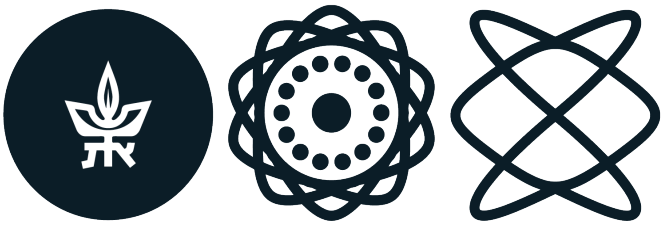

The Raymond and Beverly Sackler Faculty of Exact Sciences
Tel Aviv University

~~Verifying~~ Testing and Optimizng Data Structure Implelentations Under the RC11 Memory Model

Thesis submitted in partial fulfillment

of graduate requirements for the degree

Master of Science - M.Sc

in the Blavatnik school of Computer Science, Tel Aviv University

By

Ori Saporta

This work has been carried out under the supervision of

Dr. Ori Lahav

Abstract

abstract goes here

Acknowledgments

ack

Contents

1 Introduction

Software systems are typically structured with multiple layers of abstractions, where each level of abstractions is constructed using lower-level ones. This approach also applies to concurrent programs for which the lowest level of abstraction is made up of the basic atomic operations such as read, write and compare-and-swap (CAS). These operations are used to build synchronization primitives (e.g. locks) which, in turn, are used to build concurrent data structures (e.g. queues, stacks, etc.). These data structures are then used to implement higher-level algorithms (e.g. concurrent graph traversal), which are then used as components in larger concurrent programs.

This hierarchical model also applies to the verified process. Each component of the system is specified separately, and each implementation is verified against its specification. This approach has been studied extensively in the context of interleaving concurency, a.k.a. *sequential consistency* (SC) [14]. In this context, existing work includes correctness criteria for data structures (linearizability [10] as well as automated tools for checking or proving this criteria [23].

However, in recent years, weak memory consistency (WMC) has become the de facto standard for shared-memory concurrency. The semantics of atomic operations is governed by weak memory models at both the hardware architecture level [2] [20] and the programming language level [1] [3] [16]. These semantics allow behaviours (e.g. "store buffering") which are not allowed under SC.

Concurrent data structures are implementations of abstract data structures (ADTs) which provide some guarantees when used in a concurrent environment. These guarantees include thread safety (i.e. data race freedom) as well as some functional guarantee on the expected behavior with regards to the specifications of the ADT. The introduction of the C11 memory model [3] and its repaired variation [13] has allowed researches and developers to design and implement new versions of concurrent ADTs with relaxed atomic accesses in order to improve performance at the cost of consistency. If these implementations are not properly verified, it may lead to unwanted behaviors.

For instance, in 2019, Raad, Doko, Rožić, Lahav and Vafeiadis [21] showed that the following trace[1] is a possible execution of the implementation in listing 1 (based on the Herlihy-Wing

[1]Throughout this work we use this notation to describe method calls on an ADT implementation. For queues, the method called is denoted by Enq/Deq, the arguments are given in parenthesis and the return value after the / where applicable.

```
1  typedef struct queue_t {
2      val_t * _Atomic AR[MAX];
3      atomic_int back;
4  } queue_t;
5
6  void Enq(queue_t *q, val_t * val) {
7      //should be memory_order_acq_rel
8      int k = atomic_fetch_add_explicit(&(q->back), 1, memory_order_release);
9      atomic_store_explicit(&(q->AR[k]), val, memory_order_relaxed);
10 }
11
12 bool Deq(queue_t *q, val_t **retVal) {
13     int lback = atomic_load_explicit(&(q->back), memory_order_acquire);
14     val_t * lan = NULL;
15     if (lback > 0) {
16         for (int k = 0; lan == NULL && k < lback; ++k) {
17             lan = atomic_exchange_explicit(&(q->AR[k]), 0, memory_order_relaxed);
18         }
19     }
20     *retVal = lan;
21     return lan != NULL;
22 }
```

Listing 1: The (weak) Herlihy-Wing queue implementation under WMC

Queue [10]):

$$a : Enq\,(v_1) \quad b' : Deq/v_2 \quad c' : deq/v_3 \quad d : Enq\,(v_4)$$
$$b : Enq\,(v_2) \quad c : Enq\,(v_3) \quad d' : deq/v_4 \quad a' : Deq/v_1$$

This trace is inconsistent with the expected sequential behavior of a queue. Specifically, with the expectations that each value is enqueued before it can be dequeued, and that program order between calls in the same thread is maintained. Given these constraints, there is no way to create a total order from these calls without encountering a cycle. In the aforementioned article it is suggested that altering the memory order in the FAA operation in line 8 to *memory_ order_ acq_ rel* will solve this issue and the above trace will no longer be achievable.

In this work, we propose a method to verify and optimize concurrent ADT implementations under the RC11 memory model. To that end, in section 3, we define correctness criteria, designed specifically to formalize the aforementioned expectations under the RC11 memory model.

In section 4 we describe the method to test and optimize ADT implementations. We do that by first formally defining what constitutes incorrect ADT behaviors as a logical QBF formula. We then show how to generate test sets, composed of test cases, each of which is an assignment to the logical formula. From these test cases we generate a collection of test programs, referred to as a test suite, which can be analyzed by a model checker in order to verify the correctness of

an implementations. We then suggest how to use these test suites to test, fix and optimize ADT implementation w.r.t. atomic accesses under the RC11 memory model.

In section 5 we provide concrete applications of our method to the queue and set ADTs and use it to obtain relevant test suites. Then, in section 6, we proceed to test and optimize selected implementations using the generated suites. We show how we identify and fix issues in existing implementations as well as facilitate the creation of optimized versions of these implementations with potentially improved performance. In section 7 we discuss the limitations we encountered during this process of applying our method to queues and sets, in the context of the specific tools we used to do it.

2 Background

In this section we describe two elements we rely heavilty on going forward, the RC11 memory model through examples and Model checking through the GenMC tool.

2.1 RC11 Through Examples

and its repaired version (RC11) [13] (which is more or less: $C11 + acyclic\,(PO \bigcup RF)$) *treadeof* Sample with flag and busy waiting. Message passing. store buffer sample - weaker than SC.

2.2 Model Checking

A model checker generates all possible executions for a specific test program thus allowing us to determine whether a specific history H is one of them. We suggest using a state of the art model checker, designed for the RC11 memory model, to analyze specific test programs. We can then verify whether the model checker encountered any unwanted histories during this analysis.

3 Problem definition

In this section we define the problem we are addressing in this thesis. First, we generally describe what we are trying to achieve when examining concurrent implementation of ADTs. Second, we present the specific criteria for correctness that we employ.

```
1  // enq
2  void *thread_1(void *arg)
3  {
4      int * val = (int *)malloc(sizeof(int));
5      *val = 1;
6      enqueue(&q, val);
7      return NULL;
8  }
9
10 // deq
11 void *thread_2(void *arg)
12 {
13     int * res = NULL;
14     bool succ = dequeue(&q, &res);
15     assert(succ);
16     assert(*res == 1);
17     return NULL;
18 }
```

Listing 2: Simple sample of Enqueue and Dequeue calls on a standard Queue implementation

3.1 Goal

The goal of this work is to create a method to test and optimize concurrent ADT implementations for the RC11 standard based on their specifications. This goal is twofold:

1. To have the ability to test whether specific ADT implementations can be used safely, concurrently, under the RC11 weak memory model.

2. To suggest variations on these implementations, with atomic accesses optimized to be as relaxed as possible, in a way that improves performance, on the modern CPU, while remaining correct under our criteria.

To do that we first define the criteria by which we can determine whether an implementation is *correct*.

3.2 Correctness Criteria

No Data Races Calling a method on an ADT, as specified in its interface, should never produce a data race (involving non-atomic accesses). Moreover, accessing the data previously stored in the implemented data structure, should also be race free. This requirement is further motivated by the fact the data races are considered undefined behavior (UB) under RC11. For example, if a queue Q holds a reference to object O and thread T calls the dequeue operation and receives a reference to O, no other caller can receive a reference to O at any point (before or after

T). For a concrete example, consider the sample in Listing 2, thread_2 successfully dequeues a value (lines 14-15), it then accesses it by dereferencing the pointer (line 16). We require that in a correct implementation, these operations should all be safe from contention with other threads.

Consistency w.r.t. ADT Sequential Specifications The second criteria we require for an implementation is validity under the ADT's specifications, which can be considered functional correctness. Our criteria is derived from the correctness of concurrent executions under the sequential specification as previously studied in [21].

