```
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

Abstract

1 Introduction

2 Background

more background on the issues in working with weak memory models (WMM) of RC11 (more or less: C11+acyclic(PO $\bigcup$ RF)), which is monotone. what does the model offer and the trade-of between consistency and performance.
Consider the queue implementation (based on HW #reference) in listing 1: In the article by (#ref) the following trace, is identified as theoretically achievable execution of the implementation in listing 1:

$$\begin{array}{cccc} a : enq\,(q, v_1) & b' : deq\,(q)\,//v_2 & c' : deq\,(q)\,//v_3 & d : enq\,(q, v_4) \\ b : enq\,(q, v_2) & c : enq\,(q, v_3) & d' : deq\,(q)\,//v_4 & a' : deq\,(q)\,//v_1 \end{array}$$

Note that this trace is inconsistent with the sequential definitions of queue behavior. This is evident since there is no way to create a full order of these operations while maintaining the happens-before and program-order relations with the constraints of the queue behavior without creating a cycle. In the article it is suggested that altering the memory order in the FAA operation in line 6 to *memory_order_acq_rel* will solve this issue and the aforementioned trace will no longer be achievable.

2.1 Model Checking

Our approach to identifying incorrect implementations requires us to have the ability to verify whether a certain history H of operations can be achieved on a specific implementation of an ADT. The naive approach to this task is to let a test program run many times and check whether H is produced by any of the runs. However, this approach is not robust since there is no guarantee that a possible history will be encountered. Moreover, certain HW implementations 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 these processors would miss any issues that may occur using said semantics. Therefore, we suggest to make use of 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. First, describing what we are trying to achieve when examining any concurrent implementation of an ADT. Then, the specific criteria we base our method on.

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: First, to have the ability to test whether any specific ADT implementation can be used safely, concurrently, under the RC11 weak memory model. Second, to suggest a variation of that implementation, with atomic accesses optimized to be as relaxed as possible, while maintaining its correctness.

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

3.2 Correctness Criteria

No Data Races A call to an operation on an ADT, as specified in its interface should never produce a data race. 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).

```
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

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 Under RC11 (#reference - RC11 consistency) 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 the paper by (#ref).

For this approach we assume a set of operations, denoted by $Ops = \{o_i\}_1^N$. 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 that are accepted sequential behaviors???. For Example, one possible set of operations for the queue ADT is $\{Enq(1), Deq/1\}$. For this set the Sequential Specifications would be $\{\langle Enq(1), Deq/1\rangle\}$. Consider the following definitions:

Definition 1. Executions

- An *execution* is a partial order over a set of operations.

- 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 such that $e \subseteq to$, and to is included in the ADT's sequential specification. 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... as (failed).
- A *failed execution* is an *execution* that is inconsistent w.r.t. the sequential specifications of the ADT.

Continuing the previous example, for the given set of operations and sequential specifications, $\{\langle Enq(1), deq/1\rangle\}$ is a consistent execution, while $\{\langle Deq/1, Enq(1)\rangle\}$ is a failed execution. Note that since we assume that the underlying memory model is "monotone" (as RC11 is, #ref) 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 identify all *failed executions* for relevant sets of operations on any specific ADT, using SMT solvers to produce the test set. 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 find issues and produce optimized implementations, in their use of atomic accesses, in a way that improves performance, on the modern CPU, while remaining correct under our criteria.

4.1 Test Generation

4.1.1 Defining Test Cases for ADT Consistency

We require the following definitions:

Test sets are defined as follows:

Definition 2. A failed execution f is called a *Minimal Failed Execution* if for every subset $r \subset f$, r is a consistent execution. A test set ...,

Definition 3. *Graph Isomorphism* (*GI*) (article #ref) indicates that two failed execution are semantically equivalent w.r.t. the ADT's specifications. In essence, two graphs G, G' will be considered isomorphic if there is a way to "shuffle" the

edges of one to recreate the other. However, additional requirements are added to represent the specific ADT's representation model. For example, since queue are value agnostic, the following two queue executions are equivalent and will produce isomorphic graphs:
$\{\langle deq(1), enq(2)\rangle, \langle enq(1), deq(2)\rangle\}$, $\{\langle deq(2), enq(1)\rangle, \langle enq(2), deq(1)\rangle\}$[1].

Definition 4. 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 after filtering all isomorphic test cases.

We can now show how to generate test sets for any specific ADT given a set of operations.

