ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Enabling Automatic Verification of Concurrent Data Structures

Master Thesis

Christof Leutenegger

April 28, 2025

Advisors: Prof. Dr. M. Kokologiannakis

Department of Computer Science, ETH Zürich

Abstract

Writing concurrent code is hard, especially when using weak memory models. Concurrent libraries such as Concurrency-Kit simplify this by providing correct concurrent data structures. However, the correct implementation of these concurrent data structures is often plagued by hard-to-trigger bugs. Model checkers like GenMC automate finding or verifying the absence of such bugs. It does so by enumerating and checking all possible executions of a concurrent program under the weak memory model.

In the first part of this thesis, we adapted the source code of Concurrency-Kit to enable the verification with GenMC. Most importantly, we provided a C11 mapping of the library's concurrency primitives so that we can compile the data structures to LLVM, which is currently the input format that GenMC supports. We verify Concurrency-Kit using three clients providing bag, `FIFO`, and `LIFO` semantics, allowing us to identify two bugs in their data structures.

In the second part, we extend GenMC to verify data structures that use randomness in their algorithm, by introducing data non-determinism. We model the non-determinism using concolic expressions and use an SMT solver to explore all the different paths that the non-deterministic value can trigger. We show that introducing non-deterministic values can simplify and speed up the verification of data structures, such as list-based sets.

Contents

Chapter 1

Introduction

Writing correct concurrent code is difficult. Not only is it hard to think of all the different interleaving that can occur while running code in parallel, but to make matters worse, modern platforms employ weak memory models, allowing the compiler or the CPU to reorder the instructions in a thread. Synchronization bugs commonly exist undetected in a code base and only appear when using different hardware or are just very unlikely to appear. For these reasons, tests are insufficient to detect these hard to spot bugs. Instead concurrent libraries exist that lift the complexity of writing correct concurrent code from the programmer. These libraries implement well-established synchronization primitives, such as barriers and locks and concurrent data structures that are formally proven to be correct. However, the proofs of the synchronization primitives and concurrent data structures often assume sequential consistency [1] for their correctness, and the implementation in weak memory models is usually argued informally. Model checkers are built exactly for such use cases. They enumerate all possible executions of a program and verify that the specified properties hold for all of them. One of these model checkers is GenMC [2], which specializes in verifying concurrent code with weak memory models. The algorithm used by GenMC [3] explores no equivalent execution twice and uses as a stateless model checker only a linear amount of memory. GenMC is limited by the number of threads, as the number of executions to be explored in most cases increases exponentially with the number of threads. With the *small scope hypothesis* that assumes that a high proportion of the bugs are found by testing the program within a small scope, testing the program with a small number of threads is often enough.

In the first of part of this thesis we verify the concurrent data structures in the concurrent C library *Concurrency-Kit* (ck) using GenMC. For this, we make the library code verifiable by GenMC. For this, we implement the ck concurrency primitives that have been implemented so far in assembly in atomic memory operations provided by C11. This is necessary as GenMC takes LLVM code as input to its verification process. We verify data structures with three clients that check for bag, `FIFO`, and `LIFO` semantics. This way, we found two bugs in ck, for which we also provide the code to fix them.

In the second part of this thesis, we extend GenMC with support for non-deterministic values. Currently that is missing, as for the model checker to work, the exploration of the different executions must be deterministic. We model the value non-determinism using concolic expressions, a combination of a concrete value and symbolic expressions. This allows us to efficiently execute a program on the one hand and explore an alternative execution using symbolic expressions on the other. We show that the new extension does not induce an overhead on programs that do not use it. Further, we provide measurements that show non-deterministic values can reduce the number of executions by avoiding executions with different concrete values but the same symbolic expressions.

The rest of this paper is structured as follows: In §2, we provide the required background about weak memory models and the GenMC model checker. In §3, we describe the verification of the data structures of the ck library. In §4 we describe and evaluate our implementation of data-nondeterminism in GenMC. The §3 and §4 are independent of each other and can be read independently.

Chapter 2

Background

In this chapter, we provide the needed background on memory models and the difficulties in writing correct and fast programs. Then, we look at how the GenMC model checker approaches the verification of programs written in weak memory models.

2.1 Memory Models

Memory models define how different threads of a program interact with each other. From a programmer's perspective, the memory model specifies how memory accesses are ordered and the values that can be expected by a load on a shared memory location. Memory models exist at every level between the programmer and the processor, ranging from the high-level programming language to the processor hardware. The specification of memory models is notoriously difficult. For example, the Java memory model needed multiple iterations [4], [5], [6]). The C/C++ memory model was first defined in C11/C++11 and then fixed with RC11 [7]. Prior to C11, the semantics of threaded code of a C program were unspecified, making it impossible to write correct and portable code using C alone. In order to circumvent this issue, libraries such as ck map threaded operations down to assembly code using the memory models of the machines, using the memory models of the machine directly.

To see how memory models can go against our intuition about parallel execution, let us observe Figure 2.1 The figure shows a possible execution of a very small program as a graph. Two threads (Thread 0 and Thread 1) interact with the shared variables (x and y), both writing to and reading from these variables. Both shared variables are initialized to 0. Thread 0 first writes 1 to x and then reads 0 from y. Conversely, Thread 1 first writes 1 to y and then reads 0 from x. Intuitively, this would be impossible since either Thread 1 writes x before Thread 2 reads x or Thread 2 writes y before

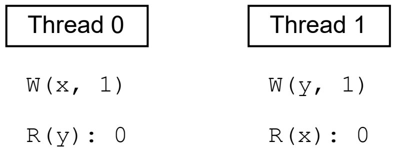

Figure 2.1: Store buffer (SB) example

Thread 1 reads it. This intuitive understanding is best captured in sequential consistency (SC) [1], which ensures that all instructions follow a global total order. However, all major memory models allow for the behavior shown in Figure 2.1. Why is this the case? The reason is performance. Writes within hardware take many cycles, and if they are deferred, we get a performance gain because the hardware is able to do other work (such as reading another variable). To the programmer, it appears as if the read is reordered before the write.

When a memory model permits more possible program executions, it is called *weaker*. A weaker memory model allows for more optimizations. For example, the C11 memory model allows for more optimizations in the compiler because we can reorder or even drop whole operations. While we get the increased performance, the difficulty of writing a correct program increases. To ensure correctness, memory models rely on special operations that disallow possible executions that would invalidate the assumptions of a program.

C11/C++11 memory model

Here, we give a brief introduction to the C11/C++11 memory model. For a more detailed discussion, see [7] and [8]. In C11/C++11, multiple threads can communicate with each other using *atomic* variables. Any communication between threads involving a concurrent read from and a concurrent write to a non-atomic variable triggers a data race that causes undefined behavior and can be considered a bug.

The C11/C++11 memory model provides four basic access modes for accessing atomic variables.

- *relaxed access:* The weakest access mode. The memory model does not impose any order on the operations of the program. It just stops data races from causing undefined behavior.
- *release-consume:* An access mode that is rarely used and is deprecated with C++26, so we will not discuss it here.
- *release-acquire:* Helps to synchronize two operations in different threads

with one another. If we have an acquire-read that reads from a release-write, it establishes a happens-before relation. The happens-before relation makes all operations before the release-write visible.

- *sequential-consistence:* Like release-acquire, reads can synchronize with writes with the happens-before relation. Additionally, as mentioned before, we get a total global order of all atomic operations. Sequential-consistency is the strongest access mode but also disallows the most optimizations.

Most parallel code is fine with using sequential consistency to synchronize its threads or, even better, using concurrent data structures to do so. If more performance is needed, the memory accesses can be weakened to release-acquire or even relaxed, with the trade-off that reasoning about correctness becomes harder.

Finally, to iterate the difference between release-acquire and sequential consistency, let us discuss Figure 2.2.

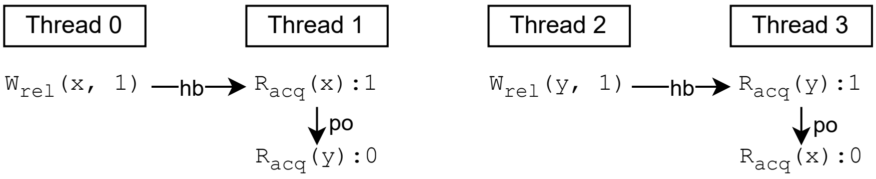

Figure 2.2: Concurrent program with independent reads and independent writes (IRIW)

Compared to Figure 2.1, we add two more pieces of notation. Firstly, we add `rel` and `acq` to the memory operations specifying what access mode they are using. Secondly, we add the relations, here `hb` (happens-before) and `po` (program-order), between the events to signal where and which threads are synchronizing. In C11, the programmer can always assume that the operations within a thread appear to happen in program order.

The execution in Figure 2.2 is allowed as the writes and the reads are independent. Thus, we do not have a happens-before relation between the read of y in thread 1 and the write of y in thread 2. If we change the access mode to sequential-consistency instead, this execution becomes disallowed. As there is now a total global order that forces the writes in an order, both threads 1 and 3 will observe that order as well.

2.2 GenMC Model Checker

GenMC [2] is a state-of-the-art stateless model checker for concurrent programs. Its goals are:

- *Generality*: GenMC should work for different languages and memory models.
- *Efficiency*: The state space exploration should be efficient and not re-explore already explored states.
- *Usability*: In case of errors, e.g., data races, GenMC should report understandable error messages
- *Extensibility*: GenMC should be easily adaptable to ensure the incremental improvement of the other three goals.

Modular Design

GenMC's architecture is split across different components that communicate with each other via clear interfaces. This makes it possible to work on single components, improving them without affecting the other components or even swapping them out completely, e.g., to support a new memory model.