For this approach we assume a set of operations, denoted by Ops. Each operation represents a method call on a specific ADT and consists of a label, describing the method called, as well as the arguments and return values of the call. Furthermore, we assume the sequential specifications of the ADT formulated as the set of all total orders over the set of operations Ops that intuitively represent the set of all sequences of operations allowed by the ADT. For Example, one possible set of operations for the queue ADT is $Ops = \{Enq\,(1)\,, Deq/1\}$. For this set the sequential specification is $\{\{\langle Enq\,(1)\,, Deq/1\rangle\}\}$ since the value 1 has to be present in the queue before it is removed. To define consistency w.r.t. a sequential specification we use the following definitions:

Definition 1. Executions

- An *execution* is a partial order over the set of operations Ops.
- An execution e is called *consistent w.r.t. the sequential specifications of the ADT* if there exists a total order to over the set of operations in the ADT's sequential specification such that $e \subseteq to$. Below, in the context of a given sequential specification, we will refer to executions consistent w.r.t. a sequential specifications of the ADT as *consistent*, and those not consistent w.r.t. the sequential specifications as *failed.*

Continuing the previous example, for the given set of operations and sequential specification,{} (the empty partial order) is a consistent execution, since it is contained in the total order $\{\langle Enq(1), Deq/1\rangle\}$, while $\{\langle Deq/1, Enq(1)\rangle\}$ is a failed execution. Note that by definition, if fe is a failed execution, any extension of it fe' such that $fe \subset fe'$, will be a failed execution as well.

We require that a correct ADT implementation will only produce consistent executions. Since this correctness criteria is exhaustive, in the sense that proving it for a specific implementation

requires validating all possible executions, it can become very difficult to prove correctness directly. However, if an implementation is flawed (such as the HW Queue implementation in Listing 1), finding *failed executions* to prove that should be a more reasonable task. This will be our approach going forward.

4 Method

In this section we will describe the method we suggest for testing and optimizing concurrent ADT implementations. The method is comprised of several steps. First, we show how to use an SMT solver to generate a test set, comprised of all the minimal failed executions over a set of operations on any specific ADT. Then, we demonstrate how to convert a test set into a suite of test programs that can be analyzed by the model checker. These steps have to be done only once per ADT, and can subsequently be used to test any implementation. Lastly, we suggest a way to use these test programs to both identify issues and produce optimized[2] implementations, while remaining correct under our criteria.

4.1 Generating Test Sets

Next we will show how to use a quantified boolean formula (QBF) to identify test cases that make up the test set.

4.1.1 Defining Test Cases for ADT Consistency

The test sets are defined as follows:

Definition 2. A failed execution f is called *minimal* if every $r \subset f$ is a consistent execution.

Since we assume that the underlying memory model is "monotone" (as RC11 is), it is sufficient to test implementations only with minimal executions since any behavior observable on a non-minimal execution, will be observable on the minimal subset as well.

Definition 3. A *Test Case* is a minimal failed execution. A *Test Set* consists of all the test cases that can be found over a set of operations.

[2]From this point forward we use the term optimized to indicate that an implementation's atomic accesses are as relaxed as possible

We use Graph Isomorphism (*GI)* [22] to indicate that two failed execution are semantically equivalent w.r.t. the ADT's specifications. Intuitively, two graphs *G, G'* will be considered isomorphic if there is a way to "shuffle" the edges of one to recreate the other in a way that is indistinguishable to the ADT's implementations. For example, the following two queue executions are considered isomorphic:
$\{\langle deq(1), enq(2)\rangle, \langle enq(1), deq(2)\rangle\}$, $\{\langle deq(2), enq(1)\rangle, \langle enq(2), deq(1)\rangle\}$[3].
To keep the test sets minimal, in cases where the ADT supports a definition of isomorphism, the test sets will be further reduced by filtering isomorphic test cases.

4.1.2 Using SMT Solvers to Generate Test Sets

We can now show how to generate test sets for any specific ADT given a set of operations. We reduce the problem of finding a test set over a given set of operations to an extension of the problem of finding all assignments that satisfy a given quantified boolean formula (QBF), adding the filtering of isomorphic solutions, where applicable. Each assignment we identify is a test case. Combined, they are the test set. The QBF formula is encoded as follows. Let $Ops = \{o_i\}_1^N$ be our set of operations and e an execution. We define the set of boolean variables X_{ij} for every $i, j \in \{1...N\}$with the intuition that: $X_{ij} = true \iff \langle o_i, o_j\rangle \in e$. We define the following traits:

1. *po* (partial order) - transitive and anti-reflexive:
   $po(X) = \left(\bigwedge_{i\in\{1...N\}} \neg X_{ii}\right) \wedge \left(\bigwedge_{i,j,k\in\{1...N\}} (X_{ij} \wedge X_{jk} \implies X_{ik})\right)$

2. *to* (total order) - an augmentation of the partial order to include all *Ops*:
   $to(X) = po(X) \wedge \left(\bigwedge_{i,j\in\{1...N\}, i\neq j} (X_{ij} \vee X_{ji})\right)$

3. *spec* - formal description of the correct behavior of the specific ADT we wish to test. For now, we employ the black-box $spec(X)$ to determine if X is included in the sequential specification of the ADT. In Section 5, we will give concrete instantiations of this formula for the specific ADTs that we studied.

4. *failed execution* - X is a failed execution if every total order superset of it is not included in the sequential specification.

[3]An execution is a partial order, which is transitive by definition. For conciseness, from here on end we omit edges that stem from transitivity from our execution descriptions.

$failed\,(X) = po\,(X) \wedge (\forall Y.\,(X \subseteq Y \wedge to\,(Y)) \implies \neg spec\,(Y))$,
where Y is also a partial order represented by boolean variables as above and
$X \subseteq Y \equiv \forall i, j.X_{ij} \implies Y_{ij}$.

5. *minimal* - X is a minimal failed execution, if no subset of it is a failed execution.
$minimal\,(X) = failed\,(X) \wedge (\forall i, j \in \{1...N\}\,.X_{ij} \implies \neg failed\,(X \setminus X_{ij}))$,
where $X \setminus X_{ij}$ defines a partial order that is identical to X in all indexes except for X_{ij} which is set to *false*.

6. *Graph Isomorphism (GI)* - formal description of the isomorphism criteria of the specific ADT. As with the spec condition, concrete instantiations of this formula for the specific ADTs that we studied will be given in section 5. Since this criteria is dependent on previously found solutions it is added iteratively throughout the run of the SMT solver.

In listing 3 we specify how to employ the above definitions to generate the test set:

- Line 1 - define the boolean variables that are used in the formula.
- Line 2 - define the formula to identify assignments that constitute a minimal failed execution, as discussed above.
- Line 3 - initialize the empty test set.
- Line 4 - find the first assignment that satisfy the formula.
- Line 5 - we start the iterative process of finding all assignments satisfying the formula.
- Line 6 - add the current assignment to the test set.
- Line 7 - add the GI condition to the formula so subsequent iterations will not include current assignment or any assignment equivalent to it.
- Line 8 - obtain the next assignment.
- Line 9 - the process repeats until no further assignments can be found.
- Line 10 - return the test set

```
1  X := {X_ij | ∀i, j ∈ {1...N}}
2  formula := (minimal(X))
3  testSet := {}
4  res := solver.check(formula) //find a satisifiying assignment
5  REPEAT
6      testSet.add(res)
7      formula=formula ∧ ¬GI(res)
8      res = solver.check(formula)
9  UNTIL res is empty
10 RETURN testSet
```

Listing 3: SMT solver iterations to find failed execution graphs

4.2 Generating Test Suites

Given a test set and an implementations, our goal is to check if the given implementation can produce any of the test cases in the set. The naive approach would be to let a test program run many times and check whether the runs reproduced any of the executions. However, this approach is not robust since there is no guarantee if or when a possible execution will be encountered. Moreover, certain hardware architectures and compilers prohibit behaviors even if the programmatic model allows them. Case in point, x86 processors do not support the relaxed order falling back to release/acquire behavior instead. As such, testing by running test programs on specific processors would miss any issues that may occur using said semantics. Therefore, we chose to make use of model checking. This means that given a test set, we construct a *test suite*, which is a collection of executable C programs, each represnting a single test case and can be verified by a model checker. Next, we present the general construction of test suites, and then the necessary adjustments for efficient model checking.

4.2.1 Generating a Test Program from a Test Case

We suggest the following method to turn a failed execution into a simple C test program for the model checker. Note that since each test program correlates with a failed execution, in a correct implementation, the model checker should not find any complete execution paths running through the entire test. To achieve a generic, yet simple, method of test program generation, we encapsulate each operation in its own thread. To force order between specific operations we use atomic boolean variables, with release/acquire memory order accesses. Concretely, each thread sets an atomic flag before exiting which other threads can "choose" to wait on it if required. See listing 4 for a detailed template. To complete the test, we create a main method that runs all the thread functions and waits for them to complete. Since the tests should not complete, we

```
1  void *thread_{{i}}(void *arg) {
2      // add the following for every other method call j that proceeds this one
3      int val_{{j}} = 0;
4      __VERIFIER_assume(atomic_load_explicit(&val_{{j}}, memory_order_acquire) == 1)
5
6      // perform the method call on the ADT
7      // for example, Enq:
8      int * val = (int *)malloc(sizeof(int));
9      *val = {{i}};
10     enq(&q, val);
11
12     // or Deq:
13     int * res = NULL;
14     bool res = deq(q, &v)
15     // validate success of the operation
16     __VERIFIER_assume(res);
17     // validate expected result
18     __VERIFIER_assume(*v == {{i}}/2 + 1)
19
20     // set the flag
21     atomic_store_explicit(&f_{{i}}, 1, memory_order_release);
22     return NULL;
23 }
```

Listing 4: Template for generated thread code for queues,{{i}} representsthe index of a specific thread.

```
1  int main()
2  {
3      // init the ADT, for example:
4      init(q);
5
6      // add the following for every method call in the execution
7      pthread_t t_{{i}};
8      if (pthread_create(&t_{{i}}, NULL, thread_{{i}}, NULL))
9          abort();
10
11     // add the following for every thread created
12     pthread_join(t_{{i}}, NULL);
13
14     assert(0);
15 }
```

Listing 5: Template for generated main code,{{i}} represents the index ofa specific thread

add an assert in the end, as seen in listing 5 (line 14).