4.1.2 Using SMT Solvers to Generate Test Cases

We reduce the problem of finding a test set over a given set operations to an extension of an All-SAT problem. That is, finding all assignments that satisfy a given formula (#ref?). The extension being the added filtering of isomorphic solutions. Each assignment we identify is a test case. Combined, they are the test set. The SMT formula is encoded as follows. Let $Ops = \{o_i\}_1^N$ be our ~~Ops~~ and e an execution. We define the set of boolean variables X_{ij} for every $i, j \in \{1...N\}$~~like so~~: $X_{ij} = true \iff \langle o_i, o_j\rangle \in e$. ~~Using this encoding~~ We define the following traits:

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

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. Will be discussed in greater detail with relevant examples in the following section. For now, we define the black-box $spec(X)$ to determine if X is a *consistent execution* for the given specification.

4. *failed execution* - X is a failed execution if every total order superset of it is a failed execution.
   $failed(X) = po(X) \wedge (\forall Y.(X \subseteq Y \wedge to(Y)) \implies \neg spec(Y))$,
   Where Y is also a partial order and $X \subset Y \equiv \forall i, j. X_{ij} \implies Y_{ij}$.

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*.

[1] 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.

```
1  X := {X_ij | ∀i, j ∈ {1...N}}
2  formula := (minimal(X))
3  res := solver.check(SMT)
4  while (res) {
5       formula = formula ∧ (¬(res) ∧ ¬GI(res))
6       res = solver.check(formula)
7  }
```

Listing 3: SMT solver iterations to find failed execution graphs

6. *Graph Isomorphism (GI)* - formal description of the isomorphism criteria of the specific ADT. Will be discussed in further detail with the relevant examples in the following section. Since this criteria is dependent on previously found solutions it will be added iteratively throughout the run of the SMT solver.

We use items 1-5 to generate the initial formula. We chose to use the Z3 solver (#reference) which does not directly address the All-SAT problem so to solve it we run this formula iteratively through the solver. Each iteration identifies a single test case *tc* which we then add as a constraint to the following iterations thus enumerating all the possible solutions to the formula. To address the isomorphism, we also add an additional constraint $\neg GI(tc)$ on every iteration as described (in pseudo code) in listing 3. Note the parameter N which determines the total amount of method calls. It is determined by the limitations of the solver itself and for Z3 is capped at about 10. Limitations will be discussed at length on section 6. Additionally, the method calls are not equivalent and the different operation types are reflected by the definition of the specification. More on this subject on section 5.

4.2 Generating Testable Code

4.2.1 Generating Test Programs from Test Cases

We suggest the following method as a way to turn failed executions into test programs for a model checker. We show how to to create a simple C test program that describes a specific failed execution. First, in order to achieve a generic, yet simple, method of test program generation, we encapsulate each method call in its own thread. Then, we force order between specific threads and leave others unordered as defined by the failed execution. This order is enforced using atomic boolean variables, with release/acquire memory order accesses. Each thread setting an atomic flag before exiting while other threads can "choose" to wait on it if required. See listing 4 for a detailed template. To complete the test, create a main method that runs all the thread functions and waits for them to complete. Since the tests should not complete under a correct implementation, we add an assert in the end, as described in listing 5.
For example, a sample test program for the simplest failed execution for a queue

```
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, for example:
7      int * res = NULL;
8      bool res = deq(q, &v)
9      // validate success of the operation
10     __VERIFIER_assume(res);
11     // validate expected result
12     __VERIFIER_assume(*v == {{i}}/2 + 1)
13
14     // set the flag
15     atomic_store_explicit(&f_{{i}}, 1, memory_order_release);
16     return NULL;
17 }
```

Listing 4: Template for generated thread code for queues, {{i}} represents the 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 of a specific thread

$(\{\langle deq(1), enq(1)\rangle\})$ is given in Listing 6.

Note the use of pointers as ADT values in listing 4 and 6. This is crucial when testing for data races as described in the previous section. 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 both expects implementations that work with pointers values, while also making sure to dereference these pointers after retrieving objects from the data structure.