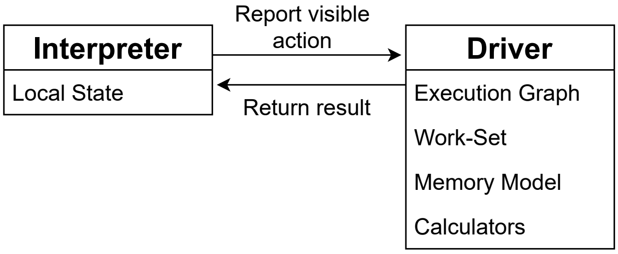

Figure 2.3: Components of GenMC and communication between Interpreter and Driver

In Figure 2.3, we see the core components of GenMC and the main interaction between the driver and the interpreter in the verification loop.

First, the components. We have the interpreter, of which only one implementation exists so far. It runs LLVM-IR code. Everything local to threads is stored directly in the interpreter itself. In the driver, we store and coordinate different components. We store the events of the current executions (such as reads from and writes to memory) in the *execution graph*, possible alternative states, we encounter while running the programming, we store in the *work-set*, the *memory model* is used to determine what executions are possible and should be explored, and finally the calculators, which provide the different relations to the memory model, such that it is able to make its decisions.

The verification loop works as follows: The interpreter runs the program until it reaches a statement for which it needs help from the driver. For example, when a thread starts/finishes or the program reads from or writes to memory, it yields to the driver. The driver then takes this action and enters it in the execution graph, depending on the action and the memory model there can exist multiple possible return values, the driver decides for one and stores possible other executions in the work-set. Then, the result is returned to the interpreter. The interpreter and the driver work in tandem as coroutines. This process repeats until the interpreter finishes, at which point the driver checks if there are unexplored executions. If that is the case, it restarts the interpreter, which now explores the other execution.

Maximality

At the core of GenMC efficiency lies the TruSt (Truly Stateless) algorithm [3]. The algorithm allows to explore the state space optimally (only one exploration per equivalence class) with only linear memory requirements. The memory models can be easily swapped out as they only filter the search space considered by the algorithm. The ingenious idea of the algorithm is to choose a special execution based on a notion of *maximality* and use it to ensure that there is exactly one exploration per equivalence class. To explain the maximality, we must first explain how GenMC explores the state space.

Forward and Backward Revisit For simplicity, let us assume that we only have reads and writes. Then, as mentioned before, each time we add a new event, we could have many possible choices on how to add it to the graph.

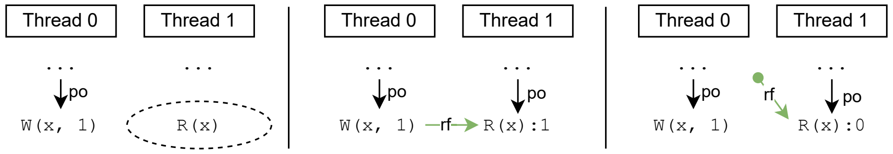

Figure 2.4: Choice of where to read-from (rf) the value for a newly added read.

See Figure 2.4, there we add a new read to thread 1, this read can either read from the write in thread 0 or read from the initial event. The driver decides on one and stores the other back in the work-set. When the interpreter finishes the execution, the driver returns to the time we inserted the read, removing all other events that were added afterward. Then, it sets the read-from edge accordingly and starts the interpreter anew. This is the simple case. We only revisit different executions and are not able to end up in the same state twice. This simple case is called *forward revisit*.

The other case is a *backward revisit*, which happens when we add a write to the graph and if there is already a read in place. Theoretically, the read could also come from the new write, so we need to explore that possibility as well. But we cannot just swap the read-from edge to the new write, because other events could depend on that value. Instead, the chosen approach is to remove all other events except those on which the read and the new write depend to re-create a graph in which the new write was inserted before the read. This deletion step is where the problem with the backward revisit lies.

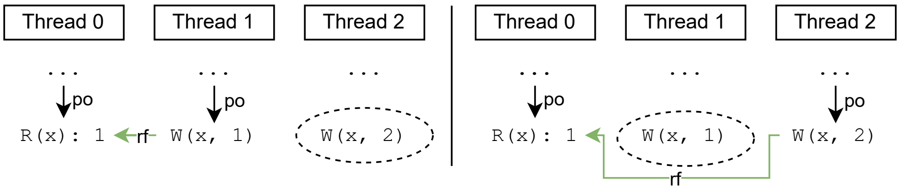

Figure 2.5: Choice of where to read-from (rf) the value for a newly added read.

Consider the left part of Figure 2.5, where thread 2 adds a write that the read in thread 0 could read from. So, in a backward revisit, we remove all other events except the ones the read and the new write depend on and set the read-from edge accordingly. If, at some point (right side of the figure), we re-add the write that we removed from thread 1, we could backward revisit the read again, eventually leading to the same result as on the left side of the figure again. We are stuck in a loop.

To avoid this, TruSt defines the notion of a maximal extension. Intuitively, it works as follows: A graph is maximally extended if the added writes and reads follow the insertion order. The insertion order is a fix total order in which we add events to the graph. If we add a new write, it overwrites the value of the last write in the insertion order with the same memory location. If we add a read, it gets its value from the last write to that memory location in insertion order. If the graph is not maximal (not all events are added as maximal extension) then we don't consider them for a backward revisit. For Figure 2.5, neither the left nor the right graph is maximal because the read gets its value from a write that was added after it in insertion order (assuming left-to-right insertion order). If there were no maximal graph, that would also cause some problems, as it would make valid backward revisits impossible. But the paper proves that there is exactly one such maximal execution, allowing for backward revisits to explore every state. Additionally, backward revisits are unable to re-trigger other revisits, as a backward revisit makes a graph automatically non-maximal. Since the maximality definition requires no more additional memory as it is computed out of the execution graph, which we store anyway, the whole TruSt algorithm requires $O(n)$ memory.

2.2.1 Formal Definition

While for the background, we are mostly interested in the intuitive understanding of GenMC, we want to close this section with a formal definition of events and execution graphs that we copy from [3].

Def 2.1 An event $e \in$ Event is either the initialization event `init`, or a thread event $\langle t, i, lab \rangle$, with $t \in Tid$ being the thread identifier, $i \in Idx$ the index within the graph, and $lab \in$ Lab one of the following labels:

- *Write label*: $W(l, v)$ where $l \in$ Loc is the location accessed, and $v \in Val$ the value written.
- *Read label*: $R(l, v)$ where $l \in$ Loc is the location accessed.
- *Error label*: denoting safety violation

In the following R denotes the set of all reads $\{\langle t, i, lab \rangle | lab = R(\_)\}$, W denotes the set of all writes $\{\langle t, i, lab \rangle | lab = W(\_, \_)\} \cup \{\text{init}\}$, and W_l denotes the set of all writes to the same location $\{\langle t, i, lab \rangle | lab = W(l', \_) \wedge l' = l\} \cup \{\text{init}\}$.

Def 2.2 An execution graph G consists of:

1. a set E of events with init $\in E$ and duplicate events with the same thread identifier and thread index.
2. a total order $\leq_G$ on E representing the insertion order to the graph.
3. function rf $\in R \times W$ called the reads-from function, which maps the read to the place where to the write, where it gets its value
4. a strict partial order co $\subseteq \bigcup_{l \in \text{Loc}} W_l \times W_l$

And now for the definition of how to add an event maximally

Def 2.3 co-maximality for an event is defined as:

- A write event $w \in W$ is co-maximal $\neg\exists w' \in W$ such that $\langle w, w' \rangle \in$ co
- A read event $r \in R$ is co-maximal if rf(r) is co-maximal

The intuition behind co-maximality is that either the new event overwrites or reads from the last write to the same location.

Def 2.4 Event e is *maximally* added before write event w if

1. e is co-maximal to $\text{Previous}_G(e, w) ::= \{e' \in E | e' \leq_G e \vee \langle e', w \rangle \in \text{porf}\}$
2. There exists no $r \in \text{Previous}_G(e, w)$ such that rf $= e$

This definition now uses co-maximality on the prefix of a write event in case of a backward revisit. It follows the intuition we established before in the non-formal part of the background. New is only porf that is the transitive closure of the union of program order po and read-from function rf.

Chapter 3

Verifying the Concurrency Kit Data Structures

Writing parallel computing code remains notoriously difficult, with many bugs arising from improperly synchronized code. As we saw in 2.1, weak memory models make this even worse since each thread's instructions can be reordered either by the compiler or by the hardware.

Concurrent libraries can mitigate some of these challenges by providing thread-safe implementations of standard data structures such as queues and stacks. These building blocks allow the programmer to write programs without worrying about the correctness of their parallel code sections.

Concurrency-Kit (ck) is one of these libraries. It is hosted on github and has been written and maintained by Samy Al Bahra since 2011. It provides not only the standard data structures but also other primitives upon which the library builds the concurrent data structures. From the concurrency primitives that make it possible to write concurrent and parallel code in the first place (atomic reads and writes, compare-and-swap). Then, the safe memory reclamation mechanisms, such as hazard pointers [9] and epoch-based reclamation [10]. And finally, there are synchronization primitives such as locks and barriers. For better performance, ck implements lock-free data structures and uses the weak memory models of their respective supported platforms.

ck accomplishes this by mapping its concurrency primitives to all architectures it supports. However, this makes it necessary to test these data structures on all the architectures. But that is probably not the case. Some data structures, such as the `bitmap` and `array`, are not tested using multiple threads at all, there is only a single sequential test. This makes the ck library a good target for verification with the GenMC stateless model checker and for testing GenMC's usability in a real-world example.

To do this, we need to build a scaffold for the environment the library assumes so that its code is self-contained and executable with GenMC. This includes writing a mapping to code that the model checker can run, which is LLVM, which is why we use C11 and its concurrent primitives such that we can compile it to LLVM and pass it to GenMC.