For example, a sample test program for the simplest failed execution for a queue $\{\langle Deq/1, Enq(1)\rangle\}$ is given in Listing 6. In the listing, *thread_ 0* (lines 8-20) performs the Enq operation, *thread_ 1* (lines 23-38) performs the Deq operation and the main method (lines 40-55) manages the life cycles of the threads. Note the order enforced between the operations by the release/acquire pair of operations on the *f_ 1* variable (lines 35 and 11 respectively) and the assertion at the end of the main method (line 54) that fails the test in case it is about to complete.

Additionally, note the use of pointers as ADT values in listing 4 and 6 as this is crucial when

testing for data races. When working with a pointer, any dereference has the potential to create a race condition, if not handled properly as part of the data structure implementation. To make sure we correctly test for these potential issues, our method expects implementations that work with pointers values and makes sure to dereference these pointers after retrieving objects from the data structure.

4.2.2 Adjusting test cases for GenMC

We use the GenMC model checker [12]. GenMC is exhaustive, in the sense that it attempts all possible executions paths within the test client program. As such, it is limited in the amount of concurrency it can handle computationally as each branch in the code, created by conditional or loop statements, creates another execution for it to explore.

GenMC provides some tools to help deal or get around these limitations. We used these tools to optimize both the test programs and the tested implementations. The most significant of these is the "assume" semantic introduced by GenMC via the __VERIFIER_assume statement which instructs the checker to discard execution branches in which the given condition is not met. For instance, the __VERIFIER_assume statement in line 4 of listing 4 and line 11 of listing 6 is equivalent to a the following loop

```
while(!atomic_load_explicit(&val_1, memory_order_acquire))
```

but provides better performance. This is also used to validate return values as seen in listing 4 lines 16-18 and listing 6 lines 32-33, thus limiting the checker's analysis to branches of interest for this specific test. As a result, it reduces the amount of branches GenMC has to evaluate dramatically and allows for more tests to complete. Occasionally, we also employed this technique on specific parts of the tested implementations. This has to be done very delicately so not to alter the original's behavior and possibly mask issues that were there to begin with.

Another possible optimization to reduce GenMC's overhead is to manage the degree of freedom in the test code. The degree of freedom is defined by the number of unordered threads that run concurrently as part of the test. For example, consider the *execution* discussed in the previous section, $\{\langle deq(1), enq(1)\rangle\}$. Generating this test program with over a set of *N=2 Ops* will result in a trivial sequential test, which is not very interesting to run on the model checker. Alternatively, over a set of *N=10 Ops*, we will produce 8 completely unordered additional threads.

```c
// ['deq(1)_enq(1)']
queue_t q;

atomic_int f_0;
atomic_int f_1;

// enq(1)
void *thread_0(void *arg)
{
    int val_1 = 0;
    __VERIFIER_assume(atomic_load_explicit(&val_1, memory_order_acquire) == 1)

    int * val = (int *)malloc(sizeof(int));
    *val = 1;
    q_enqueue(&q, val);

    atomic_store_explicit(&f_0, 1, memory_order_release);

    return NULL;
}

// deq(1)
void *thread_1(void *arg)
{
    set_thread_num(1);

    unsigned int res = 0;
    bool succ = q_dequeue(&q, &res);

    int * res = NULL;
    bool succ = q_dequeue(&q, &res, 5);
    __VERIFIER_assume(succ);
    __VERIFIER_assume(*res == 1);

    atomic_store_explicit(&f_1, 1, memory_order_release);

    return NULL;
}

int main()
{
    q_init_queue(&q, 2);

    pthread_t t_0;
    if (pthread_create(&t_0, NULL, thread_0, NULL))
        abort();

    pthread_t t_1;
    if (pthread_create(&t_1, NULL, thread_1, NULL))
        abort();

    pthread_join(t_0, NULL);
    pthread_join(t_1, NULL);
    assert(0);
}
```

Listing 6: Sample test generated for a simple queue failed execution

These additional threads, while important to add "stress" to the test, also create a very large number of branches that have a disabling effect on GenMC. We use the following logic to optimize the number of threads per test program. Let K be the width of the *failed execution*, as defined on partially ordered set [7], and C the selected thread cap. If $K \geq C$, we keep all the participating threads in the test. Otherwise, we add additional $C - K$ unordered threads. We used Dilworth's theorem to indirectly calculate K by calculating the length of the max antichain in the set.

To complete our previous example, for $C = 6$ we would get $K = 2$ ordered threads along with additional $C - K = 4$ unordered threads. The value of C itself varies by the implementations as their inner complexity has a large impact on GenMC's performance.

4.2.3 Testing with Noise

An alternative to using the original operations as unordered thread functions, is to define configurable noise functions. The noise functions can perform several sequential method calls on the tested implementation that add more stress and additional data into the data structure, without further increasing the parallelism for the model checker to test. See listing 7 for a how such a noise function can be implemented for queues.

4.3 Testing and Optimizing an Implementation

Using generated test suites, we can now test any compatible[4] implementation using the model checker. The testing process consists of linking the test programs in a test suite to the given ADT implementation and running the model checker on the test programs one by one. For each run, the checker outputs the number of executions it analyzed (completed and blocked). In case the checker encounters an error, such as an assertion failure, a data races or an access violation, it stops the analysis of the specific test and outputs the execution which caused the error. We sort these outputs into these possible results.

1. Test passed - all executions were blocked and no errors were encountered.

2. Test failed with type I issue - the checker reported an access violation or data race error.

[4] written in a language supported by the model checker

```
typedef struct noise_args
{
    queue_t *q;
    int tid;
    int count;
} noise_args;

void *noise(void *arg)
{
    noise_args *args = (noise_args *)arg;
    for (int i = 0; i < args->count; ++i)
    {
        int * val = (int *)malloc(sizeof(int));
        *val = (args->tid * 100) + i;
        enq(args->q, val);
    }

    for (int i = 0; i < args->count; ++i)
    {
        int * res = NULL;
        bool succ = deq(args->q, &res);
        if (succ) {
            assert(*res != -1);
        }
    }
    free(args);

    return NULL;
}
```

Listing 7: Implementation for a noise function for queues,note that the returnvalue is not validated,unlike in the standard thread function

3. Test failed with type II issue - the checker reported a completed execution branch, indicating the implementation produced a failed execution.

We suggest a two phased approach for testing and optimizing an implementation. In the testing phase, we run the test suites on the original implementation, fixing any issues encountered, until all the tests pass. In the optimization phase, we iteratively relax the memory order of atomic accesses as much as possible without breaking any of the tests. This phase can potentially be very exhaustive, as some implementations have many atomic accesses. We chose to perform it manually and for the most part, it converged pretty quickly to produce an optimized version of the implementations. However, we do believe that for large scale testing of many complex implementations, this entire process can be automated.

5 Applying The Method to Queues and Sets

In this section we demonstrate how to apply our method to two selected ADTs, the Queue and Set ADTs. For each of these ADTs, we will discuss how to define the sequential specifications.

In the case of the queue ADT, we also discuss how to define the graph isomorphism condition.

5.1 Queues

A queue can be informally specified as follows:

1. It supports two functions:
   (a) Enqueue (denoted *Enq*) - adds an item to the tail of the queue.
   (b) Dequeue (denoted *Deq*) - remove and return the item at the head of the queue.
2. It uses FIFO semantics - the Deq operation will attempt to remove the oldest item in the queue.
3. If Deq is called on an empty Queue, it returns the predefined value $\perp$.

Using these traits we can now define the sequential specifications for the queue ADT.

We suggest the following queue specific definitions to generate the logical formula that can be used to identify failed executions. We chose to work with distinct values. That is, every Enq operation will use a new value. We find this selection appropriate since we assume the queues' implementations are value agnostic, specifically meaning that the values enqueued should not affect the queue's behavior. This is expected based on the aforementioned traits and was subsequently verified on the implementations we tested as they never even read the actual value being enqueued. Therefore, using distinct values should not affect the comprehensiveness of the testing and serves as a convenience. We define a set of N operations, comprised of k pairs of Enq/Deq operations on k distinct values along with $N - 2k$ additional calls to $Deq/\perp$. To make the operations easier to reference, we derive an ordered list from it. The list is sorted ascending by value, with operations on the same value appearing consecutively, starting with the Enq operation. All the $Deq/\perp$ calls appear at the end of the list. Therefore, an operations list[5] with $N = 5, k = 2$ is defined as follows: $[Enq(1), Deq/1, Enq(2), Deq/2, Deq/\perp]$.

Sequential Specifications of Queues Given an ordered operations list, as described above, the following defines a valid sequential behavior that can be used in the logical formula.