4.2.2 Working with GenMC

We use the GenMC model checker (#reference). 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 5 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 10-12 and listing 6 lines 32-33, thus limiting the checker's analysis only to branches of interest for this specific test. As a result, it reduces the amount of branches GenMC has to evaluate dramatically and allows more tests to complete. Occasionally, we also employed this technique on specific parts of the tested implementations themselves. 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 with a total of *N=2 Ops* (and therefore threads) will result in a trivial sequential test, which is not very interesting to run on the model checker. Alternatively, setting *N=10 Ops*, we will produce 8 completely unordered additional threads that on one hand will test the implementation more robustly but on the other create a very large number of branches that have a disabling effect on GenMC. We came up with 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 (#ref), and C the selected thread cap. If

```
1  // ['deq(1)_enq(1)']
2  queue_t q;
3
4  atomic_int f_0;
5  atomic_int f_1;
6
7  // enq(1)
8  void *thread_0(void *arg)
9  {
10     int val_1 = 0;
11     __VERIFIER_assume(atomic_load_explicit(&val_1, memory_order_acquire) == 1)
12
13     int * val = (int *)malloc(sizeof(int));
14     *val = 1;
15     q_enqueue(&q, val);
16
17     atomic_store_explicit(&f_0, 1, memory_order_release);
18
19     return NULL;
20 }
21
22 // deq(1)
23 void *thread_1(void *arg)
24 {
25     set_thread_num(1);
26
27     unsigned int res = 0;
28     bool succ = q_dequeue(&q, &res);
29
30     int * res = NULL;
31     bool succ = q_dequeue(&q, &res, 5);
32     __VERIFIER_assume(succ);
33     __VERIFIER_assume(*res == 1);
34
35     atomic_store_explicit(&f_1, 1, memory_order_release);
36
37     return NULL;
38 }
39
40 int main()
41 {
42     q_init_queue(&q, 2);
43
44     pthread_t t_0;
45     if (pthread_create(&t_0, NULL, thread_0, NULL))
46         abort();
47
48     pthread_t t_1;
49     if (pthread_create(&t_1, NULL, thread_1, NULL))
50         abort();
51
52     pthread_join(t_0, NULL);
53     pthread_join(t_1, NULL);
54     assert(0);
55 }
```

