_fail (2)	vr	SAFE	1680	40044
tb-stack-dyn.sync_fail (3)	vi	SAFE	1680	40044
tb-stack-dyn.sync_fail (4)	vr	SAFE	1680	40044
tb-stack-dyn2x2	v	SAFE	20	43
tb-stack-dyn3x3	v	SAFE	408	3126
tb-stack-dynNxN	v	SAFE	20	43
tb-stack-dynNxN (2)	v	SAFE	36	141
tb-stack-dynNxN (3)	v	SAFE	408	3126
tb-stack-dynNxN (4)	v	SAFE	4224	55716
tb-stack-dynNxN (5)	v	SAFE	13104	453748

--- Test time: 258.77

A.5 Evaluation results for Crossbeam

A.5.1 ArrayQueue

Test 0

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 148
Number of blocked executions seen: 12
Total wall-clock time: 5.01s
```

Test 1

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 33571
Number of blocked executions seen: 10623
Total wall-clock time: 15.87s
```

Test 2

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 2209506
Number of blocked executions seen: 6995964
Total wall-clock time: 1062.42s
```

Test 3

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 613362
Number of blocked executions seen: 195418
Total wall-clock time: 299.19s
```

Test 4

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 1
Total wall-clock time: 8.50s
```

Test 5

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 12
Number of blocked executions seen: 3
Total wall-clock time: 4.29s
```

A.5.2 SegQueue

Test 0

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 584
Number of blocked executions seen: 50
Total wall-clock time: 6.73s
```

Test 1

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 9368
Number of blocked executions seen: 1864
Total wall-clock time: 12.74s
```

Test 2

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 1571202
Number of blocked executions seen: 1136762
Total wall-clock time: 404.50s
```

Test 3

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 311152
Number of blocked executions seen: 147294
Total wall-clock time: 240.93s
```

Test 4

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 1
Total wall-clock time: 10.50s
```

Test 5

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 24
Number of blocked executions seen: 6
Total wall-clock time: 3.81s
```

A.5.3 Michael-Scott Queue

Test 0

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 32
Number of blocked executions seen: 4
Total wall-clock time: 4.52s
```

Test 1

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 835
Number of blocked executions seen: 26
Total wall-clock time: 7.63s
```

Test 2

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 3238018
Number of blocked executions seen: 485815
Total wall-clock time: 464.72s
```

Test 3

```
LLVM ERROR: ERROR: Constant unimplemented for type: { i32, i32 }
Aborted
```

Test 4

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 1
Total wall-clock time: 7.35s
```

A.5.4 Treiber's Stack

Test 0

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 14
Number of blocked executions seen: 2
Total wall-clock time: 5.12s
```

Test 1