[5] for future reference, we consider the list index to start from 1

1. Asserting that for any specific value x it is always enqueued before it can be dequeued:
   $\bigwedge_{i\in\{1\ldots k\}} X_{2i-1,2i}$

2. Asserting that values must be dequeued in the same order they were enqueued:
   $\bigwedge_{i,j\in\{1..k\},i\neq j} (X_{2i-1,2j-1} \iff X_{2i,2j})$

3. Asserting that $Deq/\perp$ can only occur after all values enqueued before it have been dequeued:
   $\bigwedge_{i\in\{1..k\},j\in\{N-2k\ldots N\}} (X_{2i-1,j} \iff X_{2i,j})$

Graph Isomorphism. Following is the relevant implementation of GI for queues. Two execution graphs $G_1=(Ops, X), G_2=(Ops, X')$, describing queue executions as previously defined, are considered isomorphic if it is possible to construct a mapping relation $M : \{1\ldots N\} \to \{1\ldots N\}$ denoted by M_{ij}, which adheres to the following:

1. Every operation in included in the mapping:
   $\bigwedge_{i\in\{1\ldots N\}} \left(\bigvee_{j\in\{1\ldots N\}} M_{ij}\right)$

2. No two different operations in the source are mapped to the same target operation:
   $\bigwedge_{i,j,k\in\{1\ldots N\}} (\neg M_{ik} \vee \neg M_{jk})$

3. An edge in Xcannot be mapped to a non-edge in $X'$and vice versa:
   $\bigwedge_{i,j,k,l\in\{1\ldots N\},i<j,k\neq l} ((X_{ij} \iff \neg X'_{kl}) \implies (\neg M_{ik} \vee \neg M_{jl}))$

The aforementioned rules are completely generic and apply to any graphs. However, considering the specifics of queues and the way we chose to describe them, some additional rules are required in order to get a minimal result set. We consider these additional rules as a form of "color coding" for the nodes in the graph, both by the function they describe and values. Denote $type(i) \in \{Enq, Deq, Deq\perp\}$ as the type of function the i-th call represents. Remembering how we chose to order our operations list, $type$ is actually directly derived from the index i as follows:

$$type(i) = \begin{cases} Deq\perp & i > 2k \\ Enq & i \mod 2 = 1 \\ Deq & i \mod 2 = 0 \end{cases}$$

We can now use this notation to define the final rules:

1. Operations must be mapped to operations of the same type:

   $\bigwedge_{i,j\in\{1\ldots N\},type(i)\neq type(j)} (\neg M_{ij})$

2. Mapping must retain the order between Enq and Deq operations of the same value:

   $\bigwedge_{i,j\in\{1\ldots k\}} (M_{2i-1,2j-1} \implies M_{2i,2j})$

5.2 Sets

We continue with the application our method to the set ADT. A set can be informally specified as follows:

1. $Add\,(x)$ - adds x to the Set if it does not already exist. Returns true iff the operation successfully added the item to the Set.
2. $Remove\,(x)$ - removes x from the Set if it exists. Returns true iff the operation successfully removed the item from the Set.
3. $In\,(x)$ - Returns true iff x currently in the Set.

We can now formally define the sequential specifications for the set ADT.

We suggest the following set specific definitions to generate the logical formula to find failing executions. It is easy to see from the above definitions that Set operations are not value agnostic. That is, all methods are called on specific values and consecutive operations with the same values may return different results. For example, whereas in Queues, calling $Enq\,(x)$ k times should all complete successfully and result with a queue of k entries. Calling $Add\,(x)$ k times should return *False* for all but the first call and result in a set with a single entry. Considering this, we see value in testing all operations with every possible return value. This will produce at least 6 operations to test per value x:

$$\{Add\,(x)\,/True, Add\,(x)\,/False, Remove\,(x)\,/True,$$
$$Remove\,(x)\,/False, In\,(x)\,/True, In\,(x)\,/False\}$$

Note that $Add\,(x)\,/False$ and $In\,(x)\,/True$ are equivalent in the sense that they do not alter the state of the set and provide the same information on it. The same can be said for $Remove\,(x)\,/False$ and $In\,(x)\,/False$. Therefore, for simplicity and scalability we claim that it

is sufficient to test the following subset of operations for each value:

$\{Add\left(x\right)/True, Remove\left(x\right)/True, In\left(x\right)/True, In\left(x\right)/False\}$.

Regarding the values themselves, considering many implementations employ some sort for of Search Tree as the basis for the set, using operations with different values in different orders appears to be mandatory to generate interesting test cases. Therefore, the list of operations should include adding and removing at least 3 values.

Similarly to how it was defined for queues, an operation list with k values would be ordered so operations relating to the same value would be consecutive. Specifically, the first operation would be *Add*, followed by *Remove*, successful *In* and lastly unsuccessful *In*. For example, such a list with $k = 3, N = 4k = 12$ operations would look as follows:

$$\left[Add\left(1\right), Remove\left(1\right), In\left(1\right)/True, In\left(1\right)/False, Add\left(2\right), Remove\left(2\right),\right.$$
$$\left. In\left(2\right)/True, In\left(2\right)/False, Add\left(3\right), Remove\left(3\right), In\left(3\right)/True, In\left(3\right)/False\right]$$

Sequential Specifications of Sets Given an operation list as described above, the following defines a valid sequential set behavior that can be used in the logical formula:

1. For any specific value x, it is always successfully added to the set before it can be successfully removed from it:

   $\bigwedge_{i\in\{1\ldots k\}} X_{4i-3,4i-2}$

2. For any specific value x, if $In\left(x\right)/True$ has to occur between add and remove:

   $\bigwedge_{i\in\{1\ldots k\}} \left(X_{4i-3,4i-1} \wedge X_{4i-1,4i-2}\right)$

3. For any specific value x, if $In\left(x\right)/False$ has to occur either before x was added or after it was removed:

   $\bigwedge_{i\in\{1\ldots k\}} \left(X_{4i,4i-3} \vee X_{4i-2,4i}\right)$

Considering what we described above about the nature of the implementations, permutations of executions with different orders between values, can produce non-equivalent executions in how the inner state of the implementation behaves. For example, the following executions

$$\{\langle Add\left(1\right)/True, Remove\left(2\right)/True\rangle, \langle Add\left(1\right)/True, Remove\left(3\right)/True\rangle,$$
$$\langle Remove\left(1\right)/True, Add\left(2\right)/True\rangle\}$$

$$\{\langle Add\,(2)\,/True, Remove\,(1)\,/True\rangle, \langle Add\,(2)\,/True, Remove\,(3)\,/True\rangle\,,$$
$$\langle Remove\,(2)\,/True, Add\,(1)\,/True\rangle\,\}$$

Can produce different results on certain implementations. Therefore, we chose to keep the GI condition empty (i.e. no two execution graphs are considered isomorphic).

6 Implementation and Results

In this section we review the results of implementing our method on the queue and set ADTs. In both cases, we implemented the logical formulas in Python, using the pySMT API and ran them using the binding to Z3 SMT solver [6]. We will first review the test sets generated for each ADT. Then, the process of converting the test sets to test suites. Finally, we provide examples for applying the testing and optimization process described in section 4.3, on selected implementations of these ADTs.

6.1 Queues

In this section we bring the generated test sets for the queue ADT, followed by examples of queue implementations we tested and optimized using these tests.

6.1.1 Generated Test Sets

We used the algorithm described in the previous section to generate two separate test sets, both for $N = 10$ operations. One with $k = 5$ distinct value pairs and another with $k = 4$ distinct value pairs and 2 $Deq/\bot$ operations. The Python process took about 15 minutes to complete per set. Following are the result test sets per parameter.

For $k = 5$ (i.e. no $Deq/\bot$ operations):

1. $\{\langle Deq/1, Enq\,(1)\rangle\}$
2. $\{\langle Enq\,(1)\,, Enq\,(2)\rangle\,, \langle Deq/2, De1/1\rangle\}$
3. $\{\langle Deq/1, Deq/4\rangle\,, \langle Deq/2, Deq/3\rangle\,, \langle Enq\,(3)\,, Enq\,(5)\rangle\,, \langle Enq\,(4)\,, Enq\,(2)\rangle\,, \langle Deq/5, Enq\,(1)\rangle\}$
4. $\{\langle Enq\,(1)\,, Enq\,(2)\rangle\,, \langle Deq/2, Enq\,(4)\rangle\,, \langle Deq/4, Deq/3\rangle\,, \langle Enq\,(3)\,, Deq/1\rangle\}$

5. $\{\langle Deq/1, Enq\,(5)\rangle, \langle Enq\,(2), Enq\,(4)\rangle, \langle Deq/3, Deq/2\rangle, \langle Deq/4, Enq\,(1)\rangle, \langle Deq/5, Enq\,(3)\rangle\}$
6. $\{\langle Deq/1, Deq/4\rangle, \langle Deq/2, Enq\,(1)\rangle, \langle Deq/3, Enq\,(2)\rangle, \langle Enq\,(4), Enq\,(3)\rangle\}$
7. $\{\langle Enq\,(1), Deq/2\rangle, \langle Enq\,(2), Enq\,(5)\rangle, \langle Deq/3, Enq\,(4)\rangle, \langle Deq/4, Deq/1\rangle, \langle Deq/5, Enq\,(3)\rangle\}$
8. $\{\langle Deq/1, Deq/2\rangle, \langle Enq\,(2), Enq\,(4)\rangle, \langle Deq/4, Deq/3\rangle, \langle Enq\,(3), Enq\,(1)\rangle\}$
9. $\{\langle Deq/1, Enq\,(2)\rangle, \langle Enq\,(3), Enq\,(1)\rangle, \langle Deq/2, Deq/3\rangle\}$
10. $\{\langle Deq/2, Enq\,(1)\rangle, \langle Deq/1, Enq\,(2)\rangle\}$
11. $\{\langle Deq/1, Enq\,(2)\rangle, \langle Deq/2, Enq\,(3)\rangle, \langle Deq/3, Enq\,(1)\rangle\}$
12. $\{\langle Deq/1, Enq\,(5)\rangle, \langle Deq/2, Enq\,(4)\rangle, \langle Deq/3, Enq\,(1)\rangle, \langle Deq/4, Enq\,(3)\rangle, \langle Deq/5, Enq\,(2)\rangle\}$
13. $\{\langle Deq/1, Enq\,(4)\rangle, \langle Deq/2, Enq\,(3)\rangle, \langle Deq/3, Enq\,(1)\rangle, \langle Deq/4, Enq\,(2)\rangle\}$

? minimal! 4 to 6! ?