Unfortunately, this disallows the verification of the mappings from the concurrency primitive to the assembly of the different architectures. With the scaffolding in place, we can prove functional correctness for the various data structures provided by the library.

3.1 Concurrency Primitives

In this section, we explain what concurrency primitives ck provides and how they map them to assembly in the case of ARMv7. ARMv7 because it is one of the weaker hardware memory models, and thus is able to show more of the semantics of the operations, as behind stronger memory models the intent could be hidden behind the stronger guarantees. While ck provides 140 different primitives, most of them are duplicates with different widths. For a simpler understanding, we divide them into four groups.

Atomic Loads and Stores Atomic loads and stores are available in different widths (8, 16, 32, 64 bits) and C types (`char`, `short`, `int`, `uint`, `ptr`, `double`). In total, there are 10 for each load and store. For the ARMv7 mapping, all operations are mapped to simple load and stores without extra synchronization. The exceptions are the 64 bits and the `double` loads and stores, which get mapped to a `ldrexd` (Load Register Exclusive Doubleword).

Atomic Read-Modify Operations This group includes all operations that load a variable, perform an action, and store it back to memory in one atomic step. This is the largest group of concurrency primitives, with 88 primitives in total. The operations include add, decrement, increment, fetch-and-add, fetch-and-set, negate, bitwise-and, bitwise-or, bitwise-not. For every operation, we have 8 implementations for the different widths and types, missing the 64 bits and the `double` variant. For ARMv7, all operations map to a `ldrex` for the load, followed by the operation, and closed by a `strex`. If the store fails, the code retries in a loop.

Compare and Swap Compare-and-swap is the basis for most wait-free algorithms and is available in many different variants in ck. Again, we have the same widths and types variants as for atomic loads and stores. Notably, some architectures support a double-width CAS that atomically swaps a two-entry 64-bit array with another two-entry 64-bit array. For every width and type, two CAS variants exist, one that returns a boolean on success and another that additionally stores the old value back into a pointer. The ARMv7 implementation is similar to the atomic read-modify operations, without the loop in case the `strex` fails.

Memory Fences The attentive reader observes that all operations so far did not include any inter-thread synchronization. ck uses memory fences for this. There are many fence variants available. Most fences are defined by the operations before and after the fence. For example, if we want to synchronize a load before the fence, there are different choices available depending on which operation follows after the fence. If we want to synchronize on all operations, we use `ck_pr_fence_load`, if we want to synchronize on the next store `ck_pr_fence_load_store`, and finally `ck_pr_fence_load_atomic` that synchronizes with an atomic read-modify operation. These combinations exist as well if `store` and `atomic` are before the fence. In addition, ck provides fences for `acquire`, `release`, `lock`, and `unlock` which we believe should follow the release-acquire semantics of C11/C++11. And lastly, `ck_pr_fence_memory` is a full memory fence. In comparison, the mapping for ARMv7 is tremendously simple. The store and atomic fences map to DMB ST (Store-only Data Memory Barrier), and all the other fences map to DMB (Data Memory Barrier).

3.2 Data Structures

In Table 3.1, we provide a comprehensive overview of the concurrent data structures available in the ck library. The library implements only ten data structures. Six data structures (`stack`, `hp_stack`, `fifo`, `hp_fifo`, `ring`, `bitmap`) support multiple readers and multiple writers, while the others (`hs`, `ht`, `rhs`, `array`) are limited to a single writer. It is important to note that ck defines a writer in these data structures as a thread that modifies the data structures and the readers as threads that only read from the data structure without any modifications.

The data structures `stack` and `stack_hp` differ by which method they implement safe memory reclamation, while `stack` uses epoch-based reclamation as of [10], `hp_stack` uses the treiber stack adaption with hazard pointers from [9]. `hp_fifo` uses the hazard pointers as well, but for `fifo` the caller must provide their own memory reclamation. This is true also for `ring`, where the caller must provide their own memory reclamation as well. `bitmap` sets and gets bits in an array of integers using accesses without any synchronization at all. `hs`, `ht`, `rhs` are specialization of the same hash-set implementation. `hs` is a hash-set, `ht` is a hash table based on `hs`, and `rhs` utilizes robin-hood hashing for improved performance, which is not the main concern for our verification with GenMC.

ck-Name	multiple readers	multiple writers	note
`stack`	yes	yes	epoch-based stack [10]
`hp_stack`	yes	yes	treiber stack + hazard pointers [9]
`fifo`	yes	yes	ms-queue [11]
`hp_fifo`	yes	yes	ms-queue + hazard pointers [9]
`ring`	yes	yes	fifo-based circular buffer implementation
`bitmap`	yes	yes	
`hs`	yes	no	ck hash-set implementation
`ht`	yes	no	ck hash-table (hash-set specialization)
`rhs`	yes	no	ck hash-set (with round-robin hashing)
`array`	yes	no	

Table 3.1: Concurrency-Kit data structures

3.3 New C11 Mapping for Concurrency Primitives

GenMC currently only provides an LLVM interpreter. However, the ck library uses assembly for its concurrency primitives. To allow for verification, we map these primitives to C11 atomics. This way, we can verify ck data structures without making any major changes to the source code of the data structures.

For the mapping, we closely follow the semantics of the hardware mapping ck provides as described in Section 3.1. As all synchronization is done with memory fences, all other operations use the relaxed access mode. We directly translate load, store, compare-and-swap, and other atomic read-modify operations to their C11 equivalents. But fences are more complicated, as in C11, fences do not synchronize with the next operation but with another fence in another thread. To synchronize a load, both ck and C11 need a fence after the load. This allows us to map the load fences to acquire fences in C11. ck synchronizes a store with a fence after the store, while in C11, we synchronize a store with a fence before the store. As we do not want to change the position of all fences, we map the store fences to acquire-release fences so we can synchronize with the operation following the store instead. The fences for `acquire/lock` we mapped to their fitting acquire-fences, similar we mapped the `release/unlock` fences to release-fences. We map the full memory fence to a sequential-consistency fence.

There is one operation we could not map to a C11 primitive: the double width compare and swap. GenMC has no direct support for it, either. Unfortunately, that means we are not able to verify the `fifo` data structure as the double-width CAS is used for the epoch-based safe memory reclamation.

3.4 Adaption of Source Code for Verification

Missing Data Dependency

For most of the data structures, our first attempt works at mapping the concurrency primitive, excluding some of the adaptions and problems we will talk about further down in this section. But there occurred a problem we found while verifying the hp_fifo data structure with GenMC, which is due on a assumption that ck does regarding data-dependency. A data-dependency is a dependency between value *a* read by a load and value *b* written by a store, where value *b* is based off a calculation on *a* (or just the value *a* itself). It does not exist in the C11 memory model, but in the ARMv7 memory model the data-dependency stops re-orderings and thus has effect on the possible executions.

```
void ck_hp_fifo_enqueue_mpmc(ck_hp_record_t *record,
      struct ck_hp_fifo *fifo,
      struct ck_hp_fifo_entry *entry,
      void *value) {
  struct ck_hp_fifo_entry *tail, *next;

  entry->value = value;
  entry->next = NULL;
  ck_pr_fence_store_atomic();

  for (;;) {
    tail = ck_pr_load_ptr(&fifo->tail);
    ck_hp_set_fence(record, 0, tail);
    if (tail != ck_pr_load_ptr(&fifo->tail))
      continue;

    next = ck_pr_load_ptr(&tail->next);

    if (tail != ck_pr_load_ptr(&fifo->tail))
      continue;

    if (next != NULL) {
      ck_pr_cas_ptr(&fifo->tail, tail, next);
      continue;
    }
    if (ck_pr_cas_ptr(&tail->next, NULL, entry))
      break;
  }

  ck_pr_fence_atomic();
31   ck_pr_cas_ptr(&fifo->tail, tail, entry);
32   return;
33 }
34
35 void ck_hp_set_fence(struct ck_hp_record *record,
36                      unsigned int i, void *pointer)
37 {
38   ck_pr_store_ptr(&record->pointers[i], pointer);
39   ck_pr_fence_memory();
40 }
```

Listing 3.1: Fifo enqueue with hazard-pointers

To show this assumption and what this means for our mapping, we consider an example based on the Listing 3.1. The listing contains the code for the enqueue method for the `hp_fifo` data structure. In lines 8-10 we set the new values of entry and protect them by a fence. In lines 13-21 we get `fifo.tail` and `fifo.tail.next`, protect the access to the `fifo.tail` with a hazard point and then a full-memory fence (see lines 38-39). If pointers gets out of date, tested with the if branches (line 15 and line 19), we retry. Lines 22-24 are helping another thread if it did not manage to advance the tail pointer in time (Line 30-31). And finally, line 26 inserts the entry into the queue.

Problems arise if we have 3 threads concurrently write to the queue. The first thread runs before the other two. It allocates the new entry before entering the function call, and succeeds in inserting the entry into the queue, but stops before advancing the tail pointer. The second thread reads from the first thread and sees that `fifo.tail` does not point to the last entry and helps advancing the tail pointer and then stops. The third thread then triggers a potential bug by reading the `fifo.tail` from the second thread. Because of this hand-over from first thread to second thread and then second thread to third thread, there is no proper synchronization between the first and third thread.

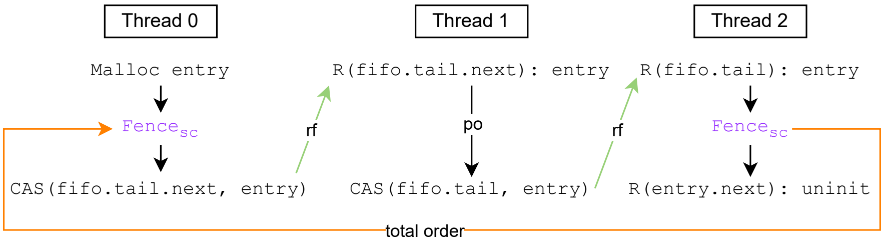

Figure 3.1: Three concurrent writer in ck FIFO implementation with possible C11 execution

This situation we see in Figure 3.1 as well. Thread 1 reads from (rf) Thread

0 and Thread 2 reads from Thread 1 but there is no ordering that creates a happens-before relation, thus we can observe this time-bending effect, where we read that `entry` is not even allocated.