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

```c
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 return value is not validated, unlike in the standard thread function

$K \geq C$, keep all the participating threads in the test. Otherwise, add additional $C - K$ unordered threads. We used to Dilworth's theorem (#ref) to indirectly calculate K by calculating the length of the max antichain in the set. Completing 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 implementation as its complexity has a large impact on GenMC's performance.

4.2.3 Testing with Noise

An alternative to using the original unordered thread functions as discussed in the previous section, is to define configurable noise functions. The noise functions can perform several sequential method calls on the tested implementation and can be used to add more strain and also additional data into the data structure compared to the standard single method call unordered threads, without further increasing the parallelism for the model checker to test. See listing 5 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 implementation[2] 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.
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 found encountered, until such time as 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 (more on that on section 7).

5 Applying The Method to Queues and Sets

In this section we will review how to apply the method described in the previous section to two selected ADTs, the Queue and Set ADTs. For each of these ADTs, we will discuss how to define the sequential specifications and use them to complete the logical formula.

5.1 Queues

Informally, we start by discussing the application of the suggested method to the Queue ADT. To do that we first need to define the following traits for correct sequential queue behavior.

1. It supports two functions:
   (a) Enqueue (denoted *Enq*) - adds an item to the tail of the queue.

[2] supported by the model checker.

(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.

5.1.1 SMT Model for Queues

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 implementation 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 all tested queue implementations. 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[3] 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 in a manner that can be used in the logical formula.

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.
   $\bigwedge_{i \in \{1..k\}, j \in \{N-2k \ldots N\}} (X_{2i-1,j} \iff X_{2i,j})$

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

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

1. Every operation in included in the mapping.
   $\bigwedge_{i\in\{1...N\}} \left(\bigvee_{j\in\{1...N\}} M_{ij}\right)$
2. No two different operations in the source are mapped to the same target operation.
   $\bigwedge_{i,j,k\in\{1...N\}} \left(\neg M_{ik} \vee \neg M_{jk}\right)$
3. An edge in Xcannot be mapped to a non-edge in $X'$and vice versa.
   $\bigwedge_{i,j,k,l\in\{1...N\},i<j,k\neq l} \left(\left(X_{ij} \iff \neg X'_{kl}\right) \implies \left(\neg M_{ik} \vee \neg M_{jl}\right)\right)$

The aforementioned rules are completely generic and apply to any graph. However, considering the specifics of queues and the way we chose to describe them, some solutions will still be considered isomorphic and therefore require additional rules to filter 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 $type(i) \in \{Enq, Deq, Deq\bot\}$ 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:

$$\begin{cases} Deq\bot & i > 2k \\ Enq & i \mod 2 = 1 \\ Deq & i \mod 2 = 0 \end{cases}.$$

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

1. Operations must be mapped to operations of the same type.
   $\bigwedge_{i,j\in\{1...N\},type(i)\neq type(j)} \left(\neg M_{ij}\right)$
2. Mapping must retain the order between Enq and Deq operations of the same value.
   $\bigwedge_{i,j\in\{1...k\}} \left(M_{2i-1,2j-1} \implies M_{2i,2j}\right)$

5.2 Sets

We continue with the application of the suggested method to the Set ADT. Again, we first need to define the following traits for correct sequential set behavior.

1. $Add\left(x\right)$ - adds x to the Set if it does not already exist. Returns true iff the operation successfully added the item to the Set.
2. $Remove\left(x\right)$ - 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.

5.2.1 SMT Model for Sets

We suggest the following Set specific definitions to generate the logical formula to find failing executions. It is plain 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 results with a set with a single entry. Considering this, we see value in testing all operations with every possible return value. This would give us 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 its state. 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: $\{Add\,(x)\,/True, Remove\,(x)\,/True, In\,(x)\,/True, In\,(x)\,/False\}$.

Regarding the values themselves, considering many implementations use some sort for of Search Tree as the basis for the set implementations, using operations with different values in different orders appears to be mandatory to generate relevant interesting test cases. Therefore, the list of operations should include adding and removing at least 3 values. Similarly to how it was defined in 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: $[Add\,(1)\,, Remove\,(1)\,, In\,(1)\,/True, In\,(1)\,/False, Add\,(2)\,, Remove\,(2)\,, In\,(2)\,/True, In\,(2)\,/False, Add\,(3)\,, R$

Sequential Specifications of Sets Given an operation list as described above, the following defines a valid sequential set behavior in a manner 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 $In\,(x)\,/True$ has to occur between add and remove.
   $\bigwedge_{i\in\{1\ldots k\}} (X_{4i-3,4i-1} \wedge X_{4i-1,4i-2})$
3. For any specific value x, if $In\,(x)\,/False$ has to occur either before x was added or after it was removed.
   $\bigwedge_{i\in\{1\ldots k\}} (X_{4i,4i-3} \vee X_{4i-2,4i})$

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
$\{\langle Add\,(1)\,/True, Remove\,(2)\,/True\rangle, \langle Add\,(1)\,/True, Remove\,(3)\,/True\rangle, \langle Remove\,(1)\,/True, Add\,(2)\,/True\rangle\}$
$\{\langle Add\,(2)\,/True, Remove\,(1)\,/True\rangle, \langle Add\,(2)\,/True, Remove\,(3)\,/True\rangle, \langle Remove\,(2)\,/True, Add\,(1)\,/True\rangle\}$
Could 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 to 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 (#ref). Therefore, we will first review the test sets generated for each ADT. We then, review 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

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

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 worth noting when examining these sets:

1. The generated sets are indeed minimal. 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 brought here for the second set.
3. The execution reproducing the issue found in the original implementation of the HW queue as presented in Part 2 is found as execution #4 of the first set and #15 of the second.
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 minimal set. This was corroborated by running the SMT solver with $N = 9, k = 4$ and obtaining the same result set.

6.1.2 From Test Cases to Test Programs

We used the process of generating a suite of test programs from a set of test cases as described in section 4.1.3 directly to the generate two test suites from the aforementioned test cases of for the Queue ADT. For our testing, we generated 4 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 any specific values.

6.1.3 Testing And Optimizing Selected Queue Implementations

We tested our 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 finding regarding these queues, followed by a short description of the work performed on each implementation. Note that due to GenMC limitations, some tests could not complete on some implementations. We address that 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 suites 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 of the first correctness criteria (i.e. data races) in the original implementation.
- Type II - whether we identified issues of 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.

Note that 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 (3 acq, 2 rlx, 2 rel, 1 rel/acq)
Fast-MPMC	Completed	14m	No	No	8 (2 rlx, 2 acq, 2 rel, 2 rlx/rlx)	9 (5 rlx, 2 rlx/rlx, 1 acq, 1 rel)
HW	1 test t/o with $C = 4$ 4 tests t/o with $C = 5$	6h 15m	No	Yes	4 (2 rel, 1 acq, 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 acq, 2 rel, 2 rlx, 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/rel)	15 (3 rlx, 7 acq, 5 rel/acq)
QU	1 test t/o in both suites	3h	No	No	12 (4 rel, 3 acq, 3rlx, 1 rlx/rel, 1 rlx/rlx)	12 (9 rlx, 1 acq, 1 rel/acq, 1 rlx/rlx)
UNIQ	Completed	40m	Yes	No	8 (All SC)	13 (9 rlx, 2 rlx/rlx, 1 acq, 1 rel)

Chase-Lev Queue Based on (#ref), original implementation 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 };
```