```
*** Verification complete.
No errors were detected.
Number of complete executions explored: 24
Number of blocked executions seen: 176
Total wall-clock time: 5.96s
```

Bibliography

[1] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Nidhugg github. https://github.com/nidhugg/nidhugg/blob/master/tests/smoke/C-tests/treiber_pop.inc, 2022. Accessed: 2025-07-23.

[2] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. *SIGPLAN Not.*, 46(1):55–66, January 2011.

[3] Tokio Contributors. Tokio github repository. https://github.com/tokio-rs/tokio/blob/master/tokio/src/sync/tests/loom_mpsc.rs, 2024. Accessed: 2025-08-05.

[4] Tokio Contributors. Loom github repository. https://github.com/tokio-rs/loom, 2025. Accessed: 2025-07-20.

[5] The Crossbeam Project Developers. Crossbeam github repository. https://github.com/crossbeam-rs/crossbeam, 2025. Accessed: 2025-07-12.

[6] Free Software Foundation. memcpy_chk.c. GNU glibc source repository, 2024. Commit 6f8628ab945c5e8d2738f7382238c1e1006afe41.

[7] Linux Foundation. Linux standard base core specification 4.0: memcpy_chk. Online HTML documentation, 2008.

[8] Pavel Golovin, Michalis Kokologiannakis, and Viktor Vafeiadis. Relinche: Automatically checking linearizability under relaxed memory consistency. *Proc. ACM Program. Lang.*, 9(POPL), January 2025.

[9] The Open Group. *The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024*. The IEEE and The Open Group, 2024 edition.

[10] Maurice Herlihy. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2nd edition, 2020.

[11] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, 325462-072us edition, May 2020. Chapter 11 Programming with Intel® Streaming SIMD extensions 2 (Intel® SSE2), Section 4.4.4 Pause.

[13] Michail Kokologiannakis. *Automated Reasoning under Weak Memory Consistency*. doctoralthesis, Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau, 2024.

[14] Michalis Kokologiannakis and Viktor Vafeiadis. Genmc: A model checker for weak memory models. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 427–440, Cham, 2021. Springer International Publishing.

[15] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, page 267–275, New York, NY, USA, 1996. Association for Computing Machinery.

[16] Michael Kerrisk. *futex(2) - Linux manual page*, 2024. Accessed: 2025-07-24.

[17] Brian Norris. Cdschecker: Model checker benchmarks for mpmp-queues. https://github.com/computersforpeace/model-checker-benchmarks/blob/master/mpmc-queue/mpmc-queue.cc, 2013. Accessed: 2025-07-23.

[18] Brian Norris. Cdschecker: Model checker benchmarks for ms-queue. https://github.com/computersforpeace/model-checker-benchmarks/blob/master/ms-queue/main.c, 2013. Accessed: 2025-07-23.

[19] Brian Norris and Brian Demsky. Cdschecker: checking concurrent data structures written with c/c++ atomics. *SIGPLAN Not.*, 48(10):131–150, October 2013.

[20] Oliver Pearce, Julien Lange, and Dan O'Keeffe. Rustmc: Extending the genmc stateless model checker to rust, 2025.

[21] Sujal Singh. Github: Formal verification of lock-free concurrent queue. https://github.com/sujalsin/concurrent-verification/blob/main/src/main.rs, 2024. Accessed: 2025-07-23.

[22] Robert Tarjan. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 114–121, 1971.

[23] LLVM Development Team. Llvm api doxygen documentation. https://llvm.org/doxygen/, 2025. Accessed: 2025-07-22.

[24] LLVM Development Team. Llvm language reference. https://llvm.org/docs/LangRef.html, 2025. Accessed: 2025-06-09.

[25] LLVM Development Team. Llvm language reference. https://llvm.org/docs/Passes.html, 2025. Accessed: 2025-07-24.

[26] The KLEE Team. Klee github repository. https://github.com/klee/klee, 2025. Accessed: 2025-06-15.

[27] The Rust Programming Language Team. rust-bindgen: Automatically generates rust ffi bindings to c (and some c++) libraries. https://github.com/rust-lang/rust-bindgen, 2025. Accessed: 2025-04-24.

[28] The Rust Programming Language Team. Rust programming language. https://github.com/rust-lang/rust, 2025. Accessed: 2025-04-24.

[29] The Rust Programming Language Team. The rust programming language team. https://github.com/rust-lang/miri, 2025. Accessed: 2025-07-22.

[30] The Rust Programming Language Team. The rust reference. https://doc.rust-lang.org/reference, 2025. Accessed: 2025-07-22.

[31] The Rust Programming Language Team. The rust standard library. https://doc.rust-lang.org/std/, 2025. Accessed: 2025-04-24.

[32] The Rust Programming Language Team. The rustc book. https://doc.rust-lang.org/rustc, 2025. Accessed: 2025-07-22.

[33] Robert K. Treiber. Systems programming: Coping with parallelism. Research Report RJ 5118, IBM Almaden Research Center, 1986.

[34] Dmitry Vyukov. Bounded mpmc queue. https://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue, 2021. Accessed: 2025-07-23.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. **In consultation with the supervisor**, one of the following two options must be selected:

☐ I hereby declare that I authored the work in question independently, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

☒ I hereby declare that I authored the work in question independently. In doing so I only used the authorised aids, which included suggestions from the supervisor regarding language and content and generative artificial intelligence technologies. The use of the latter and the respective source declarations proceeded in consultation with the supervisor.

Title of paper or thesis:

A Rust Frontend for GenNC

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Kamberi	Arbenit

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guidelines.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zurich, 14.07.25	Kamberi

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] For further information please consult the ETH Zurich websites, e.g. https://ethz.ch/en/the-eth-zurich/education/ai-in-education.html and https://library.ethz.ch/en/researching-and-publishing/scientific-writing-at-eth-zurich.html (subject to change).