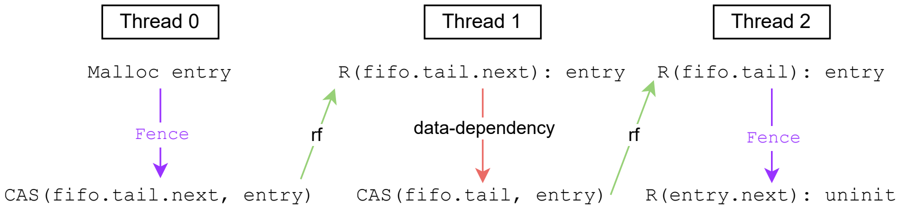

Figure 3.2: Three concurrent writer in ck FIFO implementation with illegal ARMv7 execution

If we change from the C11 memory model to the ARMv7 memory model, see Figure 3.2, the situation changes. Now, we have a data-dependency between the read and the compare-and-swap in thread 1. With the data-dependency, the execution is not allowed under the ARMv7 memory model. This is not only true for ARMv7 but for all architectures ck supports, so we do not consider this behavior a bug.

There are two possibilities to adapt to that. The first possibility is to change the mapping of the load and store concurrency primitives from using the relaxed access mode to acquire for the loads and release for the stores. With the release-acquire semantics, the threads synchronize where we now have a rf relation, further disallowing the execution in Figure 3.1. Another approach is using the `ck_pr_fence_load_depends`, which is currently a NOOP in all architectures. If we use sequential-consistency in the C11 mapping instead and insert it where the data-dependency is missing, we are able to synchronize the data structure without influencing the other architectural mappings.

We choose the extra fence we insert in 3.1 between lines 22-23, as it introduces less synchronization overall. Using acquire-release semantics everywhere could hide bugs due to the additional synchronization not intended in the ck library. If the ck library wants to support C11 semantics directly, it would be advantageous to introduce `store_release()` and `load_acquire()` primitives, which would allow us to rewrite this data structure in C11 style instead of depending on the hack of introducing the `ck_pr_fence_load_depends` fence.

Mixed Sized Accesses

At the time of writing, GenMC struggles with mixed-sized accesses. If a program reads and writes to an address with different sizes, GenMC will generate an error. In many cases, this is a false positive. For example, an optimization that replaces two 32-bit writes into an array with one 64-bit write is allowed. Mixed accesses are triggered in ck by explicitly using memcpy and memset to manipulate the states of its data structures. GenMC sees these functions as 8-bit accesses on the data of the data structures, which are mostly pointers and not bytes. To handle that, we introduce conditional compilation that, in case of verification, replaces the memcpy and memset with a for-loop managing the memory operations.

```
#if defined(VERIFICATION)
  for (size_t j = 0; j < array->n_entries; j++)
    target->values[j] = array->active->values[j];
#else
  memcpy(target->values, array->active->values,
   sizeof(void *) * array->n_entries);
#endif
```

Listing 3.2: Conditional compilation replacing memset

Hazard Pointers

For both hp_fifo and hp_stack exists a hazard pointer implementation in the ck library. Instead of using these, we swap them out for the hazard-pointer mock-ups provided by GenMC. As with this mock-up, we get improved verification performance and it allows us to focus on the data structure implementations.

3.5 Verification Infrastructure

For the verification of the different data structures we implement a shared interface, which is used by three different clients. We keep the interface simple and hide different implementation choices behind it.

```
void client_init(unsigned nr_threads);
void client_insert(int value);
int client_remove(void **garbage_bin);
void client_garbage_cleanup(void *garbage);
void client_cleanup(void);
```

Listing 3.3: Client interface for the verification of ck data structures

We can see the interface in Listing 3.3. For the data type, we choose int, which we insert with client_insert and get back with client_remove. Special is the void **garbage_bin input for client_remove. We need the void **garbage_bin pointer in case the data structure does not implement safe memory reclamation. This is the case for the fifo data structure, where we handle it by simply postponing the freeing to the end of the execution, at which point we use client_garbage_cleanup.

As for the clients, we implemented three different variants:

1. *Bag semantics*, with this client, we check that all reads are coming from a valid write; duplicates are allowed.
2. *FIFO semantics*, with this client, we check that all reads follow the insertion order.
3. *LIFO semantics*, with this client, we check that all reads follow the inverse insertion order.

Anytime a read cannot get a value from the data structure, e.g., if the data structure is empty, we return a value noise value. The semantic checks ignore noise values. This ensures we can validate executions where the reads are performed before the writes.

The bag client we implement for `stack`, `hp_stack`, `hp_fifo`, `ring`, `array`, and `bitmap`. The FIFO client we implement for `hp_fifo`, `ring`, and `ring`. The LIFO client for `stack`, and `hp_stack`.

All clients are adaptable by the number of reader threads, number of writer threads, number of noise threads, how many reads a reader thread performs, and lastly, how many writes a writer thread performs. The noise threads either remove and drop a value or write a noise value to the data structure. It decides that based on some deterministic random value given by the GenMC function `__VERIFIER_nondet_int()`.

Some data structures do not fit well into the interface in Listing 3.3. The `bitmap` implementation ck does not provide any orderings between operations. Thus, we chose to iterate the readers from the least-significant bit, removing the first bit that is set, leading to a FIFO order, but also the possibility of removing a bit multiple times. For `array`, only one concurrent modifier of the data structure is allowed. Thus, instead of removing an entry, we iterate through the array, returning the last entry set, possibly multiple times.

The hash set implementations do not fit this model at all, as readers cannot access any random value but must provide the key to access the set entries. Further, we encountered a problem in catching mixed-sized access in the hash function. Additionally, the documentation does not mention which methods are safe to use for the readers, resulting in scrapping code, which we were unsure would work. This limited our time available for testing these data structures drastically, leading us to decide only trying to verify `hs`, as `rhs` and `ht` are just specializations of `hs`. For this purpose, we built a small special client that only uses `put` and `get`, two methods we are sure can be used by a single writer and multiple concurrent readers.

The `fifo` data structure we do not verify due to the already mentioned double-width compare and swap.

3.6 Verification Result

Client	Data-Structure	W	R	Noise	Executions	Blocked	Time [sec]
bag	stack	3	3	-	40 296	35 072	51.23
		5	0	-	122 880	0	122.67
	hp stack	3	3	-	64 920	17 9912	170.10
		5	0	-	122 880	0	159.26
	hp_fifo	3	3	-	43 434	1 181 360	853.54
		5	0	-	687 960	2 593 905	1591.59
	ring	3	3	-	31 992	24 110	65.48
		5	0	-	60	3 929	4.60
	array	1	3	-	8	0	0.10
	bitmap	3	3	-	8 124	0	10.80
		5	0	-	120	0	0.47
fifo	hp_fifo	2	2	1	1 104	20 068	13.06
	ring	2	2	1	31 992	24 079	64.16
	bitmap	2	2	1	590	0	0.79
lifo	stack	2	2	1	7 152	1 324	6.40
	hp_stack	2	2	1	11 128	10 224	16.12

Table 3.2: Measurement of data structures that were tested with clients of Section 3.5.

Table 3.2 shows most of our results from the data structures we verified. The biggest group is the bag client, the client with the least requirements, as it only tests whether a number comes from a writer. This group we tested in two ways: first, a run with the same number of writers and readers, and then a run with only readers. This allows us to increase the number of writers without running into an exponential runtime that we cannot check anymore. We want to highlight again that the `array` data structure only supports one thread that modifies the data structure. Thus, we do not execute the run with only writers and only use one writer in the other.

Then we have the other two clients `fifo` and `lifo`, which we run with an additional noise thread. It is important to note that the times reported in the table are not tested exhaustively and are meant as general information about how long the verification took. However, not all verifications went through, and we also found some bugs in the data structures.

3.6.1 Data-Race in `array`

We found a bug in the macro `CK_ARRAY_FOREACH` of `array` that allows the readers to iterate through the array concurrently with a single writer. The `array` data structure is relatively simple, it works with two arrays to which we push changes and can be swapped out in case a commit is needed. There is the `active` array and the `transaction` array. If an element gets removed, the change is stored in the `transaction` array. When storing a new element, this can go to either the `active` or the `transaction` array (if created by a remove operation). The modifying thread owns a private count of elements in the `n_elements` variable that is published on the commit operation to `n_committed`, which makes the changes done by the removing and storing operation visible.

In the code that looks like this:

```c
bool ck_array_commit(ck_array_t *array) {
    // [...] code that swaps arrays
  ck_pr_fence_store();
  ck_pr_store_uint(&array->active->n_committed, array
   ->n_entries);
  return true;
}
```

Listing 3.4: Commit function that makes changes visible to the reader threads

In contrast the macro `CK_ARRAY_FOREACH` reads the `n_committed` with a non-atomic read resulting in a data race with undefined behavior.

```c
#define CK_ARRAY_FOREACH(a, i, b) \
(i)->snapshot = ck_pr_load_ptr(&(a)->active); \
ck_pr_fence_load(); \
for (unsigned int _ck_i = 0; \
    _ck_i < (a)->active->n_committed && \
    ((*b) = (a)->active->values[_ck_i], 1); \
    _ck_i++)
```

Listing 3.5: `CK_ARRAY_FOREACH` macro to iterate through the array data structure

The fix is simple, we just need make the access atomic and need to introduce a load fence, to assure the `n_committed` can be used safely to access the `active` array.

```c
#define CK_ARRAY_FOREACH(a, i, b) \
(i)->snapshot = ck_pr_load_ptr(&(a)->active); \
ck_pr_fence_load();  \
for (unsigned int _ck_i = 0; \
    _ck_i <
    ck_pr_load_uint(&(i)->snapshot->n_committed)) &&\
    (ck_pr_fence_load(), 1) && \
    ((*b) = (a)->active->values[_ck_i], 1); \
    _ck_i++)
```

Listing 3.6: `CK_ARRAY_FOREACH` macro with additional atomic access and snychronization

We also change the pointer within the loop condition to snapshot, to make use of the fact that we loaded it before, although it does not make a difference for correctness. With this fix, we are able to verify the array like all the other data structures.