Additional failed executions For $k = 4$ (i.e. with 2 $Deq/\perp$ operations):

1. $\{\langle Enq\,(1), Deq/3\rangle, \langle Enq\,(1), Deq/\perp_1\rangle, \langle Enq\,(2), Deq/1\rangle, \langle Enq\,(3), Deq/2\rangle, \langle Deq/\perp_1, Deq/3\rangle\}$
2. $\{\langle Enq\,(1), Deq/2\rangle, \langle Enq\,(2), Enq\,(3)\rangle, \langle Deq/3, Deq/4\rangle, \langle Deq/3, Deq/\perp_1\rangle, \langle Enq\,(4), Deq/1\rangle, \langle Deq/\perp_1, Deq/4\rangle\}$
3. $\{\langle Deq/1, Deq/4\rangle, \langle Enq\,(2), Enq\,(1)\rangle, \langle Enq\,(3), Deq/2\rangle, \langle Enq\,(4), Deq/3\rangle, \langle Enq\,(4), Deq/\perp_1\rangle, \langle Deq/\perp_1, Deq/3\rangle\}$
4. $\{\langle Enq\,(1), Deq/2\rangle, \langle Enq\,(1), Deq/\perp_1\rangle, \langle Enq\,(2), Deq/1\rangle, \langle Deq/\perp_1, Deq/2\rangle\}$
5. $\{\langle Enq\,(1), Enq\,(2)\rangle, \langle Enq\,(1), Deq/\perp_1\rangle, \langle Deq/2, Deq/3\rangle, \langle Enq\,(3), Deq/1\rangle, \langle Deq/\perp_1, Enq\,(2)\rangle\}$
6. $\{\langle Enq\,(1), Deq/1\rangle, \langle Enq\,(1), Deq/\perp_1\rangle, \langle Deq/\perp_1, Deq/1\rangle\}$
7. $\{\langle Enq\,(1), Deq/3\rangle, \langle Deq/2, Deq/1\rangle, \langle Deq/2, Deq/\perp_1\rangle, \langle Enq\,(3), Enq\,(2)\rangle, \langle Deq/\perp_1, Deq/1\rangle\}$
8. $\{\langle Deq\,(1), Deq\,(3)\rangle, \langle Enq\,(3), Deq/2\rangle, \langle Enq\,(3), Deq/\perp_1\rangle, \langle Enq\,(2), Enq\,(1)\rangle, \langle Deq/\perp_1, Deq/2\rangle\}$
9. $\{\langle Enq\,(1), Enq\,(2)\rangle, \langle Enq\,(1), Deq/\perp_1\rangle, \langle Deq/2, Deq/4\rangle, \langle Enq\,(3), Deq/1\rangle, \langle Enq\,(4), Deq/3\rangle, \langle Deq/\perp_1, Enq\,(2)\rangle\}$
10. $\{\langle Enq\,(1), Deq/3\rangle, \langle Enq\,(2), Deq/1\rangle, \langle Enq\,(2), Deq/\perp_1\rangle, \langle Enq\,(3), Deq/4\rangle, \langle Enq\,(4), Deq/2\rangle, \langle Deq/\perp_1/Deq/1\rangle\}$

There are several things to note when about these sets:

1. They are indeed minimal. It is easy to see that no two executions are equivalent or contain another execution or an equivalent of it.

2. ~~Since the sets were generated independently,~~ all executions that use 4 values or less with no $Deq/\perp$ operations appeared in both sets, as expected. For the sake of brevity, we filtered these duplicate executions from the results listed for the second set ($k = 4$).

3. The failed execution reproducing the issue found in the original implementation of the HW queue, as presented in Part 2, is execution #4 of the first set.

4. In the second set, although we introduced two instances of $Deq/\perp$ operations, no execution contains two such separate instances. This suggests, although does not formally prove, that adding multiple instances of this operation does not affect the resulting test set. This was corroborated by running the SMT solver with $N = 9, k = 4$ and obtaining the same set.

6.1.2 From Test Cases to Test Programs

We used the process of generating a suite of test programs from a test set as described in section 4.1.3 directly to the generate two test suites for the queue ADT. We generated 4 such test suites, 2 for each test set. The first with a thread cap $C = 4$ and the second with $C = 5$. This had two objectives: first, to try and create as much stress as possible on the queue; Second, to try and asses if this added noise is significant to the testing process.

Noise Generations Using the fact that Queues are value agnostic, our only requirement from the noise was that it would not "interfere" with the test cases. This was easy to do simply by using Enq operations with different values from the ones used in the test case. For the Deq operations we only made sure that they were successful and did not validate the specific values.

The test suites are available in ...

6.1.3 Testing And Optimizing Selected Queue Implementations

We tested several queue implementations. Some of which we took from reference libraries, such as GenMC's own testing library, others from peer recommendations or around the net. The following table summarizes our findings regarding these queues, followed by a short description of the work performed on each implementation. Note that due to GenMC's limitations, not all

tests completed on all implementations which we discuss with more detail per implementation. References to all implementations, including both the original version and our optimized one, appear in appendix A. The summary table covers the following attributes:

- Name - the name of the implementation.
- Completion - the completion status for the test suite runs on this implementation. Each test program was given at most one hour to complete before it was terminated (indicated as t/o).
- Time - the time it took to complete the runs of all the test suites on this implementation's optimized version.
- Type I - whether we identified issues from the first correctness criteria (i.e. data races) in the original implementation.
- Type II - whether we identified issues from the second correctness criteria (i.e. inconsistencies) in the original implementation.
- Accesses Before - details the amount and type of atomic accesses in the original implementation.
- Accesses After - details the amount and type of atomic accesses in the optimized implementation.

We used the following shorthand to describe memory order: rlx (relaxed), acq (acquire), rel (release), SC (sequential consistent) and acq_rel (acq followed by release, relevant for certain accesses such as FAA). For CAS operations, we used the notation x/y to note the memory order used by the operation when successful and unsuccessful respectively.

Name	Completion	Time	Type I	Type II	Accesses Before	Accesses After
Chase-Lev	1 test t/o with $C = 4$ 2 tests t/o with $C = 5$	6h	Yes	No	7 (2 rlx, 2 acq, 1 rel, 1 SC, 1 rel/rel)	8 (2 rlx, 3 acq 2 rel, 1 rel/acq)
Fast-MPMC	Completed	14m	No	No	8 (2 rlx, 2 acq, 2 rel, 2 rlx/rlx)	9 (5 rlx, 1 acq, 1 rel, 2 rlx/rlx)
HW	1 test t/o with $C = 4$ 4 tests t/o with $C = 5$	6h 15m	No	Yes	4 (1 acq, 2 rel, 1 acq_rel)	4 (2 rlx, 1 acq, 1 acq_rel)
LCRQ	Failed	N/A	N/A	N/A	16 (all SC)	N/A
MPMC	Completed	2h 15m	No	No	8 (4 acq, 2 rel, 2 acq_rel/ acq_rel)	8 (2 rlx, 2 acq, 2 rel, 2 rlx/rlx)
MS	half of the tests t/o with $C = 4$ did not run $C = 5$ suite	N/A	Yes	No	15 (5 rlx, 5 acq, 5 rel/rlx)	15 (3 rlx, 7 acq, 5 rel/acq)
QU	1 test t/o in both suites	3h	No	No	12 (3rlx, 3 acq, 4 rel, 2 rlx/rlx)	12 (9 rlx, 1 acq, 1 rlx/rlx, 1 rel/acq)
UNIQ	Completed	40m	Yes	No	8 (All SC)	13 (9 rlx, 1 acq, 1 rel, 2 rlx/rlx)

Chase-Lev Queue Based on [5], the original implementation was taken from GenMC testing library. The original implementation was known to be flawed to begin with, evident by the following excerpt from the original code:

```
1  struct Deque {
2      atomic_uint_fast64_t bottom;
3      atomic_uint_fast64_t top;
4      int64_t buffer[LEN]; // in fact, it should be marked as atomic
5      //due to the race between push and
6      // steal.
7  };
```

Listing 8: Chase-Lev struct definition

Note the comment about the expected race condition. Since this is originally a Deque implementation, the terminology is different from what we use. In this case, the *push* function is mapped to *Enq* while *steal* is *Deq*. Although taking this comment into consideration, we decided to test the implementation as is, mainly because it still passed all the tests in the GenMC testing library with this flaw. As expected, when tested with our generated test suite, this flaw was immediately revealed and subsequently fixed. As a result, the implementation now had 3 global atomic variables, one of which is the data buffer. At first, we attempted to use relaxed accesses for the data buffer. However, this still resulted in a race, this time between two Enq operations. As a results, we used release/acquire accesses, as indicated by the added accesses in the "after" column. Note that with the added order, we were able to remove the SC access from the original implementation. Due to the relative complexity of the implementation, specific tests failed to complete.