3.6.2 Bug in hs

For the hs implementation, we found a bug in the ck_hs_get function in combination with the resizing of the map.

```
1  void *ck_hs_get(struct ck_hs *hs, unsigned long h,
2      const void *key) {
3    struct ck_hs_map *map;
4    unsigned int probe;
5    unsigned int *generation;
6
7    do {
8      map = ck_pr_load_ptr(&hs->map);
9      // missing ck_pr_fence_load();
10     generation = &map->generation[h & CK_HS_G_MASK];
11     g = ck_pr_load_uint(generation);
12     probe  = ck_hs_map_bound_get(map, h);
13     ck_pr_fence_load();
14         // [...]
15 }
```

Listing 3.7: ck_hs_get method

In Listing 3.7 line 9, we can see the missing load fence. We need this additional fence, as in the resizing, we store a new map in &hs->map, and to ensure that this read goes through, we need an additional load fence. Unfortunately, due to time limitations, we could not fully verify the hs data structure, so this is still possible for future work.

3.7 Conclusion and Future Work

In this chapter, we presented the real-world appliance of GenMC on the concurrent library Concurrency Kit to verify its data structures. We provided a C11 alternative to the assembly-level concurrency primitives that ck uses to build its data structures. Further, we modified the source code to allow for verification, removing mixed-sized accesses that GenMC cannot handle, introducing a fence for missing data-dependency in C11, and using GenMC hp for efficiency. We wrote three different clients that verify the semantics of the data structures. With GenMC, we can verify the correctness in using three different clients with bag, FIFO, and LIFO semantics. GenMC allowed us to automatically explore complex behaviors difficult to detect through conventional testing, especially for more subtle interactions of weak memory models. With GenMC, we uncovered two bugs in the production code, the data-race in the `CK_ARRAY_FOREACH` macro and the missing synchronization in `ck_hs_get` in case of concurrent resizing. With the simple test framework in place, we are able to extend it in multiple ways.

Comparison to other model checker The clients were written in such a way that they are model checker agnostic. That means we could reuse the same clients in another model checker to evaluate its performance compared to GenMC. We can also use the clients as benchmarks for other concurrent libraries to evaluate their correctness and their performance against the ck library.

Adding more a single modifier, multiple reader client In our work, we have seen that certain data structures, like the hash-set implementation in ck, do not allow for multiple consumers to modify the data structure. But instead, multiple readers read the current state. It would be useful to define and implement an additional client that defines the semantics to be tested.

Chapter 4

Adding Data-Nondeterminism to GenMC

GenMC requires that an execution of a program is deterministic, else revisiting an execution could fail as the interpreter could take a different branch than initially planned. Nevertheless, having non-deterministic data can be useful, for example, if we want to model the randomness of a hash within a data structure. In this chapter, we present how we introduced a new data-nondeterminism type we call `Nondet`. A `Nondet` works similarly to a random integer value, which we can manipulate with arithmetic expressions, comparisons, and other operations. However, in contrast to a random value, it explores all the different paths the program could take because of the randomness.

We show how we implemented `Nondet` in GenMC using concolic expressions and evaluate how performant and usable this new addition is.

4.1 Implementation

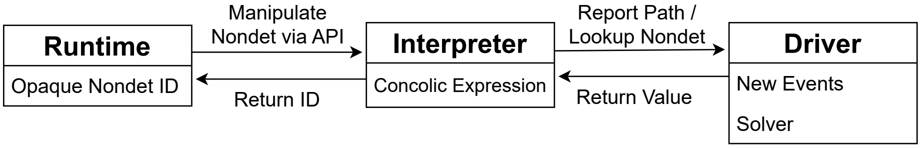

Figure 4.1: Overview of the implementation of data-nondeterministic values in GenMC

To implement non-deterministic values, we touched three components of GenMC as shown in Figure 4.1. In the runtime, we allow C programs to create and manipulate *Nondet* values via an API, which in turn calls the interpreter that handles the manipulation of concolic expressions. For some API calls, the interpreter must call the driver, which looks up the values the interpreter does not know. For this, additional data, such as new events, must be stored in the graph and the solver. In the following subsections, we go into more depth for the single components.

4.1.1 Runtime Interface

For the C/C++ runtime, we add multiple functions that create and manipulate `Nondet` values. The function calls handled in the interpreter return an opaque identifier, which can be combined again to create a new `Nondet`.

To create a new `Nondet`, we can use either `new()` for a seemingly random value or `concrete()` for a fixed concrete value. For combining `Nondets`, we provide different functions that implement simple arithmetic, comparison, and logical operations in unsigned and signed variants, such as addition, less-than, and conjunction. The C programming language does not allow operator overloading, making these explicit calls necessary. If we were to avoid that, an alternative approach to these explicit calls would be to track the `Nondets` through every statement the interpreter executes, intervening every time a `Nondet` is part of an expression. This is how the symbolic execution engine KLEE [12] does it. We decided against this approach for a simpler and modular implementation that needs to modify less existing code. The `Nondet` identifiers used as input arguments in a function call are moved and should not be used in another function call. The interpreter will raise a descriptive error if a program nonetheless reuses a `Nondet`. We also provide the `clone()` function if the reuse is necessary. We choose this approach to avoid unnecessary copies in the interpreter.

If we implement a new runtime in the future, e.g., for a programming language such as Rust, the approach with the opaque identifier is flexible, and we could even use ownership semantics, if available.

To make a decision based on the `Nondet`, we provide the `concretize()` function that returns a boolean for logical or comparison expressions. We have one point where we suffered an abstraction leak, namely in the two functions `send()` and `receive()`. `send()` allows for threads, other than the one that called `new()`, to use the `Nondet`, else this is forbidden. A call to `receive()` then creates a clone for the current thread. We need these functions because of an implementation detail in the interpreter, which we will explain in more detail in the next section.

In Listing 4.1, we see a small example using the `Nondet` for a decision which thread should write to a global variable. We even added a superfluous `add()` on line 27 to show an arithmetic expression in combination with a concrete value.

The full interface is in the Appendix A.

```c
Nondet writer;
int x = 0;

void *thread_1(void *unused)
{
  Nondet local = __VERIFIER_nondet_receive(writer);
  local = __VERIFIER_nondet_eq(local,
   __VERIFIER_nondet_concrete(1));
  if (__VERIFIER_nondet_concretize(local))
    x = 1;

  return NULL;
}

void *thread_2(void *unused)
{
  Nondet local = __VERIFIER_nondet_receive(writer);
  local = __VERIFIER_nondet_ne(local,
   __VERIFIER_nondet_concrete(1));
  if (__VERIFIER_nondet_concretize(local))
    x = 1;

  return NULL;
}

int main(void)
{
  Nondet n = __VERIFIER_nondet_new();
  n = __VERIFIER_nondet_add(n,
   __VERIFIER_nondet_concrete(1));
  writer = __VERIFIER_nondet_send(n);

  pthread_t one, two;
  pthread_create(&one, NULL, thread_1, NULL);
  pthread_create(&two, NULL, thread_2, NULL);
  pthread_join(two, NULL);
  pthread_join(one, NULL);
  assert(x == 1);
}
```

Listing 4.1: Small example using the interface

4.1.2 Interpreter

The interpreter's job is to handle the call from the runtime, maintain the concolic expression state, and look up values in the driver.

Concolic Expressions We store the Nondet as a concolic expression in the interpreter. Concolic is a portmanteau combining concrete and symbolic. The idea is to simultaneously run the code with a concrete value while recording all operations to the symbolic expression in tandem.

```
typedef struct Nondet {
  uint64_t concrete;
  std::unique_ptr<SExpr<NondetId>> symbolic;
} Nondet;
```

Listing 4.2: Internal Nondet type

Our Nondet type in Listing 4.2 is just a simple struct, with a uint64 for the concrete value, and for the symbolic value, we use the SExpr class, which stores the symbolic expression in an operation tree. Suppose we do branching on a Nondet, (the concretize() function in the interface). In that case, we report that to the driver with the current symbolic expression, such that at the end of the execution, the driver can select an alternative path. The concrete value allows us to execute a path until its end without asking the driver multiple times if that path is possible.

Leaking Abstraction As mentioned in the last section, the abstraction leaks into the API with the send() function in the GenMC runtime API. This leak happens due to the combination of two things. First, if we start a new exploration all local state is erased, including the concolic expressions in the interpreter. Second, while exploring an alternative execution, the threads can run before their parent creates them. This is because the execution is deterministic. Thus, the thread already knows with which arguments it will be called. Except, of course, we store our information in the local state. Then, the other thread tries to look up a Nondet identifier that does not exist yet. To preserve the information, we need to store it somewhere durable, which is the execution graph in the driver. But that does not happen automatically, so we need an extra function send(), for which the interpreter calls the driver to store the concolic expression in the graph, and receive() to get it back into the local state.

Local Storage We have seen that the Nondet is just a simple struct. We store all the different structs in a map in the interpreter state. As keys, we use uint64, which we return to the runtime as identifiers. This approach allows us to identify and write useful error messages about the error states in which the Nondet can end up. For example, if the runtime reuses a moved Nondet, it was removed from the map in the previous call. Thus, we can generate a message explaining the use-after-move error.

We use the identifier not only to index into the map but also to store additional information about the state of the Nondet.

- A dirty bit stores if the Nondet was made available to other threads via send() or not.
- The remaining bits store the thread identifier and the position within the thread.

If you, for example, try to use the Nondet identifier in another thread and forget to add a send(), then the interpreter can issue an error message as it sees that the current thread and the thread stored in the identifier differ.

The map allows the runtime to allocate a Nondet without being obliged to free it afterward, as the map gets dropped at the end of the execution. This allows us not to include a free operation in the API, further simplifying the interface.

When the interpreter needs to handle the calls of new(), concretize(), send(), receive(), it cannot do it alone. Instead, it calls the driver, which stores new events to the graph and returns the concrete value, or it records the path in the solver in case of the concretize().

4.1.3 Driver

The driver is responsible for storing everything necessary to restore the state of an execution or to find a new execution. It is also where we add new events to the graph, for which we need to extend the definitions from Section 2.2.1. Also, the solver that finds the concrete values for the different paths is stored in the driver as well.

Solver For concolic expressions, the typical approach is to use an SMT solver to evaluate the different formulas in the path to get the concrete value for the next run. This approach we choose as well. We implemented a new SMTSolver class that provides the interface to interact with an SMT solver. More specifically, we use Z3 [13] for our implementation. Alternatively, we can easily replace the solver for another SMT solver if it supports the same methods as in Listing 4.3

```
template <typename T> class SMTSolver {
private:
  z3::context ctx;
  z3::solver solver;

public:
  using Term = z3::expr;

  Term make_symbol(const T &id)
  Term make_constant(uint64_t value)
  Term make_term(SExpr<T>::Kind op, const Term &t1,
   const Term &t2)
  Term make_term(SExpr<T>::Kind op, const Term &term)

  void push(const SExpr<T> *expr)
  void pop(unsigned n = 1)
  std::unique_ptr<SMTSolver<T>> clone()
  std::optional<std::unique_ptr<SMTModel<T>>> solve()
};
```

Listing 4.3: SMTSolver class

The make_ methods provide the translation from the GenMC SExpr class to the expression type of Z3. For adding formulas to the solver from the concretize() calls, we use the push method. We use the pop method to remove formulas if, in case of a forward revisit, we restrict the execution graph to a subset. We use push and pop for incremental solving that reuses computation in case of a forward revisit. In a backward revisit, the execution is not re-winded to a prior state, but instead, a new state is built from the ground up. Thus, we need to do the same with the formulas in the solver. Lastly, the most crucial method solve() returns a model for the conjunction of formulas if they are satisfiable, or alternatively, nothing if they are unsatisfiable.

Handling of Interpreter Calls The driver handles the four interpreter calls as follows:

- new(): For this the driver creates a new Nondet. The concrete value comes from the interpreter, but by default, we choose 0. The symbolic expression contains just a new symbolic variable. We store the Nondet in the NondetNew event in the graph and return a clone to the interpreter as well.

- send(): Handled in the same way as new(), but the difference is that the interpreter provides both the concrete and the symbolic expression. Both are stored again in a NondetNew event and returned as a clone to the interpreter.

- receive(): Here comes into play that, within the key to the interpreter Nondet map, we store the position of when we used send(). The interpreter passes this information to the driver, which can use it to directly access the correct event in the execution graph without any search. The looked-up event is a `Nondet`, which we can clone and return to the interpreter.

- `concretize()`: The interpreter can already compute the concretize's concrete value on its own. Thus, the driver records the formula in a `Concretize` event and additionally adds it to the `SMTSolver` with a `push`.

Events

For handling the `Nondet`, we extend the events in GenMC with two new ones, `NondetNew` and `Concretize`. We explain what the rationale behind these two events is.

`NondetNew` is an event that marks the position where a new `Nondet` is created. It stores the concrete value a `Nondet` starts with and the `SExpr`. The `SExpr` is inserted in the model of the solver to get a new concrete value for the next execution.

`Concretize` is an event to store the path taken through the program. It stores the formula as `SExpr` and additionally a boolean `maximal` field that returns if the event is added maximally to the graph. If we re-add the event in a revisit, we invert the maximal field.

The only formal definitions of Section 2.2.1 that we need to extend are the definitions of events and co-maximality.

Def 4.1 An event $e \in \text{Event}$ is either the initialization event `init`, or a thread event $\langle t, i, lab \rangle$, with $t \in Tid$ being the thread identifier, $i \in Idx$ the index within the graph, and $lab \in \text{Lab}$ one of the following labels:

- *Write label*: $W(l, v)$ where $l \in \text{Loc}$ is the location accessed, and $v \in Val$ the value written.
- *Read label*: $R(l, v)$ where $l \in \text{Loc}$ is the location accessed.
- *NondetNew label*: $N(s, v)$ where $s \in \text{S}$ is the symbolic expression, and $v \in Val$ is the value written.
- *Concretize label*: $C(s, b)$ where $s \in \text{S}$ is the symbolic expression and $b \in \text{bool}$ is the boolean indicating if the event is added maximally.
- *Error label*: denoting safety violation

The co-maximality definition for the new events is very simple.

Def 4.2 co-maximality for an event is defined as:

- A write event $w \in W$ is co-maximal if there does not exists a $w' \in W$ such that $\langle w, w' \rangle \in \text{co}$
- A read event $r \in R$ is co-maximal if $\text{rf}(r)$ is co-maximal
- A NondetNew event $n \in N$ is always co-maximal
- A concretize event $c(\_, b) \in C$ is co-maximal if $b = \text{true}$.

Proving the property of the TruSt algorithm, that there is still exactly one execution that is maximal, is trivial. The NondetNew itself cannot trigger revisits at all, so there exists exactly one way to add it. For concretize, we explore if the formula is either true or false. One of them, we mark as maximally with the flag, thus ensuring there is exactly one way to add the concretize event.

Let us now explain the implementation of the `Nondet` through a running example in Listing 4.4. For clarity, we shortened the function names that manipulate `Nondet`.

```
void main() {
    Nondet n = new();
    Nondet m = new();
    concretize(eq(n, concrete(2)));
    concretize(eq(m, concrete(2)));
}
```

Listing 4.4: Small example with a new and concretize function calls

In the program, we compare two `Nondet`s to 2. As the concrete value of the `Nondet` is either 2 or not, we expect a total of four executions.

In Figure 4.2, we can see the first execution of the program in Listing 4.4. The `new()` calls are represented with the `NewNondet` event in the graph, both with the concrete value of 0 (the default) and the symbolic variable (n and m) as symbolic expression.

The interpreter stores the computations on the `Nondet` in its own state, which we can see on the right side of the figure. We explained in Section 4.1.2 how the key is composed of different parts. For example, (0, 1, 3) means that the `Nondet` is local to thread 0 and was manipulated last by the third statement in the thread. It may appear as the `Concretize` label stores the wrong formula, $n \neq 2$, instead of the $n = 2$ we defined in the program. But this is correct, as we want to record the path the execution takes, which is with n set to 0 $n \neq 2$. Additionally, observe that the `maximal` field of `Concretize` is `true`.

The driver could now explore two alternative executions in setting $n = 2$ or $m = 2$. We choose the former. The exploration of alternative `Concretize` events is always a forward revisit, as it is never influenced directly by another event added after it.

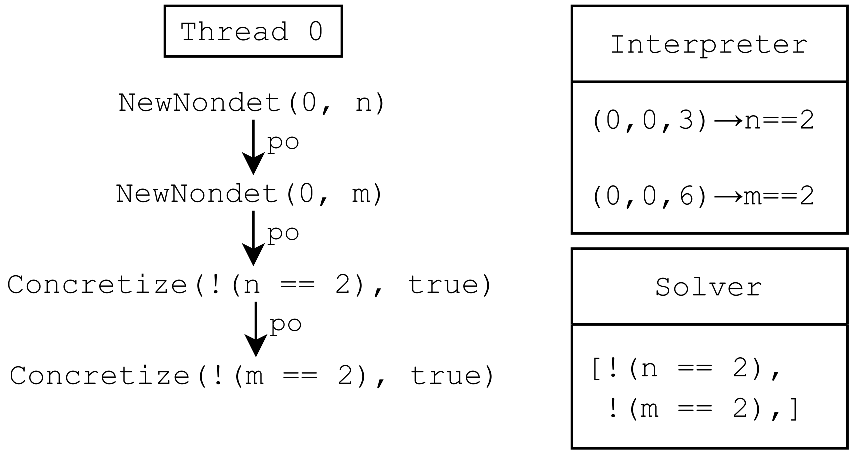

Figure 4.2: Execution graph and state at the end of program of Listing 4.4

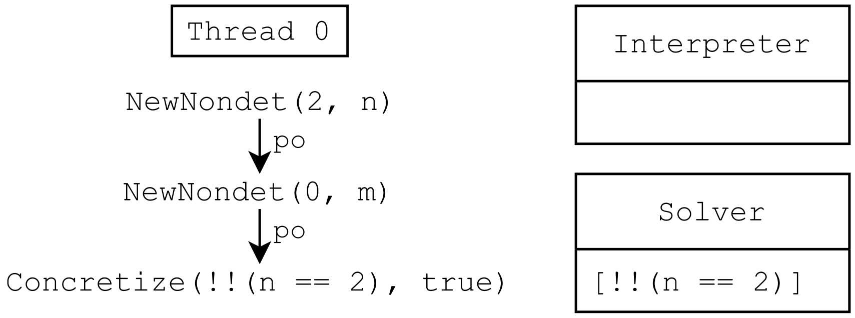

Figure 4.3: Execution graph and state right before the start of second execution

See Figure 4.3 for the state before running the second exploration. To revisit the first concretize, we restrict the graph right before we inserted it, which includes popping both formulas off the solver. Then, we re-add the `Concretize` event with an inverted formula (as we now explore the alternative path) and add the inverted formula to the solver as well. Now, we use the solver to solve for the formulas stored to get a model for it. If the conjunction of the formulas is unsatisfiable, we would stop here. If they are satisfiable, we re-evaluate the symbolic expressions in the `NewNondet` events with it and store the concrete value back into the label. The model chooses for the free variables $n = 2$ and $m = 0$.