Since each uncompleted test takes an hour of runtime, the total runtime of the completed tests is only about 3 hours.

Fast-MPMC Queue The original implementation taken from a public GitHub page (see Appendix A). The original code was written in C++ and used templates extensively so the first part of the work was to create an equivalent C version. Moreover, in the original implementation, operations would fail in cases where subsequent atomic accesses produced inconsistent results. That behavior is not consistent with our expectation from a tested implementation so we wrapped it with a `while(true)` loop as is the standard in these cases. Once wrapped, we noticed that a *Deq* operation would block on an empty queue, which meant that, among other things, all $Deq/\perp$ operations became impossible to test. We mitigated this by adding another check in the code to

address this case. This resulted in an additional relaxed access in the final version of the code. In total, this implementation uses 2 global atomic variables referencing the head and tail of the queue along with another atomic variable per entry that specifies its version. In the original code, all accesses to the version variable used release/acquire memory order. However, we noticed that the tests only require one such pair of accesses to use release/acquire which allowed us to relax the others.

HW Queue Based on [10], the original implementation was taken from the GenMC testing library. This implementation's specific issue was previously identified in [21] as discussed in section 1. Our test sets surfaced this issue in a completely standalone way, as is evident from this excerpt:

```
1  ./tests/generated/generic_10_0_10_True_4/t4.c
2  Sun Feb  7 09:22:09 IST 2021
3  Number of complete executions explored: 4
4  Number of blocked executions seen: 3562
5  Total wall-clock time: 0.26s
```

Here we can that for test $t4$, that corresponds with execution #4 in the first queue test set, the model checker was able to identify several possible code path, meaning that these code paths were not blocked by the implementation. By changing atomic FAA access memory order from release to acquire_release we were able to address this issue which the tests then verified. In total, as seen in listing 1, the implementation uses 2 global atomic variables, one of which is the data buffer. During optimization, we were able to relax the two accesses the the data buffer (AR).

LCRQ This queue was designed and implementated by Adam Morrison and Yehuda Afek [18]. It was originally written with pre-C11 atomic semantics. We rewrote it using C11 semantics, keeping all atomic accesses SC as a first step, for parity. However, this implementation was highly optimized by its designers making it significantly more complex than the others we tested. It consists of two global atomic variables pointing to the head and tail of the queue with each entry containing four additional atomic variables. These variables are accessed 16 times which is also significantly more than any other implementation we tested. As such, we failed to complete almost any test runs on it and could not progress to the optimization phase. We consider this a good example for an implementation that is beyond the limitations of the current tools we used to implement our method. If and when some of these limitations are alleviated, we hope this

implementation can be tested further.

MPMC Queue The original implementation taken from the GenMC testing library. It makes use of a total of 3 global atomic variables that indicate the number of read, write and total accesses made to the queue. In the original implementation, all accesses (4 during Enq and 4 during Deq) used acquire/release memory order. We were able to optimize and relax the accesses to one of the variables (rdwr), which account for half of the total accesses, leaving 2 in each method. Additionally, we observed that the other two counters (read and write) are used in a manner that very much resembles a spinlock. As seen in listing 8 the while loops in lines 18 and 44 are spin-waiting for other threads to synchronize the values written to the rdwr variable with those written to the read and write counters, essentially waiting for these threads to complete. This explains why accesses to these counters could not be relaxed and indicates that we cannot refer to this implementation as a wait-free or lock-free implementation.

MS Queue Based on the Michael-Scott Queue [17], the original implementation was taken from the GenMC testing library. It requires 2 global atomic variable (head and tail) and another one per queue entry (next). This implementation is also very complex compared to most others we tested, as exemplified by the 15 atomic accesses. Due to its complex nature we were not able to fully test this implementation. However, even with the partial testing we performed, we were able to identify races in the current implementation and had to make a couple of relaxed atomic reads stronger, changing their memory order to acquire. Since the testing was not complete, we cannot consider this queue optimized. Having said that, we do claim that our suggested version is more correct than the original. Similar to LCRQ, we consider this implementation another potential case for further testing once some limitations are mitigated.

QU Queue Also based on the Michael-Scott Queue with the original implementation taken from the GenMC testing library. It uses the same atomic variables as MS queue but with with different access patterns. In the original implementation, most accesses used acquire/release semantics. We were able to optimize the implementation so only a single access to the next field per operation required acquire/release. The rest were relaxed.

UNIQ Queue The original implementation was taken from a public GitHub page (see Appendix A). Moreover, the original implementation has changed significantly since we started working with it. It introduced busy waiting using sleep into the algorithm which is a big departure from the algorithm we wanted to optimize. Therefore, we chose to reference a previous revision. Like some of the other implementations, UNIQ was originally implemented in C++ and required us to rewrite an equivalent C implementation. In this case, the original implementation was not robust enough and in fact not thread safe according to our testing (#ref to our correspondence with the developer???). The original implementation used a non-atomic global vector of integers (isFree) and performed concurrent operations on it as part of both the Enq and Deq functions with no additional synchronization. The implementation did make use of two global atomic integers (in, out) all of the accesses to which employed SC memory order. Considering that, it is probable that the data race on the isFree field would go unnoticed in most test programs. However, the issue was discovered during our testing. For our optimized version, we made the vector field into an atomic field but kept all accesses to it relaxed. In fact, we were

```
1  bool deq(mpmc_boundq_1_alt *q, val_t ** retVal)
2  {
3      unsigned int rdwr = atomic_load_explicit(&q->m_rdwr, memory_order_relaxed);
4      unsigned int rd, wr;
5
6      while(true) {
7          rd = (rdwr >> 16) & 0xFFFF;
8          wr = rdwr & 0xFFFF;
9
10         if (wr == rd) // empty
11             return false;
12
13         if (atomic_compare_exchange_weak_explicit(&q->m_rdwr, &rdwr, rdwr + (1 << 16),
14             memory_order_relaxed, memory_order_relaxed))
15             break;
16     }
17
18     while((atomic_load_explicit(&q->m_written, memory_order_acquire) & 0xFFFF) != wr);
19
20     *retVal = (q->m_array[rd % t_size]);
21
22     atomic_fetch_add_explicit(&q->m_read, 1, memory_order_release);
23
24     return true;
25 }
26
27 void enq(mpmc_boundq_1_alt *q, val_t * val)
28 {
29     unsigned int rdwr = atomic_load_explicit(&q->m_rdwr, memory_order_relaxed);
30     unsigned int rd,wr;
31
32     while(true) {
33         rd = (rdwr>>16) & 0xFFFF;
34         wr = rdwr & 0xFFFF;
35
36         if (wr == ((rd + t_size) & 0xFFFF)) // full
37             break;
38
39         if (atomic_compare_exchange_weak_explicit(&q->m_rdwr, &rdwr,
40             (rd << 16) | ((wr + 1) & 0xFFFF), memory_order_relaxed, memory_order_relaxed))
41             break;
42     }
43
44     while((atomic_load_explicit(&q->m_read, memory_order_acquire) & 0xFFFF) != rd);
45
46     (q->m_array[wr % t_size]) = val;
47
48     atomic_fetch_add_explicit(&q->m_written, 1, memory_order_release);
49 }
```

Listing 9: MPMC implementation

able to relax all atomic accesses keeping only a single release operation in Enq that corresponds with a single acquire operation in Deq.

6.1.4 General Queue Observations

Generally speaking, our testing framework is designed to identify two types of infringements on the two success criteria we defined in Section 3, data-race freedom and consistency, respectively. From our work with the aforementioned implementations, it appears that eliminating the first type negated most issues with the second as well. Moreover, the way we test surfaces type I problems first since the model checker stops running a test when a data race violation occurs. Therefore, it is possible that type I problems mask type II ones and that by solving the former, we also solve the latter without even knowing. This can be further explained by analyzing the potential consistency issues that can arise when testing a Queue implementation. Generally speaking, we can identify two such specific issues. The first, denoted by "*Deq before Enq*", suggests that a *Deq* operation would remove and retrieve a value from the queue before the *Enq* operation was finalized. The second, denoted as "*out of order Deq*", suggests that a *Deq* operation retrieved an incorrect value considering the expected FIFO behavior. Regarding the "*Deq before Enq*" issue, the use we made of pointers made such issues observable as data races since the dereferencing operation would result in an access violation if the *Enq* operation has not yet been finalized. Therefore, the potential for type II issues was mostly reduced to "*out of order Deq*" issues which we observed, as expected, on the HW implementation.

Once we identified type I problems in the tested implementations, we were able to reproduce them using only "noise" threads. For example, if an access violation error was encountered while running a test with 3 Enq and 3 Deq operations, running any such test, including strictly based on "noise" threads, would reproduce the error. This is expected since from the model checker's perspective, we were testing a program comprised of the same set of functions but without any constraints. Therefore in the test run was even more exhaustive as more executions were covered, including, but not limited to, those covered by the original test.

As described in the previous section, we ran two test suites, with different levels of noise. It is worth noting that all the issues we observed occurred in both. As such, we could not assess the value provided by the added noise. This does not mean that we believe it will not be valuable in some specific cases, but we could not identify such cases.

All the implementations with uncompleted tests specifically failed test #10 from the second test set. This is one of the most complex tests in terms of thread involvement and as such, it pushed the limitations we are aware of with the model checker.

6.2 Sets

In this section we bring the generated test sets for the set ADT, followed by set implementation we tested and optimized using these tests.

6.2.1 QBF solution

As described in the previous section, we chose an operation set with 12 operations, and no filtering. This leaves us with all the permutations of failed executions that could be found on 12 operations and was beyond the limits of what we were able to generate with Z3 (more on this in the next section). Instead, split the *Ops* into two separate sets, one with $In\,(x)\,/True$ operations on all values and the other with $In\,(x)\,/False$. The reasoning behind this is that since these operations do not alter the state of the set, it should not be crucial to test them as part of the same test. Subsequently, we generated test sets for the following lists of operations:

$$[Add\,(1)\,, Remove\,(1)\,, In\,(1)\,/True, Add\,(2)\,, Remove\,(2)\,, In\,(2)\,/True, Add\,(3)\,, Remove\,(3)\,, In\,(3)\,/True]$$

$$[Add\,(1)\,, Remove\,(1)\,, In\,(1)\,/False, Add\,(2)\,, Remove\,(2)\,, In\,(2)\,/False, Add\,(3)\,, Remove\,(3)\,, In\,(3)\,/False]$$

Generating the sets took about 10 minutes per set and since there was no filtering of isomorphic solutions, both generated sets were significantly longer than those generated for queues. In total, we found 90 test cases in the first set and 184 in the second. For the sake of brevity they are brought in full in appendix B.

6.2.2 From Test Cases to Test Programs

We used the same process used for queues to generate multiple test suites from the two aforementioned test sets with different levels of noise. Due to the more complex nature of the set implementations as well as the tests themselves, we ended up testing only with minimal noise level (i.e. $C = 2$).

Depending on their inner data structure, set implementations perform some logical function on the input value to determine its desired destination within the data structure. Therefore, the value must be comparable to other values that may be added to the set. To allow for more generic use, specifically in languages like C which do not support interfaces or templates, sets may work with key/value pairs, allowing for the key to be of a comparable type and the value to be any object (i.e. void pointer). This differs from Maps or Dictionaries in that keys cannot be remapped to other values and these pairs are considered a single unit. In these cases, we chose to use an integer as the key and a pointer as the value, making sure to dereference and free the value immediately after it is removed from the set.

Noise Generation Since every operation on a Set is performed on a specific value, the trivial way to generate noise will be to perform additional Add and Remove operations on any values that are not part of the test case. For example, if the values in the test are from the range $\{1...n\}$ any operations performed on $\{n+1...\infty\}$ would not interfere. However, we consider this approach overly simplistic, specifically for Search Tree based implementations, since values which are out of range with the original tested values may cause only marginal change in the structure of the tree in the vicinity of the tested values. Therefore, we chose to space out the original values more and generate the noise values in between them. We used multiples of 10 and 11 for the values and noise respectively.

6.2.3 Testing And Optimizing a Set

Concurrent Sets are more complex to implement than Queues. Subsequently, many set implementations found in public testing libraries for general use employ mutex based locks and as such are not very interesting to analyze using our method. Additionally, the complexity also makes the optimization process much more delicate since the affect of relaxing an atomic access may not be confined to its close vicinity in the code. Therefore, we focused on one specific set implementation that is lock and wait free. As with queues, due to GenMC limitations, not all tests were able to complete. Following is the summary of the testing done on this implementation:

- Name - Howley Set [11]
- Completion - 3/274 tests timed out on both original and optimized versions.

- Time - about 15 hours on the optimized version.
- No Type I issues were found in the original implementation.
- No Type II issues were found in the original implementation.
- Accesses Before - 42 (All SC)
- Accesses After - 42 (20 rlx, 3 rlx/rlx, 7 acq, 10 rel/acq, 1 SC, 1 SC/SC)

The Howley Set is an extension of a BST. The original implementation was based on pre-C11 atomic operations, which were all SC. Therefore, the first step we did was to rewrite it completely with C11 atomic operations, keeping all accesses SC for parity. Post rewrite, the implementation uses 6 atomic variables per set entry. We then started the optimization process, which, due the complexity of the implementation, and the relative high amount of atomic accesses, took significantly longer than for the queues we tested. Eventually, our optimized version passed the same tests as the original one.

Almost all atomic accesses in the optimized version were relaxed to some extent, with only 2 SC accesses remaining. Since the model checker outputs the number of branches it analyzes during each test run, we were able to ascertain that the optimization actually increased the potential execution paths. For example, for test 115 in the second set the number of executions processed by the model checker went from 4,954,973 in the original implementation to 5,657,411 in the optimized one. These additional execution paths can be viewed as potential future optimizations for a compiler or processor to take advantage of. Full log files for test suite run on both SC and optimized versions are brought in Appendix B.

It is important to note that during the optimization process, tests frequently failed with both type I and type II issues that surfaced due to the use of the more relaxed accesses. Although this is just a single implementation, we believe it is a very good test case to demonstrate the validity of this method.

7 Limitations

In this section we offer a more in-depth analysis of the limitations we encountered and suggest possible directions to address these limitations, which were out of scope for this work. We focus

on the tools we used to implement our method, specifically the Z3 SMT solver and GenMC model checker. Furthermore, regardless of these tools and their limitations, we suggest additional work that can be done to extend our method and its use.

7.1 SMT Solver Limitations

One of the major quality factors of the our method is the amount of test cases it can generate, which directly affects the coverage it provides. The test cases are correlated with the number of potential threads in each case, the more potential threads, the more possible test cases. We assume that there is some kind of "convergence" point with regards to the potential bugs. That is, that it should not be necessary to test with an infinitely growing number of threads to find all the potential bugs (#ref, is there a study on this???). However, we do not know where that limit aspires to and assume it is larger than the 9-10 threads we are now able to cover.

The Z3 SMT solver we used to test the suggested method was blocked at about 10-11 threads, depending on the formula, and could not complete the iterative run with a larger number in a reasonable period of time (24 hours).

A possible direction to address this is to try other SMT solvers. Other solvers may give better overall performance, or at least better performance for some parts of the formula, allowing us to mix and match solvers on different parts of the logical problem. It may also be possible to further optimize the formula itself to have a better fit with one (or more) of these solvers. This can be done even more effectively as a joint effort with the solver's developers to not only optimize the formula for the solver but also the solver for the formula.

7.2 Model Checker Limitations

Test Suite Scale The GenMC model checker is also limited in the scale it can analyze in reasonable time. There are two major contributing factors this limitation. First, the complexity of the implementation, which for this case is mostly determined by the amount of conditional branches in the code (such as *if* statements and loops). The more conditional branches in the code, the number of paths the model checker has to analyze increases, potentially exponentially, when branches spawn other branches. The second factor is the complexity of the test itself. This is measured mainly by the number of "degrees of freedom" in the thread behavior of said

test (see section 4.2.2). For the simpler implementations, we were able to reach a total of about 6-8 threads that are not fully ordered between themselves, which coincided pretty well with the solver limitations of about 10 threads in total. Ultimately, as we saw in the previous section, this was enough to cover almost all of the generated tests for most all implementation but not all.

Since GenMC is still in active development, better results should be available as the work on the it continues and its overall performance is improved. Specifically, future version of GenMC are designed to employ a mechanism for symmetry reduction, which is meant find and ignore equivalent executions. This can prove a significant improvement to GenMC and allow it to analyze significantly more executions in less time. Additionally, manual optimization, such as those described in section 4 can be applied to some implementations to block more branches, based on their eventual outcome. If applied correctly, these optimization could allow to complete the testing of more complex implementations without sacrificing the result's validity over the original implementation.

Still, if we were to generate more extensive test sets, with significantly more threads we may still hit GenMC's hard limit at some point. At this point, we can look into other model checkers, with different approaches, not necessarily as full replacements but possibly as complementary to GenMC. As it seems that there is not necessarily a single checker best suited for all possible tests and implementations. Other approaches, based on more heuristic branch prediction instead of being 100% robust, might be able to improve performance while still being able to identify edge cases when analyzing the tests. One such potential model checker is the C11Tester [15]. It is based on iterative heuristic branch taking and could prove a good fit for out tests. At the moment, it has some limitations that prevent us from using it directly alongside GenMC. The main limitation is the lack of ability to "guide" the checker towards more specific branches we wish to explore, as we do with GenMC's "assume" semantic. If such features are added, it could prove to be a very complementary alternative for exploring very complex implementations and test cases that GenMC has more difficulties with.

C++ Support At the moment, GenMC only supports C programs. However, since our method addresses the RC11 memory model, which also applies to C++, we wanted to support C++ based implementations as well. For that, we had to rewrite some implementations, converting C++ to C while attempting to make all the memory usage completely equivalent. It

would be both more convenient and more robust to be able to test and optimize C++ implementations directly without the need for rewriting. This would reduce the work associated with this process and appeal to a much wider variety of developers and implementations, especially considering that a significant portion of software library development is done in C++. This can happen if C++ support is added to GenMC itself or with the use of additional model checkers such as the aforementioned C11Tester which supports C++ programs.

8 Related work

Gibbons, Bruno & Philips introduced a method to verify the correctness of specific behaviors of parallel data structures [9]. They provided a polynomial algorithm to validate executions. Theoretically, this algorithm can be used to find failed executions over of a set of operations but will require going over all executions one by one to do so. Moreover, in that approach, filtering isomorphic executions would have to be done separately as it is not part of the suggested algorithm. The SAT approach allowed us to do that more efficiently in a single logical formula. Additionally, their work only covered the queue and stack ADTs and did not address the set ADT at all.

Vafeiadis introduced an automatic way to validate linearizability [23] on ADT implementations. It was applied to multiple types of ADTs (including queues and sets) but only to implementations based strictly on SC atomic accesses.

In Violat [8], Emmi and Enea described a method to randomly generate tests for concurrent ADT implementations for the java virtual machine (JVM). It integrates with analysis backends to verify if unwanted behaviors (i.e. failed executions) are encountered. Besides being focused specifically on JVM programming languages, this approach differs from ours in that it is not focused on enumerating and testing failed executions and in having more limited scope of operations. By default, Violat generates tests made up of 2 threads, between 3 and 6 operations, and 2 values. Although these defaults can be extended, Violat's random and limited nature does not support identifying more complex test cases, specifically like the reference execution on HWQ we introduced in section 1.

Jonas et al. demonstrated a similar approach for optimizing atomic accesses in concurrent implementation, specifically optimizing implementations of synchronization primitives in [19].

Burckhardt, Alur & Martin used a different approach in CheckFence [4] albeit with similar tools. In their work, they convert C test programs to a SAT formula. They then use a SAT solver to enumerate all assignments possible for the formula under two different sets of constraints. One modeling a single threaded environment and the other a multi threaded one. Comparing the two, they report any exclusively multi-threaded executions they find as potential bugs. Although CheckFence was written with more relaxed memory model in mind, it was written before the C11 memory model was published and as such does not address its semantics (i.e. acquire/release) directly. The most significant difference from our approach is that CheckFence verifies a single test program, and does not enumerate or target failed executions specifically. In that sense it is more comparable to GenMC itself, which also includes an extensive testing library of ADT implementations. As with the other tools mentioned here, the tests themselves in GenMC's library are generic and not focused on failed execution.

9 Conclusion and Future Work

In this work, We have suggested a method to verify and optimize concurrent data structure implementations for the RC11 memory model. We demonstrated how it can be used to generate test suites for any ADT and provided such suites for the queue and set ADTs. We showed how these suites can be used to find and resolve bugs in existing implementations and suggested a process to optimize these implementations w.r.t. the use of atomic accesses. We provided such optimized versions of existing implementations that can be used in place of the original implementations going forward.

In the future, this method can be used to test and optimize additional existing implementations as well as become part of the R&D cycle for new RC11 optimized data structure implementations. This would be immediately applicable to additional queue and set implementations but is not limited to those in any way. Applying the method to additional ADTs (e.g. Stacks, Prioriy Queues, etc.) will require repeating the steps described in sections 5 and 6. That is, defining the correct sequential specification and graph isomorphism behavior and adding those to the logical formula. Then, generating the test sets and suites and linking them with specific implementations. As stated before, it is important to note that the first part needs to be done only once per ADT.

Another possible extension of this work is to automate the optimization process described in section 4.3. An process can be designed to relax memory order in atomic accesses of an implementation and run the test suites iteratively until the process converges and a C11 optimized version of the implementation is generated. The main question such a process will have to address is how to choose the rate and order in which to replace atomic accesses with relaxed versions. Generally, it can be determined completely randomly or by something similar to the *adaptive linear relaxation* described in VSync [19].

Appendix

A Data Structure Implementations

Chase-Lev Queue

Original https://github.com/MPI-SWS/genmc/blob/master/tests/correct/data-structures/chase-lev/deque.h

Optimized https://github.com/ori-saporta83/linearizability-testing/blob/master/wrappers/chase-lev-wrapper.h

FastMPMC Queue

Original https://github.com/spectre1989/fast_mpmc_queue/blob/master/MPMCQueue.h

Optimized https://github.com/ori-saporta83/linearizability-testing/blob/master/wrappers/fastmpmc-wrapper.h

HW Queue

Optimized Implementation: https://github.com/ori-saporta83/linearizability-testing/blob/master/wrappers/hwqu
wrapper.h

LCRQ Queue

C11-simplified https://github.com/ori-saporta83/linearizability-testing/blob/master/wrappers/lcrq/lcrq.c

MPMC Queue

Original https://github.com/MPI-SWS/genmc/blob/master/tests/correct/data-structures/mpmc-queue/mpmc-queue.h

Optimized https://github.com/ori-saporta83/linearizability-testing/blob/master/wrappers/mpmc-queue-wrapper.h

MS Queue

Original https://github.com/MPI-SWS/genmc/blob/master/tests/correct/data-structures/ms-queue/my_queue.c

Optimized https://github.com/ori-saporta83/linearizability-testing/blob/master/wrappers/ms-queue-wrapper.h

QU Queue

Original https://github.com/MPI-SWS/genmc/blob/master/tests/correct/data-structures/qu/qu.c

Optimized https://github.com/ori-saporta83/linearizability-testing/blob/master/wrappers/qu-wrapper.h

UNIQ Queue

Original https://github.com/bittnkr/uniq/blob/d4a5b9c413aef2a242a7f9654ded198fba78b9bf/cpp/uniq.h

Optimized https://github.com/ori-saporta83/linearizability-testing/blob/master/wrappers/uniq-wrapper.h

Howley Set

Original https://github.com/LPD-EPFL/ASCYLIB/blob/master/src/bst-howley/bst_howley.c

Optimized https://github.com/ori-saporta83/linearizability-testing/blob/master/wrappers/howley/bst_howley_o

B Set test sets and results

link here

References

[1] Alglave, J., Maranget, L., McKenney, P. E., Parri, A., and Stern, A. Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (2018), pp. 405–418.

[2] Alglave, J., Maranget, L., and Tautschnig, M. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS) 36*, 2 (2014), 1–74.

[3] Batty, M., Owens, S., Sarkar, S., Sewell, P., and Weber, T. Mathematizing c++ concurrency. *ACM SIGPLAN Notices 46*, 1 (2011), 55–66.

[4] Burckhardt, S., Alur, R., and Martin, M. M. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2007), pp. 12–21.

[5] Chase, D., and Lev, Y. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures* (2005), pp. 21–28.

[6] De Moura, L., and Bjørner, N. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), Springer, pp. 337–340.

[7] Dilworth, R. P. A decomposition theorem for partially ordered sets. In *Classic Papers in Combinatorics*. Springer, 2009, pp. 139–144.

[8] Emmi, M., and Enea, C. Violat: generating tests of observational refinement for concurrent objects. In *International Conference on Computer Aided Verification* (2019), Springer, pp. 534–546.

[9] Gibbons, P. B., Bruno, J. L., and Phillips, S. Black-box correctness tests for basic parallel data structures. *Theory of Computing Systems 35*, 4 (2002), 391–432.

[10] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[11] Howley, S. V., and Jones, J. A non-blocking internal binary search tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures* (2012), pp. 161–171.

[12] Kokologiannakis, M., Raad, A., and Vafeiadis, V. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019), pp. 96–110.

[13] Lahav, O., Vafeiadis, V., Kang, J., Hur, C.-K., and Dreyer, D. Repairing sequential consistency in c/c++ 11. *ACM SIGPLAN Notices 52*, 6 (2017), 618–632.

[14] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programm. *IEEE transactions on computers 28*, 09 (1979), 690–691.

[15] Luo, W., and Demsky, B. C11tester: a race detector for c/c++ atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), pp. 630–646.

[16] Manson, J., Pugh, W., and Adve, S. V. The java memory model. *ACM SIGPLAN Notices 40*, 1 (2005), 378–391.

[17] Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (1996), pp. 267–275.

[18] Morrison, A., and Afek, Y. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming* (2013), pp. 103–112.

[19] Oberhauser, J., Chehab, R. L. d. L., Behrens, D., Fu, M., Paolillo, A., Oberhauser, L., Bhat, K., Wen, Y., Chen, H., Kim, J., et al. Vsync: Push-button verification and optimization for synchronization primitives on weak memory models. In

Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (2021), pp. 530–545.

[20] Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., and Sewell, P. Simplifying arm concurrency: multicopy-atomic axiomatic and operational models for armv8. *Proceedings of the ACM on Programming Languages 2*, POPL (2017), 1–29.

[21] Raad, A., Doko, M., Rožić, L., Lahav, O., and Vafeiadis, V. On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 1–31.

[22] Torán, J. On the resolution complexity of graph non-isomorphism. In *International Conference on Theory and Applications of Satisfiability Testing* (2013), Springer, pp. 52–66.

[23] Vafeiadis, V. Automatically proving linearizability. In *International Conference on Computer Aided Verification* (2010), Springer, pp. 450–464.