The interpreter's state is erased, and the driver is ready to start the next execution with part of the graph already in place.

4.2 Evaluation

In the following section, we evaluate the performance of our implementation of Nondet in different data structures against their equivalent using GenMC and modeling the Nondet with explicit integers instead. We want to show the overhead of Nondet is minimal and that the usage of Nondet can simplify and improve the performance in certain situations.

Experimental Setup We performed all experiments on a Lenovo ThinkPad X13 Gen 4 with a AMD Ryzen 7 Pro 7840U (8 cores @ 3.3) GHz with 32 GiB of RAM. We used LLVM 18.1.3 and built our implementation on top of GenMC v0.10.2. We measured the time using hyperfine running each test at least 10 times. We measured the peak memory usage with valgrind massif. All tests run on a single thread with a timeout limit of 1 hour.

4.2.1 MS-Queue and Treiber-Stack

We evaluate the overhead of Nondets as payload in two data-independent data structures. We do this by providing two implementations, one with Nondet as payload and the other int as payload. The stack and queue code are based on GenMC test implementations. We test for two types of workloads, first workloads with only writers and then workloads with the same number of writers and readers. Both the Nondet and the int version time out at the same limits. First, Let us observe the overview in Figure 4.4. Both the Nondet and integer variant experience an exponential growth

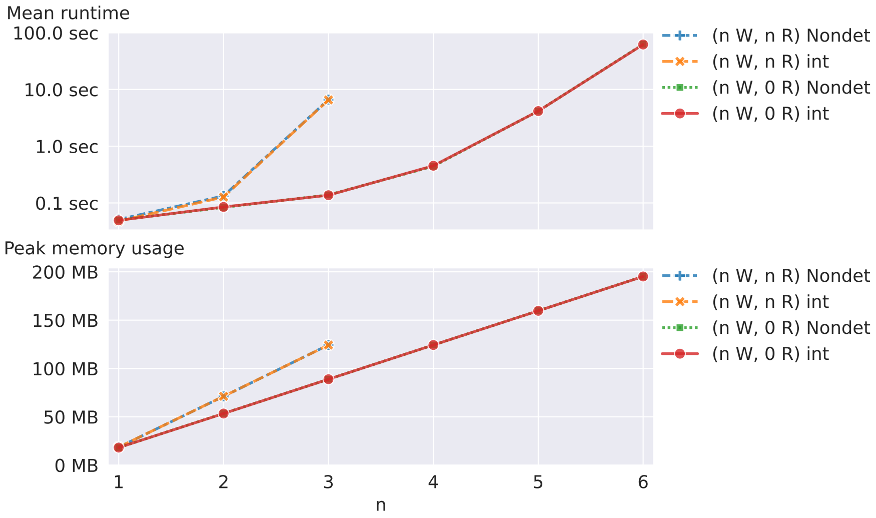

Figure 4.4: Runtime and peak memory usage for MS-queue over increasing parallel accesses

of the mean runtime with increasing numbers of accesses. We cannot see any overhead in runtime for the Nondet compared to the integer variant. On the bottom part of the figure, we see the peak memory usage. With more accesses the memory consumption increases linearly. The peak memory usage for both variants is basically the same for both, the biggest difference is the sub-percentage range. It is likely that another system in GenMC is causing the peak memory usage. The number of executions is not shown in a graph, but it is, as expected, exactly the same for both variants.

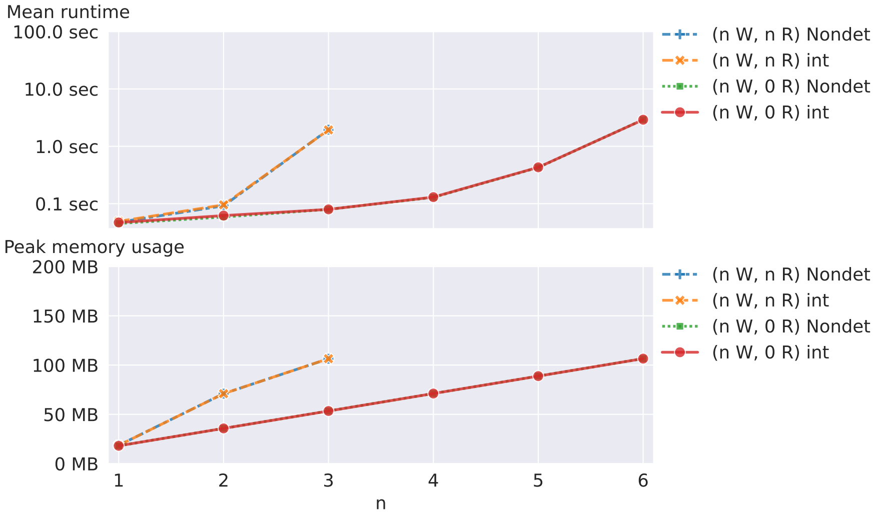

Figure 4.5: Runtime and peak memory usage for Treiber-stack over increasing parallel accesses

Similar results we can observe for the Treiber-stack in Figure 4.5. Also, there we cannot observe a difference either in memory or runtime. We can show with this that creating and passing Nondet around does not cause any significant overhead.

4.2.2 List-based sets with fine-grained and coarse-grained locking

In the next experiment, we implemented sets based on a list with either coarse-grained locking over the whole data structure or fine-grained *hand-over-hand* locking. The implementations are based on [14]. As the keys we used Nondet in the Nondet variant doing all the comparisons with the methods we introduced in Section 4.1. The data structure disallows the keys to be 0 or the maximum value of the integer type, the two sentinel values at the beginning and end reserve these. That simplifies the iteration through the list. To ensure the Nondet are in the correct range, we use assume() that blocks the execution if the value of the Nondet is outside the correct range, see Listing 4.5, adding the assume() introduces blocked executions.

```c
void assume_keyrange(Nondet key) {
  Nondet lo = ult(concrete(0), clone(key));
  Nondet hi = ult(clone(key), concrete(UINT64_MAX));
  __VERIFIER_assume(concretize(conjunction(lo, hi)));
}
```

Listing 4.5: Code for assuming `Nondet` key range (with shortened function names)

While the `Nondet` variant automatically checks all possible orderings of entries in the set, we need to do more work implementing the integer variant. We fixed the number of readers and writers to two and assigned fixed values that the reader/writer should read/write. Then we start the executions with all possible combinations in a four-dimensional Cartesian product space of $\{(t_1, t_2, t_3, t_4) \mid t_{i \in [1,4]} \in [1,4]\}$. We select the values in the range $[1,4]$ to ensure we cover all possible ordering. An alternative approach with a four level for-loop assigning the values directly without restarting the execution failed, as GenMC did not manage to unroll these nested loops.

Locking	Datatype	W	R	Executions	Blocked	Memory	Time
Fine	Integer	2	2	6144	0	71 MB	26.17 s
Fine	Nondet	2	2	464	165	76 MB	0.66 s
Coarse	Integer	2	2	6144	0	71 MB	25.85 s
Coarse	Nondet	2	2	464	165	76 MB	0.64 s

Table 4.1: Performance for the list-based set

When checking the results in Table 4.1 the first thing that one notices is that there is no difference between the coarse-locking and the fine-grained locking, at least from the perspective of verification. Both have the exact same number of completed and blocked executions, peak memory usage, and runtime.

We see that the integer variant explores a magnitude order more executions than the `Nondet` variant. This might come as a surprise as both variants explore all possible enumerations, but where the integer variant explores the same symmetric enumeration multiple times, e.g., (1, 1, 1, 1), (2, 2, 2, 2), (3, 3, 3, 3), (4, 4, 4, 4), the `Nondet` variant only explores one of those such that $t_1 = t_2 = t_3 = t_4$, this leads to the `Nondet` potentially exploring exponentially less executions. Indeed, we could change the integer variant to only check for one element in a symmetric enumeration, but that would also complicate the implementation for something we get for free in the `Nondet` variant. The additional peak memory usage we can observe for the `Nondet` could arise from different sources, as we have to additionally store all the events introduced by the ordering and update the state in the interpreter and the driver as well. The execution time of the integer variant is higher but this is probably due to the overhead starting GenMC 256 times. We conclude that using Nondet not only simplified the list-based set implementations but also improved the performance in comparison to the naive enumeration.

4.2.3 Time-stamped Stack

Time-stamped stack [15] is a relatively new stack implementation that avoids costly synchronization by using a single-producer multiple-consumer queue for each thread but still ensures linearizability via time-stamps. We test two different implementations for the time-stamps, both with an integer and a Nondet variant. First, the time-stamped is based on an atomic counter, which is incremented every time we add a new element. Second, a *stuttering* time-stamp, for which each thread has its own counter, and from all threads we take the maximum if we add a new element. This can lead to duplicate time-stamps. We performed the test with two writer threads that write 2 times (to ensure multiple writes to the thread-local queues) and a reader thread that reads once.

Time-Stamp	Datatype	Executions	Blocked	Memory	Runtime
Atomic Counter	Integer	1418	24	125 MB	1.06 s
Atomic Counter	Nondet	2393	19165	132 MB	17.63 s
Stuttering	Integer	10171	76	76 MB	4.69 s
Stuttering	Nondet	1476	4829	125 MB	4.33 s

Table 4.2: Performance measurements for time-stamped stack

In Table 4.2, we see a limitation of Nondet. For the atomic integer counter, we only need to enumerate all the cases in which order the access to the counter happen. For the atomic Nondet counter, we additionally need to use assume() to ensure the new time-stamp is bigger than the previous entry, enforcing a total order. As it is currently implemented, the driver explores the blocked executions as well. This is further a disadvantage, as if we don't block on this concrete value, we do an extra call to a solver for some execution we are going to throw away anyway. The effect is smaller for the stuttering time-stamp, here the Nondet only compares to the local time-stamp, reducing the number of executions drastically. The integer implementation needs to read all thread counters and then add one, leading to many unnecessary interleavings.

4.3 Conclusion and Future Work

In this chapter, we presented the implementation of data-nondeterminism in GenMC through the Nondet type. We integrated concolic expression in combination with the Z3 SMT solver into GenMC, allowing programs to explore all possible paths that could result from non-deterministic data. The

Nondet implementation is modular, and parts can swapped out without affecting other parts in the code. The runtime API is simple, and the Nondet appears as a value that can be easily passed around. Through the concolic approach a large part of the code is evaluated by the interpreter without interrupting calls to the driver, as the concrete value of most operations is already known.

Our evaluation demonstrated that the Nondet implementation incurs minimal overhead when used as the payload in data structures like MS-Queue and Treiber-Stack, with virtually identical memory usage and runtime performance compared to integer-based payloads. More importantly, for data structures like list-based sets, Nondet significantly reduced verification complexity by automatically exploring unique orderings instead of redundant symmetric enumerations, reducing execution count by an order of magnitude and improving runtime performance. We did identify limitations in certain scenarios, particularly with the time-stamped stack using an atomic counter, where Nondet explored many blocked executions due to constraints on timestamp order. However, there are several areas where this implementation could be extended.

Avoiding Abstraction Leak In Section 4.1.2, we explained how we have leaked implementation details into the API. We can avoid this complexity if we change the order in which GenMC is issuing the statements interpreted by the interpreter. If whatever order that is newly enforced ensures that we don't execute a thread before the `pthread_create()`, the usage of the two methods `send()` and `receive()` becomes unnecessary.

Testing out Different Implementations With the Nondet implementation being quite modular, there are now multiple possibilities to swap out components or to change how certain components are stored and to evaluate their advantages and disadvantages. The simplest thing would be to change the Nondet type from move semantics to copy semantics. The required change for this is straightforward, as we just have to clone the Nondet within the interpreter instead of moving it. Another thing would be to check the performance of an alternative SMT solver to evaluate whether that solver is more performant for the usage pattern that we have with GenMC. And last, the position where we store the SMT solver itself could be evaluated. Currently it resides in the execution beside the execution graph, but an alternative approach would be to move it into the driver itself. That could introduce more maintenance as the driver and the execution graph could be out of sync, but we would avoid copying the solver for every execution.

Better Exploration of Possible Executions This improvement is connected to the problems we mentioned in Section 4.2.3, where we explored a lot of blocked executions. One way to approach this is by better heuristics, e.g., we could provide a function that combines `concretize()` and `assume()`, such that if the current execution with the concrete value is not blocked, we add the formula as assumption and not as revisit point. Alternatively, we could expose the data structures of the SMT solvers directly to the API. For example, if we want to check collision within an array, we currently have to check for every index. With the SMT solver array, it would be possible to check directly for collisions, avoiding many blocked executions.

Bibliography

[1] Lamport, Leslie, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, Sep. 1979, Conference Name: IEEE Transactions on Computers, ISSN: 1557-9956. DOI: 10.1109/TC.1979.1675439. [Online]. Available: https://ieeexplore.ieee.org/document/1675439 (visited on 10/13/2024).

[2] M. Kokologiannakis and V. Vafeiadis, "GenMC: A model checker for weak memory models," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds., vol. 12759, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2021, pp. 427–440, ISBN: 978-3-030-81684-1 978-3-030-81685-8. DOI: 10.1007/978-3-030-81685-8_20. [Online]. Available: https://link.springer.com/10.1007/978-3-030-81685-8_20 (visited on 04/21/2025).

[3] M. Kokologiannakis, I. Marmanis, V. Gladstein, and V. Vafeiadis, "Truly stateless, optimal dynamic partial order reduction," *Replication Package for "Truly Stateless, Optimal Dynamic Partial Order Reduction"*, vol. 6, 49:1–49:28, POPL Jan. 12, 2022. DOI: 10.1145/3498711. [Online]. Available: https://dl.acm.org/doi/10.1145/3498711 (visited on 10/13/2024).

[4] W. Pugh, "The java memory model is fatally flawed," *Concurrency: Practice and Experience*, vol. 12, no. 6, pp. 445–455, May 2000, ISSN: 1040-3108, 1096-9128. DOI: 10.1002/1096-9128(200005)12:6<445::AID-CPE484>3.0.CO;2-A. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/1096-9128(200005)12:6%3C445::AID-CPE484%3E3.0.CO;2-A (visited on 10/13/2024).

[5] J. Manson, W. Pugh, and S. V. Adve, "The java memory model," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '05, New York, NY, USA: Association for Computing Machinery, Jan. 12, 2005, pp. 378–391, ISBN: 978-1-58113-

830-6. DOI: 10.1145/1040305.1040336. [Online]. Available: https://doi.org/10.1145/1040305.1040336 (visited on 10/13/2024).

[6] D. Aspinall and J. Ševčík, "Java memory model examples: Good, bad and ugly," in *Proceedings of Verification and Analysis of Multi-Threaded Java-Like Programs (VAMP 2007)*, 2007. [Online]. Available: https://www.research.ed.ac.uk/en/publications/java-memory-model-examples-good-bad-and-ugly (visited on 10/13/2024).

[7] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, "Repairing sequential consistency in c/c++11," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, New York, NY, USA: Association for Computing Machinery, Jun. 14, 2017, pp. 618–632, ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062352. [Online]. Available: https://dl.acm.org/doi/10.1145/3062341.3062352 (visited on 10/13/2024).

[8] "Std::memory_order - cppreference.com." (), [Online]. Available: https://en.cppreference.com/w/cpp/atomic/memory_order (visited on 04/14/2025).

[9] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, Jun. 2004, ISSN: 1045-9219. DOI: 10.1109/TPDS.2004.8. [Online]. Available: http://ieeexplore.ieee.org/document/1291819/ (visited on 04/16/2025).

[10] K. Fraser, "Practical lock-freedom," University of Cambridge, Computer Laboratory, UCAM-CL-TR-579, 2004. DOI: 10.48456/tr-579. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.html (visited on 04/18/2025).

[11] M. M. Michael and M. 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*, ser. PODC '96, New York, NY, USA: Association for Computing Machinery, May 1, 1996, pp. 267–275, ISBN: 978-0-89791-800-8. DOI: 10.1145/248052.248106. [Online]. Available: https://dl.acm.org/doi/10.1145/248052.248106 (visited on 04/18/2025).

[12] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs,"

[13] *Z3prover/z3*, original-date: 2015-03-26T18:16:07Z, Apr. 22, 2025. [Online]. Available: https://github.com/Z3Prover/z3 (visited on 04/22/2025).

[14] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., May 2012, 536 pp., ISBN: 978-0-12-397337-5.

[15] M. Dodds, A. Haas, and C. M. Kirsch, “A scalable, correct time-stamped stack,” in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Mumbai India: ACM, Jan. 14, 2015, pp. 233–246, ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2676963. [Online]. Available: https://dl.acm.org/doi/10.1145/2676726.2676963 (visited on 04/26/2025).

Appendix A

Runtime Interface for `Nondet`

```c
/*
 * @brief Create a new symbolic variable.
 *
 * Manipulate symbolic expression (Nondet) with genmc functions starting with ‘__VERIFIER_nondet_*‘.
 * Important! If you use a Nondet in a function call, it is moved and cannot be reused.
 * An error will be issued if you do it nonetheless. Execption is ‘__VERIFIER_nondet_clone‘. There
 * you can reuse the passed Nondet afterwards again.
 *
 * @return symbolic expression of a new variable.
 */
Nondet __VERIFIER_nondet_new(void);
Nondet __VERIFIER_nondet_concrete(uint64_t value);
Nondet __VERIFIER_nondet_clone(Nondet value);
bool __VERIFIER_nondet_concretize(Nondet value);

/* Make Nondet shareable with another thread.
 * Once shared you can use the Nondet in as many threads you want,
 * as the cloning happens with ‘__VERIFIER_nondet_receive‘. */
Nondet __VERIFIER_nondet_send(Nondet value);
/* Make shared Nondet usable by current thread */
Nondet __VERIFIER_nondet_receive(Nondet value);

/* logical nondet expressions */

```

```
Nondet __VERIFIER_nondet_conjunction(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_disjunction(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_not(Nondet value);

/* arithmetic nondet expressions */

Nondet __VERIFIER_nondet_add(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_sub(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_mul(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_udiv(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_sdiv(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_urem(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_srem(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_and(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_or(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_xor(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_shl(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_lshr(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_ashr(Nondet lhs, Nondet rhs);

/* comparison nondet expression */

Nondet __VERIFIER_nondet_eq(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_ne(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_ult(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_ule(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_ugt(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_uge(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_slt(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_sle(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_sgt(Nondet lhs, Nondet rhs);
Nondet __VERIFIER_nondet_sge(Nondet lhs, Nondet rhs);
```

Listing A.1: Nondet Interface

ETH

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 three options must be selected:

● I confirm that I authored the work in question independently and in my own words, 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 confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Enabling Automatic Verification of Concurrent Data Structures

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

Last name(s):	**First name(s):**
Leutenegger	Christof

With my signature I confirm the following:

- I have adhered to the rules set out in the Citation Guide.
- 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, 27.04.2025	

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] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard