Framework for Analyzing Highly Concurrent Algorithms In SPIN

Atefeh Maleki
Abstract

Framework for Analyzing Highly Concurrent Algorithms In SPIN

Atefeh Maleki

Human beings have gradually become dependent on computers, and more specifically, the software that they are working with. Most skillful programmers are working hard and spend much time testing or debugging their programs in order to make their software reliable.

Verification techniques have been developed to find bugs as much as possible in the early stages of development or implementation of the software. Model checking is one of the verification techniques which requires a simplified model of the system. It will test automatically whether this model accommodates the given system’s specification or not. One of the most successful tools for automatic verification is the SPIN Model Checker.

It is a general tool for verifying the correctness of distributed software models, parallel algorithms and data structures. The input language of SPIN is Promela. Promela does not support some features such as pointer declarations, dynamic memory allocation, or garbage collection in order to be a simple modeling language and decrease the size of the state space.

This master thesis presents a framework in Promela for verifying safety and liveness properties of highly concurrent algorithms in the SPIN model checker. We present some macros in Promela which are independent from the specific behavior of a particular algorithm. These macros can be used in most concurrent algorithms which are verified by SPIN. In this work also we used some techniques for memory management. Some macros are created which do garbage collection and dynamic memory allocation themselves in our framework in a way that the behavior of the original algorithm is preserved maximally. We have applied our approach to some concurrent deque algorithms and evaluated the results.
Acknowledgements

I am in debt to my respectful supervisor, Professor Bengt Jonsson. I appreciate his patience. Without his suggestions I would not be able to finish this work.
I would like to thank Dr. Gerard Holzmann, the developer of Spin Model Checker, who answered my questions by emails all along my thesis.
I would also like to thank my thesis reviewer, Professor Parosh Abdulla, who approved this thesis project. Many thanks to my friends and colleagues in UPMARC group, especially Carl Leonardsson for his helpful comments on my thesis report.

I wish to express my appreciations to best parents in the world for their love and generous support during my whole life. I love them and wish them health and happiness.
I would like to dedicate this thesis to them.

In the end, I can not forget to thank my boy friend Majid for his warm support and encouragement during my thesis work.
Contents

Acknowledgements iii

List of Figures vii

List of Tables viii

1 Introduction 1
  1.1 Motivation .................................................. 2
  1.2 Goals And Expected Results .............................. 2
  1.3 Structure of the thesis .................................... 3

2 Background 4
  2.1 Verification .................................................. 4
    2.1.1 Testing ............................................... 5
    2.1.2 Model Checking .................................... 5
  2.2 The Model Checker SPIN ................................. 6
  2.3 Concurrent Algorithms and data structures .......... 7
  2.4 Linearizability ............................................ 7
  2.5 Memory Management ...................................... 8
    2.5.1 Dynamic Memory Allocation .......................... 8
    2.5.2 Garbage Collection .................................. 9

3 Sample Problem 10
  3.1 Simple Example ............................................ 10

4 Implementation 13
  4.1 Heap modeling ............................................. 13
    4.1.1 Dynamic Memory Allocation .......................... 13
    4.1.2 Pointer Translation .................................. 14
    4.1.3 Garbage Collection Modeling ....................... 14
  4.2 Linearization point checking ................................ 16

5 Examples 18
  5.1 Model Checking of examples .............................. 18
    5.1.1 Snark Algorithm ..................................... 18
    5.1.1.1 Algorithm Code ................................... 19
List of Figures

3.1 Treiber's non-blocking stack algorithm ............................... 11
4.1 Create NewNode Macro ................................................. 14
4.2 Garbage Collection Macros ............................................ 15
4.3 Linearization Checking Macros ...................................... 16
5.1 Snark Bug .................................................................... 22
5.2 Second Snark Bug .......................................................... 24
5.3 Second Scenario With GC ................................................ 25
6.1 verification result with GC ............................................... 32
6.2 Counting with interference .............................................. 33
6.3 Counting with interference simulation .............................. 34
6.4 Counting with interference verification .............................. 34
6.5 Counting with interference verification of abstract data ......... 34
List of Tables

5.1 Verification Output of Snark ........................................... 23
5.2 Verification Output of DCAS-BASED concurrent deques ............ 29
To my merciful parents...
Chapter 1

Introduction

Verification and validation are the processes of checking a product, service, or system to meet its specifications and fulfill its intended purpose. Model-based techniques for verification and validation of reactive systems, such as model checking have witnessed drastic advances in the last decades. These techniques require a formal model which specifies the intended behavior of the system or component. Ideally, such a model is generated during the specification and design phase of software life cycle.

Model checking [1] enables specifying requirements and correctness properties of a system, and importantly to check if the system automatically satisfies them. One of the advantages of model checking is handling parallel systems. Many parallel systems use data structures such as lists, stacks or hash tables which are concurrent. It means that several threads are allowed to access and manipulate the data storage simultaneously. Historically, such data structures were used on uniprocessor machines with operating systems that supported multiple computing threads (or processes).

One of the most successful tools for automatic verification is the SPIN Model Checker (Simple Promela Interpreter) [2], a general tool for verifying the correctness of distributed software models, parallel algorithms and data structures in a rigorous and mostly automated fashion. A large number of tools and formalisms have been proposed for Model checking such as CHESS model checker [3], UPPAAL [4] and TLA+ [5], but we chose SPIN because it is efficient and widely used not only in academia but also in industry. SPIN also offers a large number of options to further speed up the model-checking process and save memory, such as partial order reduction, state compression, bitstate hashing, and weak fairness enforcement.

In this thesis we have focused on modeling concurrent data structures in SPIN to create a framework that is adaptable to most parallel algorithms. This section contains a brief introduction on the topic, goals and motivation toward completion of this thesis.
1.1 Motivation

Most critical software applications execute in a multithreaded environment, with numerous external dependencies: they are concurrent. Fortunately, today powerful tools are available to verify the logical correctness of concurrent (distributed, parallel, or multithreaded) programs.

SPIN is one of the world’s most popular model checkers, and arguably one of the world’s most powerful tools for detecting software defects in concurrent system designs [6]. However, it cannot check the source code of programs directly. It is necessary to model them in Promela as an input language to utilize this powerful tool. This motivated us to generate a Promela framework for model checking of concurrent algorithms in SPIN. This framework definitely will help us to save time during modeling of algorithms in Promela. We will explain more in the next chapters how we do this and the results of the work.

1.2 Goals And Expected Results

The main goal of this thesis is generating a framework for analyzing highly concurrent algorithms in SPIN. This idea came from professor Bengt Jonsson during a verification course, when he found that students spent more time on modeling the algorithm instead of verifying it. So, the purpose of the whole project is to construct a framework in Promela for verifying safety and liveness properties of concurrent algorithms in SPIN, in shorter time with efficient memory usage. We decided to produce some macros in Promela which are independent from the specific behavior of a particular algorithm. These macros can be used in most concurrent algorithms which are to be verified in the SPIN model checker. Promela is not a powerful language, and does not support some features such as pointer declarations, dynamic memory allocation, or garbage collection. So, this thesis presents some macros to detect bugs of concurrent algorithms by translating them in Promela with some memory management techniques in a way such that the behavior of the original algorithms are preserved maximally. We put some effort to find how the models can be optimized to have less state spaces and memory usage and then evaluate the number of states which will be created after using these methods.

As the result of the project, the following objects should be met:

- creating of macros in Promela independent from algorithm, that can be used in verification of other algorithms by SPIN,
finding sensible approaches to translate untranslatable aspects of languages in Promela, such as pointers and memory management,

- clean and clear Promela code output with nice formatting,

- it should be possible to simulate and verify the translated Promela codes in SPIN.

1.3 Structure of the thesis

This section outlines the general structure of this report and gives a brief summary of its chapters.

Chapter 2 describes related background information about principles of my work and basic knowledge about verification and modeling that is needed for understanding the rest of the report. The readers who have general knowledge about these areas can skip this chapter. A sample algorithm will be introduced in Chapter 3 and we will discuss why our goals are interesting on that algorithm. Chapter 4 describes our implementation. You can see the macros and framework we created in Promela. Chapter 5 is our examples and the results gained by applying the approach. The last several chapters do evaluations, make conclusions and describe possible future works that can be done as complementary of this work.
Chapter 2

Background

2.1 Verification

Software may have bugs, i.e., an error, flaw, mistake, failure or fault in a computer program or system that produces an incorrect or unexpected result or causes it to behave in unintended ways. Most bugs arise from mistakes and errors made by programmers in source code, and a few are caused by compilers producing incorrect code. Better development processes, enhanced knowledge, effective collaborations and verification techniques may reduce the amount of mistakes and increase the reliability of software.

The correctness of a software system is checked by two processes. One of the processes is to check if the software is what the customer wants. This process is called validation. The other process is to check if the software is bug free and it matches the specification of the software. This process is called verification.

In verification, one aim is to prove or disprove the correctness of a program or system with respect to its intended behavior. Verification techniques are developing to find bugs as much as possible in early stages of development or implementation, before running software on the system and check if the system is built right or not. It is very helpful to find if a system conforms to its correctness properties.

\begin{align*}
\text{Verification} &= \text{building the system right} \\
\text{System description} \models \text{Correctness properties}
\end{align*}

There are several verification methods, and most of them are useful, but some have more advantages. One approach to verification is to manually inspect the code of the
software. But this approach is becoming more difficult and time consuming as more and more complex systems are being developed. “Prototyping and simulation” is one technique that can be used on design level but, manual selection of exhaustive test cases and inputs is difficult and needs lots of work. “Code and design review” is another one. It is good at finding (some classes of) problems but needs organization and people. Static program analysis is one approach to analyze the source code by tools. It is completely automatic but can verify a limited set of properties and unfortunately tools are available only for some languages and properties. Formal verification is an alternative method for verifying a system. It assumes access to a so called specification of the software, i.e., a description of the correct behavior of the system. The method compares the specification to the actual behavior of the system [7]. Testing and model checking are two widely used and famous methods for verification.

2.1.1 Testing

In testing, a so called test case is generated, which is an input to the software and the expected response (output) from the software, according to the specification. The input is fed to the software and if the output is as expected, we say the system has passed the test case, otherwise it has failed. By feeding a set of these test cases to a software, different properties of the specification can be tested. Testing is the most practical technique that can verify a wide range of properties but can only be used on implementations. It is difficult to make it exhaustive and hard to make it reproducible for concurrent/distributed programs.

2.1.2 Model Checking

One of the verification techniques on which we focus in this thesis is Model checking. When the software itself cannot be verified exhaustively, we can build a simplified model of the underlying design that preserves its essential characteristics but, that avoids known sources of complexity [1]. In model checking first we analyze a prototype or model by tools, then understand what properties of the model should be verified (safety and liveness), how important they are in practice and to what extent these properties can be verified automatically by the model checker.

In other words, in model checking the specification of a system is algorithmically checked against a model of the system which describes the system behavior. The model is usually expressed as a directed graph consisting of nodes and edges. The nodes represent the states of the program and the edges represent the possible executions which change the program states. Usually, a set of properties is associated with each node. The properties
represent the conditions that should hold in executions of the program. Model checking is important for both validation and verification of a software system. The model of a system can be compared to customers needs for validating the system. Model checking can be done early in the design cycle, e.g., on design level. The problem is that model checking does not scale to very large models. A model must be constructed (at a suitable level of abstraction), and it must be maintained when the system evolves.

2.2 The Model Checker SPIN

SPIN is a model checker developed by Gerard J. Holzmann. It has since become widely used in industries that build critical and concurrent systems [8]. It is a generic verification system that supports the design and verification of systems of asynchronous processes. It will focus on proving the correctness of processes interactions [1] and we use it as one model checking tool for verification of concurrent algorithms in this thesis.

To check correctness properties of an algorithm in SPIN, a model must be written in Promela that describes the behavior of the system, then correctness properties that express requirements of the system’s behavior are specified. Such properties include absence of deadlocks, run time errors, memory leaks. Finally, the model checker checks if the correctness properties hold for the model and if not, provides a counterexample. A counter example is a computation that does not satisfy a correctness property.

SPIN’s verification procedure is based on an optimized depth-first graph traversal method. The set of nodes that have been visited on a path in a depth-first search are stored in a stack, so by checking the stack it is possible to determine that the state has already been visited and needs not be searched again. The cycle detection method used in SPIN is required to be compatible with all modes of verification, including exhaustive search, bit-state hashing, and partial order reduction techniques. SPIN uses a partial order reduction method to reduce the number of reachable states that must be explored to complete a verification. This avoids creating states that cannot be affected by interleaving the execution for the processes. The size of the interleaving product that SPIN computes can, in the worst case, grow exponentially with the number of processes. Given the size of product, we can place upper bounds on the amount of memory and time that would be required to complete various types of verification tasks but it seems that SPIN has problems with handling a large number of threads, due to the many possible interleavings, therefore work on optimization is needed to make the approach scale to a large number of threads [9].

We used SPIN to check correctness properties and linearizability of non-blocking concurrent data structure algorithms that manipulate dynamically allocated memory.
2.3 Concurrent Algorithms and data structures

Parallel algorithms are algorithms designed to run efficiently on a parallel computer. A parallel algorithm may involve a greater number of arithmetic operations than a serial counterpart. It is designed in a way such that many arithmetic operations are independent and can be performed in parallel, i.e., simultaneously. Parallel algorithms are valuable because of substantial improvements in multiprocessing systems and the rise of multicore processors. Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs.

In concurrent algorithms we may have global data structures which are a particular way of storing and organizing data for access by multiple computing threads (or processes) on a computer. These data structures are named concurrent data structures. Concurrent data structures, intended for use in parallel or distributed computing environments, differ from "sequential" data structures, intended for use on a uniprocessor machine, in several ways. Concurrent data structures are significantly more difficult to design and to verify as being correct than their sequential counterparts [10].

2.4 Linearizability

Concurrent data structure algorithms are becoming increasingly popular as they provide mechanisms for achieving high performance on multicore hardware. Typically, to achieve high performance, these algorithms use fine grained synchronization techniques. This leads to complex interaction between processes that concurrently execute operations on the data structure. Such interaction presents serious challenges both for the construction of an algorithm and for its verification.

Linearizability [11] is a widely accepted correctness criterion for implementation of concurrent data structures. It guarantees that a concurrent data structure appears to the programmer as a sequential data structure. Intuitively, linearizability provides the illusion that any operation performed on a concurrent data structure takes effect instantaneously at some point between its invocation and its response. Such points are commonly referred to as linearization points. Informally, a concurrent object is linearizable if each concurrent execution of operations on the object is equivalent to some permitted sequential executions [12, 13].

Other correctness criteria in the literature, such as sequential consistency, also require that a concurrent execution be equivalent to some sequential executions. However, these criteria differ on the requirements on ordering of operations. Sequential consistency requires that operations in the sequential execution appear in an order that is consistent with the order seen at individual threads. Compared to these correctness
criteria, linearizability is more intuitive, as it preserves the real-time ordering of non overlapping operations.

There could be many possible linearizations of a concurrent execution. Finding a single linearization is enough to declare the concurrent execution correct. The linearization point for an operation is usually a statement in the code of the operation (fixed linearization point). For more complex fine grained algorithms, a linearization point may reside in methods other than the executing operation and may depend on the actual concurrent execution (nonfixed linearization point).

Checking linearizability for concurrent objects that are implemented by a dynamically allocated linked data structure is challenging, because it requires correlating every concurrent execution with a corresponding permitted sequential execution (linearization) and it requires correlating executions that may manipulate memory states of unbounded size [14]. We also in this work tried checking linearizability of some concurrent algorithms. We worked first on how we can implement the linearization checking in Promela code for our model, then decided to create macros for our model that can check linearization points of some concurrent linked list data structures. These macros can be applied in linearizability checking of other models in SPIN.

2.5 Memory Management

Many highly concurrent optimistic algorithms [15] assume dynamic memory allocation and garbage collection (the memory is only managed by the garbage collector, not via manual memory management) but these features are not provided by SPIN.

2.5.1 Dynamic Memory Allocation

In computer science, dynamic memory allocation (also known as heap-based memory allocation) is the allocation of memory storage for use in a computer program during the runtime of that program. Dynamically allocated memory exists until it is released either explicitly by the programmer, or by the garbage collector. This is in contrast to static memory allocation, which has a fixed duration. SPIN does not support dynamically allocated memory (e.g., creating new objects, pointer dereferences), so we decided to implement dynamic memory allocation as one part of the input Promela model.
2.5.2 Garbage Collection

Garbage collection (GC) in a verification tool is crucial for verifying important concurrent algorithms. Garbage collection is a wildly used technique in actual programming environments, helping programmers to avoid memory fragmentation and invalid referencing problems.

In order to efficiently model check programs that use garbage collection, similar functionalities have to be embedded in model checkers [16]. If no garbage collection is performed during model checking, we have some new state to be added to the program’s state space. So when the state space is explored in the depth-first order, this leads to exhaustion of the system resources before any useful result is shown and the algorithm may leak an unbounded amount of memory.

We implemented GC also as the other part of our model and compare the results of verification with support of GC and without. For the comparison results see evaluation chapter.
Chapter 3

Sample Problem

Shared data structures which can be accessed concurrently by several processes are extremely tricky and can easily cause serious problems if used improperly. Most famous are concurrent queues and stacks which are widely used in both academic and industrial research for implementing parallel applications and systems. There are lots of concurrent algorithms which use concurrent data structures.

3.1 Simple Example

A Simple Example is Treibers Stack. Figure 3.1 contains C-like pseudocode for Treibers stack [17], one of the simplest non-blocking concurrent algorithms. The stack is represented as a singly linked list, which is updated using CAS (compare and swap). Compare and swap is a special instruction that atomically compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value. This guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail. The result of the operation must indicate whether it performed the substitution. This can be done either with a simple boolean response (this variant is often called compare-and-set), or by returning the value read from the memory location (not the value written to it).

Viktor Vafeiadis [17] gave a scenario and explained that this algorithm leaks memory, i.e., we cannot free popped nodes and it is possible to remove two nodes from the stack instead of one. He presented the ABA problem of the algorithm while verifying linearizability, given a specified set of linearization points. In multithreaded computing, the ABA problem occurs during synchronization. When a location is read twice, it has
the same value for both reads, this value used to indicate nothing has changed. However, another thread can execute between the two reads and change the value, do other work, then change the value back, thus fooling the first thread in to thinking nothing has changed even though the second thread did work that violates that assumption. The ABA problem occurs when multiple threads (or processes) are accessing to shared memory interleavingly.

The linearization points of Treiber are conditional: @1 and @3 are linearization points if and only if the respective CAS succeeds; @2 is a linearization point if and only if the value stored to $t$ is NULL (@2 is the linearization point of a failed pop operation: at this point we know that the stack is empty). To carry out the verification, we expect the programmer to annotate these points with auxiliary code asserting that they are linearization points.

We explained in detail what SPIN is in Chapter 2. To verify Treiber’s algorithm we can use SPIN. We need to have a Promela model of Treiber first. So, our aim is to have an efficient Promela model with some methods to find if the algorithm is linearizable or not. We tried to add some assertions in our model to look after some points which are tricky and named linearization points. SPIN will go through all possible states in which those assertions are violated.

We found that in algorithms which use CAS or DCAS (double compare and swap) operations, the probability of the ABA problem occurring is higher than in others. So, to
verify these algorithms and data structures like queues and stacks by SPIN, we need to have one sequential set which imitates the behavior of concurrent data structure. This motivates us to create a macro which can be used in all Promela models. Then we go more into details and think about algorithms which use dynamic memory allocation and garbage collection. All of these issues motivate us to create a framework for analyzing concurrent data structures in SPIN.

We chose two more algorithms that both implemented a deque (double ended queue) and verify them with the help of our framework. You can find more explanations in Chapter 5.
Chapter 4

Implementation

Since Promela is a specification language, rather than a conventional programming language, it lacks the ability to model common programming language mechanism, such as dynamic memory management, garbage collection and function call-return. The reason for limiting the modeling power is to alleviate the state explosion problem [18].

We have developed some macros and inlines in Promela based on the principles we explained in Chapter 2 to deal with these restrictions. In this chapter, first, we describe the principles behind these inlines and macros, then, we show our Promela code in some figures. We have tried to have a mostly straight-forward implementation model of the algorithm.

4.1 Heap modeling

4.1.1 Dynamic Memory Allocation

Promela does not support dynamically allocated data structures or pointers, hence we must emulate them. Since the memory can be seen as a large sequential array containing all program data, one option would be to define a global array, indexed by addresses, that simulates all memory [6]. This option obviously is too space-consuming for analysis by model checking. A less costly approach is to allocate arrays which hold those objects that are dynamically allocated by the program. We allocate an array, and the pointers to this array will be translated to indices in the array. With this approach, we obviously cannot handle pointer arithmetic. Since the array for a data type must be allocated statically at the beginning of program execution, we must also record which cells are allocated and which are not. This can be done by a special unallocated value, or by a separate array of booleans.
4.1.2 Pointer Translation

Assignments to pointers are translated to normal integer assignments. New objects will be allocated to the first available position in the corresponding array (by convention named : used[MemSize]).

We created a newNode macro. It allocates the next available node in the array which models the heap. Look at Figure 4.1. In order to reduce the state space, nodes are always allocated with increasing array indices. In implementation without garbage collection, nodes are never reclaimed. It means that there is a maximum number of nodes that can ever be allocated.

```c
#define newNode (nd)
atomic {
    node_c = 0;
    do {
        if (node_c >= MemSize) \ break
        else {
            if (used[node_c] == 0)
            used[node_c] = 1;
            set_ptr(nd, node_c);
            \ break
        }
    } while (node_c++);
    node_c = 0
};
```

Figure 4.1: Memory Allocation

4.1.3 Garbage Collection Modeling

As our sample problems, we used algorithms that require the presence of a garbage collector (GC). It means, the memory (the nodes of the linked list) is only managed by the garbage collector and not via manual memory management. Without garbage collection, the algorithm may leak an unbounded amount of memory due to the ABA problem. SPIN does not provide garbage collection support. Hence, we have defined a garbage collector in Promela ourself. We have concluded that the garbage collector should run on every pointer update, effectively leading us to implement a reference counting algorithm. Each node now contains an additional field, the RC field, which is modified on pointer update. Once the field reaches 0, 1, or 2 under some conditions, the collector runs atomically and collects the object. Once the object is collected, it is important to clear all of the node fields in order to avoid creating distinct states that differ only in those unused object fields. To use this reference counting approach, our models are augmented with the relevant operations on every pointer update statement. This requires careful additional work. We defined a macro which name is setp_tr that
not only sets the value of a pointer to a new value but also decrements and increments the RC field of related pointers. The size of the RC array is also the same size as the array which models memory and the content of each index shows the number of references which point to that cell in the memory array.

One of the shortcomings of SPIN is not having recursive functions, so to have recursive garbage collection we have to find a new way, because collecting one object and clearing all its related values may create new garbage. For this purpose, we defined a new array with the same size of the memory array (toBeCollected[MemSize]). In the garbageCondition macro, under some conditions, we will check if garbage is produced or not. If there is garbage, it will be added to the toBeCollected array, and the garbage collector will run until this array becomes empty. So, during the collection process if new garbage is generated, it will be collected also by the garbage collector, see Figure 4.2.

```c
#define garbageGenerated (ptr)
atomic{
  IF : (RC[ptr] == 0) -> toBeCollected[ptr] = 1; gc();
  :: ((RC[ptr] == 1) && ((node_mem[ptr].R == ptr) || (node_mem[ptr].L == ptr))) ->
     toBeCollected[ptr] = 1; gc();
  :: ((RC[ptr] == 2)) && ((node_mem[ptr].R == ptr) && (node_mem[ptr].L == ptr))) ->
     toBeCollected[ptr] = 1; gc();
FI
};

#define check_ifGarbageGeneratedAgain (ptr)
atomic{
  IF : (RC[ptr] == 0) -> toBeCollected[ptr] = 1;
    newGarbageGenerated = 1;
  :: ((RC[ptr] == 1) && ((node_mem[ptr].R == ptr) || (node_mem[ptr].L == ptr))) ->
    toBeCollected[ptr] = 1;
    newGarbageGenerated = 1;
  :: ((RC[ptr] == 2)) && ((node_mem[ptr].R == ptr) && (node_mem[ptr].L == ptr))) ->
    toBeCollected[ptr] = 1;
    newGarbageGenerated = 1;
FI
};

#define gc()
atomic{
  FOR(i, 0, MemSize-1)
    IF : toBeCollected[i] == 1 ->
      used[i] = 0;
      RC[node_mem[i].R]--;
      check_ifGarbageGeneratedAgain[node_mem[i].R];
      node_mem[i].R = undef;
      RC[node_mem[i].L]--;
      check_ifGarbageGeneratedAgain[node_mem[i].L];
      node_mem[i].L = undef;
      toBeCollected[i] = 0;
    IF : newGarbageGenerated-> i = 0;
    FI
  ROF(i, 0, MemSize-1);
  i = 0;
  newGarbageGenerated = 0;
};
```

Figure 4.2: Garbage Collection processes in Promela
4.2 Linearization point checking

We checked linearizability of both sample problems in this thesis. This encompasses checking that the algorithms preserve structural properties of the data structures, like being sorted and acyclic. If the algorithms would exhibit a behavior which is not equivalent to a correct sequential implementation, then they are not linearizable. I roughly

```
inline seq_pushR(v)
    atomic{
        IF : (endpointer < SeqSize) \rightarrow Seqset[endpointer] = v;
            endpointer++;
    FI
}
```

```
inline seq_popR(resultSeq)
    atomic{
        assert (endpointer != 0); resultSeq = Seqset[endpointer - 1];
            Seqset[endpointer - 1] = 0; endpointer--;
    }
```

```
inline seq_pushL(v)
    atomic{
        i = endpointer;
        IF : (endpointer < SeqSize) \rightarrow
d o i > 0 \rightarrow Seqset[i] = Seqset[i-1]; i--;
            i == 0 \rightarrow break;
            od;
            endpointer++; Seqset[0] = v;
    FI
i = 0;
}
```

```
inline seq_popL(resultSeq)
    atomic{
        assert (endpointer != 0);
        assert(Seqset[0] != 0);
            resultSeq = Seqset[0];
            Seqset[0] = 0;
        FOR(i, 0, endpointer - 2)
            Seqset[i] = Seqset[i+1];
        ROF(i, 0, endpointer - 2);
            i = 0;
            endpointer--;
            Seqset[endpointer] = 0;
    }
```

```
inline seq_popCheckIfEmpty(was_empty)
    atomic{
        if : endpointer == 0 \rightarrow was_empty = true;
            else \rightarrow was_empty = false;
    FI
}
```

![Figure 4.3: Linearization points checking by Sequential set](image)

followed the method described in [12] to handle fixed linearization points (the points are explicitly defined whit statements in the code of the algorithm by user). The main approach is to build a linearization on the fly by keeping track of a serial set implementation (via an array which keeps the states of the corresponding sequential implementation). At each linearization point, this serial set is simply updated (and checked) directly.
We instrumented the model by a sequential version of the concurrent data structure \( \text{Seqset}[\text{SeqSize}] \), which is maintained independently of the concurrent version in the same state. When we encounter a linearization point during model checking of the concurrent execution, we execute the corresponding sequential operation on the sequential version of the data-structure, and record its result. When the concurrent operation completes, we compare its result to the (recorded) result of the sequential operation.

\textbf{Endpoint} is a pointer that points to the first empty place in the sequential set and it will be updated in each iteration of pushing or popping from deque endpointer place. The sequential set exactly shows the sequential behavior of the threads, so all the elements are in the correct place in this set. Figure 4.3 shows linearization checking macros in Promela.
Chapter 5

Examples

To evaluate our approach we chose two algorithms which implement a double-ended queue (deque) using DCAS operation. A deque is a data structure that maintains a finite sequence of items and supports insertion or removal of an item at either end of the sequence [19]. In this chapter we explain pseudo codes of those algorithms and describe the verification and simulation results of them by our framework in the SPIN model checker. First we analyze the Snark algorithm and try to find its bugs by SPIN, then we use our framework also for another DCAS-based concurrent deque and discuss the benefits of using our framework.

5.1 Model Checking of examples

5.1.1 Snark Algorithm

The algorithm that we chose for the first experiment is Snark, a concurrent deque developed by David L. Detlefs et al [20]. It represents a deque as a doubly-linked list of nodes. Each node in the list contains two pointer fields, R and L and a value V. There are two global anchor variables, arbitrarily called LeftHat and RightHat, which generally point to the leftmost node and the rightmost node in the chain.

The right side push operation first obtains a fresh node structure from the storage allocator. If allocatable storage has been completely exhausted, the new operation will yield a null pointer. The push operation treats this as sufficient cause to report that the deque is full. Otherwise, the R field of the new node is made to point to Dummy and the value to be pushed is stored into the V field. A while true loop is used to iterate until this new node is added into the doubly-linked list. The RightHat is copied into local variable rh. If rh points to a right dead node, then the deque is empty. In this case,
the new node should become the only node in the deque. It’s L field is made to point to
\texttt{Dummy} and then a DCAS is used to atomically make both \texttt{RightHat} and \texttt{LeftHat} point to
the new node. If this DCAS succeeds, then the push has succeeded. If the DCAS fails, then
control will go around the while true loop to retry. If the deque is not empty, then the
new node must be added to the right hand end of the doubly linked chain. The copied
content of the \texttt{RightHat} is stored into the L field of the new node and then a DCAS is
used to make both the \texttt{RightHat} and the former right end node point to the new node,
which thus becomes the new right end node. If this DCAS operation succeeds, then the
push has succeeded, if the DCAS fails, then control will go around the while true loop
to retry. The right side pop operation also uses a while true loop to iterate until an
attempt to pop succeeds. You can find its code in section 5.1.1.1.

Assume that some push and pop operations are running in several independent
processes sharing the same memory, inserting and removing elements from the deque
concurrently. There might be problems when two pop functions are applied to a deque
with only one element. We will verify correctness properties of Snark using several test
harnesses. These test harnesses with the help of SPIN will check if the algorithm reports
that the deque is empty because of the wrong empty deque checking. Also it will check
if two pop functions can return the same value without any error being raised.

Our time is mostly used to model the algorithm in an efficient way such that SPIN
can detect these kind of bugs in the shortest path.

\subsection{Algorithm Code}

```java
boolean DCAS(val *addr1, val *addr2, val old1, val old2, val new1, val new2)
{
    atomically {
        if ((*addr1 == old1) && (*addr2 == old2)) {
            *addr1 = new1;
            *addr2 = new2;
            return true;
        }
        else return false;
    }
}

val pushRight(val v) {
    nd = new Node(); /* Allocate new Node structure */
    if (nd == null) return "full";
    nd->R = Dummy;
    nd->V = v;
    while (true) {
        rh = RightHat;
        rhR = rh->R;
        if (rhR == rh) {
            nd->L = Dummy;
            lh = LeftHat;
            if (DCAS(&RightHat, &LeftHat, rh, lh, nd, nd))
                return "okay";
        }
        else {
            nd->L = rh;
            if (DCAS(&RightHat, &rh->R, rh, rhR, nd, nd))
                return "okay";
        }
    }
}
```
val popRight() {
    while (true) {
        rh = RightHat;
        lh = LeftHat;
        if (rh->R == rh) return "empty";
        if (rh == lh) {
            if (DCAS(&RightHat, &LeftHat, rh, lh, Dummy, Dummy))
                return rh->V;
        } else {
            rhL = rh->L;
            if (DCAS(&RightHat, &rh->L, rh, rhL, rhL, rh)) {
                result = rh->V;
                rh->R = Dummy;
                rh->V = null;
                return result;
            }
        }
    }
}

val pushLeft(val v) {
    nd = new Node();
    if (nd == null) return "full";
    nd->L = Dummy;
    nd->V = v;
    while (true) {
        lh = LeftHat;
        lhL = lh->L;
        if (lhL == lh) {
            nd->R = Dummy;
            rh = RightHat;
            if (DCAS(&LeftHat, &RightHat, lh, rh, nd, nd))
                return "okay";
        } else {
            lhR = lh->R;
            if (DCAS(&LeftHat, &lh->R, lh, lhR, lhR, lh))
                return "okay";
        }
    }
}

val popLeft() {
    while (true) {
        lh = LeftHat;
        rh = RightHat;
        if (lh->L == lh) return "empty";
        if (lh == rh) {
            if (DCAS(&LeftHat, &RightHat, lh, rh, Dummy, Dummy))
                return lh->V;
        } else {
            lhR = lh->R;
            if (DCAS(&LeftHat, &lh->R, lh, lhR, lhR, lh)) {
                result = lh->V;
                lh->L = Dummy;
                lh->V = null;
                return result;
            }
        }
    }
}

Simon Doherty et al [21] showed that the Snark algorithm is not bug free but, according to our discussion with him neither of those bugs were found using Promela/SPIN. The
first bug was found almost immediately by Mark Moir, after the algorithm was explained to him. Simon Doherty found the second bug while he was attempting to verify the algorithm using simulation and theorem proving. This took several weeks to prove. As an experiment, he built a Promela model of the original Snark and verified it using SPIN. He was able to find those bugs almost immediately because he added some assertions into the model, and SPIN found a counterexample in milliseconds.

We decided to start with Snark and implement its model in Promela, then step by step extract some macros which are independent of the model and can be used in verification of other models. We added some assertions into the proctypes to capture supposed invariants of the algorithm. We found executions that broke some invariants in a few seconds.

5.1.1.2 Verification and Simulation Results

The Promela model of the Snark algorithm is available in Appendix A.1. You can find the results of verification and simulation graph in this chapter. All the verification was done using 3GB of memory in the Promela model. Both issues were found without using garbage collection in the Promela model, i.e., allocated nodes were never reclaimed. Two bugs were found. Both of them cause pop operations to behave incorrectly. So they can be fixed by modifying the pop operations [21]. The push operations are left untouched.

- First Bug We simulated our model using 4 processes, two Push and two Pop. In less than 7 seconds SPIN found a bug and gave me a guided simulation. A Pop operation can return empty even if the deque is never empty during pop’s execution. This is a violation of linearizability. The sequential deque (Seqset) helps to discover the scenario which leads to this bug. A deque has four major operations, pop from right and left, push into right and left. The same operations will be done on the sequential (Seqset) deque also. Linearization points of these functions are the points right after, they finish their intended job, i.e., pushing or poping. In that point (after the DCAS operations in our algorithms) we atomically call the corresponding sequential operation. This operation will change the data of the Seqset deque. Then we will compare the final version of the Seqset deque with the concurrent one by using an assertion. The size of Seqset deque is MemSize-1 because it is not necessary to have a cell for Dummy node. The Dummy node which is created by the first run of the model will remain during the whole execution.

Following is the scenario which leads to this bug:
A process p invokes popRight while the deque is not empty. It loads its rh variable and is then delayed. While p is delayed, other processes complete pushRight and popLeft operations so that the node referenced by p’s rh variable is popped from the deque by
a \texttt{popLeft} without the deque being empty in that period. P resumes execution and performs the test (\texttt{rh.R == rh}). It passed the test because \texttt{rh} has been removed by a \texttt{popLeft}, and \texttt{popRight} returns empty. The empty checking part of the algorithm describes that if \texttt{rh} points to a right-dead node, then the deque is empty. Since the deque was never empty during the operation of \texttt{p}, this execution is not linearizable. This is because of the deficiency of the empty checking.

Already our model works well provided that it is possible to find this bug by linearization checking macros. In Figure 4.3 you see there is one inline that checks if the sequential set is empty or not (\texttt{seq.popCheckIfEmpty ( was_empty )}). The linearization point of the pop operation that returns empty is the point at which it reads its \texttt{rh} variable, so atomically in that moment, our Promela model checks if the sequential set is empty. The result is kept by one boolean value (\texttt{was_empty}), and never changes until the end of the thread execution. The algorithm itself has lines of code for empty checking. When empty checking becomes true, it returns empty. In this moment we verify if the deque really is empty, by asserting that \texttt{was_empty} is true. In the above scenario, we can see the violation of assertion. Two processes push 9 and 20 in deque, one process pops 20 and the other one returns that the deque is empty! The execution graph of this error which is the shortest error tracing shows in Figure 5.1. The verification output data also can be found in Table 5.1.

- \textbf{Second Bug} Simon Doherty found that the Snark allows a node to be removed from the deque twice, causing its value to be returned twice. Suppose that two processes want to delete the same node. Both load their local variables, but one of them succeeds
first, does its DCAS and pops the corresponding node. Before cleaning the value of the
node another process does its successful DCAS. It pops the same node and returns the
same value (Figure 5.2). This bug is an example of the ABA problem in algorithms
which are using CAS operation. Following is the scenario which leads to this bug:

Suppose process p invokes popRight when the deque contains more than one element
and runs alone until it reaches execute the DCAS inline, but is delayed before it executes
the DCAS. Other processes execute pushRight and popLeft operations so that p.rh =
LeftHat and the deque contains more than one element. This can be achieved without
modifying p.rh.L. Process q invokes and completes an execution of popLeft, and this
operation removes the node referenced by p.rh. This also happens without modifying
p.rh.L. Other processes execute popRight operations so that once again, p.rh =
RightHat. The deque is now empty. Finally, p executes its DCAS, which succeeds because
p.rh = RightHat and p.rh.L = p.rhL, and p returns p.rh.V, which has already been
returned by q.

As you can see in our model in Appendix A, we have fixed linearization points,
and they are immediately after DCAS in the pop operations and the return value of the
concurrent operation is known in those points. Atomically after DCAS we will run
the same operation in our sequential set, and put its result in resultSeq. Therefore we can
compare the result of the concurrent operation to the result of the sequential operation.
The assertion (result == resultSeq) will do this comparison. This assertion is vio-
lated and shows that the result of the concurrent operation is not the same as that of
the sequential, because as in the above scenario resultSeq is 0, but the result that the
concurrent one returns has a value.

We simulated our model by 7 processes, 3 pushRights, 2 popRights and 2 popLefts.
We tried to find this bug with exhaustive verification but unfortunately SPIN couldn’t
find it using 3 GB memory. It reports that there is not enough memory to verify this
model. With super trace mode, verification is done completely but it couldn’t find
this bug. Because super trace or bitstate hashing is an approximate verification method
(proof approximation) and it is not guaranteed which fraction of the state space is missed
(or covered), it is not exact in all cases. So, we have to simulate it interactively to find
its ABA problem.

You can find the execution graph of this error in Figure 5.2. As you see process 5
(popLeft) returns 20, process 4 (popRight) also returns 20!
You can also see the last part of simulation result.
spin: line 456 "pan_in", Error: assertion violated saw '-2'
spin: text of failed assertion: assert((result==resultSeq))
#processes: 6
319: proc 5 (popLeft) line 467 "pan_in" (state 110)
319: proc 4 (popRight) line 456 "pan_in" (state 82)
319: proc 3 (pushRight) line 370 "pan_in" (state 105)
319: proc 2 (pushRight) line 370 "pan_in" (state 105)
319: proc 1 (collect) line 513 "pan_in" (state 2)
319: proc 0 (:init:) line 534 "pan_in" (state 46)
9 processes created

After checking the correctness properties of the Snark algorithm without garbage collection, we added garbage collection macros to the Promela model and tried to run the same test harnesses. We again found the first bug, but interestingly not the second one. David Detlefs wrote in his paper that it is assumed that, after the pop routine exits, the node just popped will be reclaimed by the automatic storage allocator, through garbage collection or some such technique. But in our implementation, after every pointer updates atomically we will check if garbage conditions hold or not, if yes, we will run the garbage collector. So atomically after DCAS, the garbage collector will recognize that the node is garbage and collect it. In the second scenario, the reason of...
creating the bug is that, \( p.rh.L == p.rhL \) holds, i.e., the left field of the previously popped node has not been cleaned yet and has a value. But GC will immediately clean all the fields of a node, when the node is found to be garbage. That’s the reason that the \( p.rh.L == p.rhL \) will not hold any more, because the left field of the node is null now. So, again go around the while loop and check the empty checking, then report that the deque is empty.

You can find the execution graph of the second scenario with GC in Figure 5.3. It is clear that one \texttt{pop} process returns 20 and the other returns that the deque is empty!

As a conclusion, after finding the second bug we found that SPIN can not verify all concurrent algorithms completely. We tried both exhaustive and super trace verification manner, but SPIN could not trace the second bug itself. It seems that SPIN has problems handling a large number of threads, due to the many possible interleavings. It seems that work on optimization is needed to make the approach scale to a larger number of threads [9]. All verification methods of SPIN except the exhaustive manner, are approximate and do not guarantee that all interleavings are tracked.

### 5.1.2 DCAS-Based Concurrent Deques

After the first implementation of the Snark algorithm, and finding its issues we tried to find another algorithm with a concurrent data structure similar to that of Snark. The other paper which we worked with is another concurrent dynamic deque developed by Ole Agesen et al [22]. Ole Agessen presents a linked-list-based algorithm that does not
restrict concurrency in accessing the deques at two ends. It has two new linearizable non-blocking implementations of concurrent deques using the DCAS operation. The first uses an array representation, and improves on previous algorithms by allowing uninterrupted concurrent access to both ends of the deque while correctly handling the difficult boundary cases when the deque is empty or full. The second uses a linked-list representation, and is the first non-blocking unbounded-memory deque implementation. It too allows uninterrupted concurrent access to both ends of the deque. Authors have proved two algorithms correct with the aid of an automatic theorem prover. We decided to verify correctness properties of the second one by the SPIN model checker and with the help of our framework which we produced in the last experiment.

5.1.2.1 Algorithm Code

typedef node {
    pointer *L;
    pointer *R;
    val or null or sentL or sentR value;
}

typedef pointer {
    node *ptr;
    boolean deleted;
}

val pushRight(val v) {
    newL.ptr = new Node();
    if (newL.ptr == null) return "full";
    newL.deleted = false;
    while (true) {
        oldL = SR>L;
        if (oldL.deleted == true)
            deleteRight();
        else {
            newL.ptr->R.ptr = SR;
            newL.ptr->R.deleted = false;
            newL.ptr->L = oldL;
            newL.value = v;
            oldLR.ptr = SR;
            oldLR.deleted = false;
            if (DCAS(&SR->L, &oldL.ptr->value,
                oldL, oldLR, newL, newL))
                return "okay";
        }
    }
}

val popRight() {
    while (true) {
        oldL = SR>L;
        v = oldL.ptr->value;
        if (v == "sentL") return "empty";
        if (oldL.deleted == true)
            deleteRight();
        else if (v == "null") {
            if (DCAS(&SR->L, &oldL.ptr->value,
                oldL, v, oldL, v))
                return "empty";
        } else {
            newL.ptr = oldL.ptr;
newL.delete = true;
if (DCAS(&SR->L, &oldL.ptr->value, "null", newL))
    return v;
}
}

deleteRight() {
    while (true) {
        oldL = SR->L;
        if (oldL.delete == false) return;
        oldLL = oldL.ptr->L.ptr;
        if (oldLL->value != "null") {
            oldLLR = oldLL->R;
            if (oldL.ptr == oldLLR.ptr) {
                newR.ptr = SR;
                newR.delete = false;
                if (DCAS(&SR->L, &oldLLR, oldL, oldLL, newR))
                    return;
            }
        }
        if (oldR.delete) {
            newL.ptr = SL;
            newL.delete = false;
            newR.ptr = SR;
            newR.delete = false;
            if (DCAS(&SR->L, &SL->R, oldR, oldL, newL, newR))
                return;
        }
    }
}

deleteLeft() {
    while (true) {
        oldR = SL->R;
        if (oldR.delete == false) return;
        oldRRL = oldR.ptr->R.ptr;
        if (oldRRL->value != "null") {
            oldRRLR = oldRRL->L;
            if (oldR.ptr == oldRRLR.ptr) {
                newL.ptr = SL;
                newL.delete = false;
                    return;
            }
        }
        if (oldL.delete) {
            newR.ptr = new Node();
            if (newR.ptr == "null") return "full";
            newR.delete = false;
            while (true) {

            }
        }
    }
}
The algorithm is based on a technique for splitting the pop operation into two steps, marking that a node is about to be deleted, and then deleting it. Once marked, the node is considered deleted, and the actual deletion from the list can then be performed by the next push or next pop operation on that side of the deque. The key to making this algorithm work is the use of DCAS to synchronize delete operations correctly when the processes detect that there are only marked nodes in the list, and attempt to delete one or more of these nodes concurrently from both ends. The cost of this splitting technique is an extra DCAS per pop operation. The benefit is that it allows non-blocking completion without needing to synchronize on both of the deque’s end pointers with a DCAS. The splitting also requires allocating a bit in the pointer word to indicate if it is pointing to a marked node that needs to be deleted. However, this extra bit can be avoided, by adding two dummy delete-bit records to the structure.

Ole Agessen believes that through the design of linearizable lock-free implementations of classical data structures such as deques, it’s possible to understand better the power of the DCAS abstraction, and whether one should continue the effort to provide support for implementing it on concurrent hardware and software processing platforms.
Table 5.2: Verification Output of DCAS-BASED concurrent deques

<table>
<thead>
<tr>
<th>Stored states</th>
<th>Matched states</th>
<th>Transition</th>
<th>Atomic steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1427833</td>
<td>958113</td>
<td>2385946</td>
<td>501999</td>
</tr>
</tbody>
</table>

5.1.2.2 Verification and Simulation Results

The Promela model of the algorithm is available in Appendix B.1. But you can find the results of verification graphs in the following parts. All the verification was done using 3GB of memory. We use all the macros and inlines which we produced for the Snark algorithm, but to suit this algorithm and implementations of the Promela language, we changed the DCAS macro a bit. Because it needs to check pointers which have two parts: one their addresses and their deleted bit. We didn’t find any bug with the help of SPIN model checker. We simulated our model according to previous test harness that we did for Snark algorithm, but non of those bugs were found during simulation. With the help of linearization checking macros we found that this algorithm is linearizable and eliminates those issues which arise from DCAS operation. Table 5.2 shows verification results for two pushRight, one popRight and one popLeft processes.
Chapter 6

Experiments And Evaluation

In this section we evaluate the performance of our implementations by modeling some other concurrent algorithms. The purpose is to estimate how well the implementation works and how much it helps the verification of concurrent algorithms by SPIN in comparison with not using it. After that we ran a test harness on simulating and verifying algorithms with and without garbage collection. We tried to find what we can do to decrease the number of states which are explored by SPIN in the verification procedure.

6.1 Using the Framework For Modeling Algorithms In Promela

As we mentioned in Chapter 3, the most important goal of this project was creating some macros in Promela independent from the specific algorithm, which can be used in verification of other algorithms by SPIN. First we started by one simple algorithm, called Snark. We defined some important principles and correctness properties which are important to evaluate them in model checking. Then we tried to develop them in the Promela language because it is a language which is used by SPIN. We finished our work on Snark by creating Memory allocation, Linearizability checking and garbage collection macros. Then we started working on other concurrent algorithms which were similar to the previous one. We chose DCAS-based concurrent deques [22], and from the first steps of implementation we augmented the model with the generated framework. It was interesting that generating the model took less than 3 days because we already had all the macros which the model needs to verify and prove its correctness properties. Indeed it motivates us to work on concurrent skiplists [23] which we did on verification course during 3 months. Now we had most parts of the model in an organized shape. We
Chapter 6. Experiments And Evaluation

just copied our macros and implemented the model again in almost 2 days. Also one of my colleagues, who was working on verification of one parallel library, named Wool, in SPIN, used our framework and he also confirmed it was very useful for him. Because he used a lot of our linearization and dynamic memory allocation macros.

We believe the Framework is useful for those who are working on verification and especially those who are working on model checking. It is not only a well-formed structure in Promela but also a good contribution to save time.

6.2 Reducing State Spaces

The success of SPIN in industrial software development is primarily due to the efficiency with which it carries out in verifications. SPIN uses depth first search algorithm to track all the possible states. The advantage of the depth-first search is that the only data you need to store is the sequence of the states on the path to the current state, together with an indication for each state. SPIN does not actually search all the states. This means that the graph which will be produced from states, is constructed and then searched. Instead, for each transition that is considered, SPIN builds the target state on-the-fly. This can make the search much more efficient because SPIN needs only construct states until the first counterexample is found.

The data stored for each state, called the state vector, consist of the location counters of the processes and the values of the variables. Clearly, verification will be more efficient if the amount of memory needed to store a state vector is as small as possible. There are several things we can do to reduce state spaces, for example: use as few processes as possible, not declare unnecessary variables, and declare variables with as narrow type as possible, avoid declaring channel capacities in excess of what is needed to verify the model and use atomic and d_step where possible, but be sure that we are not masking possible error states by incorrectly restricting the set of possible interleavings.

These tricks motivated us to try to create smaller state spaces, so that memory requirements will be reduced for verification. The first decision was to implement garbage collection in our model. Then try to use some methods which exclude symmetric states from all states, so the number of states which SPIN needs to track decrease. In the next sections you can find the results, and their comparison we got from these experiments.

6.2.1 Comparison of Using Garbage Collection With Not Using

We explained in detail what garbage collection is doing in our implementation, we tried to convince our idea that, using garbage collection reduce the number of states and
memory usage in consequence, but actually the computation which SPIN needs to do is complex enough to generate more states when using GC.

6.2.1.1 Test Harness

Figure 6.1 shows the verification result when we use garbage collection for the Snark algorithm. We ran our model with two push and two pop processes. There were 183657 states generated which was equivalent to 26.524 Mb memory usage during 61 seconds to find the first bug of Snark, (compare to 5.1, 43756 states which was equivalent to 7.090 Mb memory usage during 10 seconds to find the first bug of Snark without using GC). It is clear that using GC produces more states.

\begin{verbatim}
pan:1: assertion violated was empty (at depth 223)
State-vector 120 byte, depth reached 299, errors: 1
183657 states, stored
224215 states, matched
407872 transitions (= stored+matched)
756444 atomic steps
hash conflicts: 22541 (resolved)
Stats on memory usage (in Megabytes):
  23.120 equivalent memory usage for states (stored+(State-vector + overhead))
  24.306 actual memory usage for states (unsuccessful compression: 105.13%)
  state-vector as stored = 127 byte + 12 byte overhead
  2.000 memory used for hash table (−w19)
  0.305 memory used for DFS stack (−m10000)
  26.524 total actual memory usage
\end{verbatim}

Figure 6.1: result of verification with garbage collection

6.2.2 Symmetry Reduction

Our models were expensive in terms of use of space and exploration cost. For example, it was clear that a permutation of the array (that represents the heap) would yield an ”equivalent” state, but the permuted state is strictly speaking distinct from the original, and all its descendants would be explored by SPIN. Because of this, We are interested in working on decreasing the number of states by symmetry reduction, which (I believe) addressed this issue. We decided to use an abstraction function, i.e, make the original array ’hidden’ and introduce a new version with abstracted state. The main challenge is to make the updates to the abstract state atomic with all changes to the original array. We use also this method and try to hide some states which don’t really need to be explored but SPIN will track them. So, we hide our original array and use one abstracted one to update it atomically whenever it is necessary. Unfortunately we have some local pointers and local variables which also need to be update also atomically after updating the global array. This work needs lots of computation and the state space is already prohibitively large, so we opted not to do so.
int n = 0;
proc type P() {
  byte temp;
  byte i = 1;
  do
    : i < 10 ->
      temp = n + 1;
      n = temp; i ++;
    : else -> break;
  od;
}

init {
  atomic {
    run P();
    run P();
  }
  (np_rpr == 1) -> assert (n > 2);
}

Figure 6.2: Counting with interference

It was interesting to understand how this works but first we do need a solid understanding of how the model checker works. SPIN is a state-based model checker. If we exclude data from the state (e.g., by using the 'hidden' keyword) then the model checker cannot track that data and cannot see when it changes. If we have two versions of the data, and we know that it is sufficient to track only one version, then we can declare the other version 'hidden'. We then have to make sure that whenever the one version of the data is updated (the concrete or real data) we also update (atomically) the other version (the abstract version of the data).

To understand what the benefits of this approach are, we used an interesting example of a Promela model in Figure 6.2. We wish to print the final value of $n$ after the two processes have completed executing their statements. This program is completely described in page 40 of Ben-Ari’s book [8]. It will check that the value of $n$ must be greater than 2. Verification reports that the assertion ($n > 2$) is violated, because it found one simulation where $n$ reaches to 2. Figure 6.4 and 6.3 show its verification and part of the simulation results. 48267 states were stored to find this violation. Then we defined variable $n$ to hidden, and declared the $h$ variable as an abstracted version of $n$. Whenever the value of $n$ changes, $h$ also atomically updates its value and becomes equal to $n$. In this way SPIN just tracks the value of $h$ to find when its value reaches 2. Interestingly the number of states which was explored by SPIN decreased to 35626, (26 state space decrease) Figure 6.5.
40. proc 1 (P) line 6 "pan-in" (state 1) [(i<10)]
41. proc 2 (P) line 7 "pan-in" (state 2) [temp = (n+1)]
42. proc 1 (P) line 7 "pan-in" (state 2) [temp = (n+1)]
43. proc 1 (P) line 8 "pan-in" (state 3) [n = temp]
44. proc 1 (P) line 7 "pan-in" (state 4) [i = (i+1)]
45. proc 1 (P) line 6 "pan-in" (state 1) [((i<10)]
46. proc 1 (P) line 7 "pan-in" (state 2) [temp = (n+1)]
47. proc 1 (P) line 7 "pan-in" (state 3) [n = temp]
48. proc 1 (P) line 7 "pan-in" (state 4) [i = (i+1)]
49. proc 1 (P) line 6 "pan-in" (state 1) [((i<10)]
50. proc 1 (P) line 7 "pan-in" (state 2) [temp = (n+1)]
51. proc 1 (P) line 8 "pan-in" (state 3) [n = temp]
52. proc 1 (P) line 7 "pan-in" (state 4) [i = (i+1)]
53. proc 1 (P) line 6 "pan-in" (state 1) [((i<10)]
54. proc 1 (P) line 7 "pan-in" (state 2) [temp = (n+1)]
55. proc 1 (P) line 8 "pan-in" (state 3) [n = temp]
56. proc 1 (P) line 7 "pan-in" (state 4) [i = (i+1)]
57. proc 1 (P) line 6 "pan-in" (state 1) [((i<10)]
58. proc 1 (P) line 7 "pan-in" (state 2) [temp = (n+1)]
59. proc 1 (P) line 8 "pan-in" (state 3) [n = temp]
60. proc 1 (P) line 7 "pan-in" (state 4) [i = (i+1)]
61. proc 1 (P) line 6 "pan-in" (state 1) [((i<10)]
62. proc 1 (P) line 7 "pan-in" (state 2) [temp = (n+1)]
63. proc 1 (P) line 8 "pan-in" (state 3) [n = temp]
64. proc 1 (P) line 7 "pan-in" (state 4) [i = (i+1)]
65. proc 1 (P) line 6 "pan-in" (state 1) [((i<10)]
66. proc 1 (P) line 7 "pan-in" (state 2) [temp = (n+1)]
67. proc 1 (P) line 8 "pan-in" (state 3) [n = temp]
68. proc 1 (P) line 7 "pan-in" (state 4) [i = (i+1)]
69. proc 1 (P) line 6 "pan-in" (state 1) [((i<10)]
70. proc 1 (P) line 7 "pan-in" (state 2) [temp = (n+1)]
71. proc 1 (P) line 8 "pan-in" (state 3) [n = temp]
72. proc 1 (P) line 7 "pan-in" (state 4) [i = (i+1)]
73. proc 1 (P) line 8 "pan-in" (state 5) [else]
74. proc 1 (P) line 8 "pan-in" (state 3) [n = temp]
75. proc 2 (P) line 7 "pan-in" (state 4) [i = (i+1)]
76. proc 2 (P) line 8 "pan-in" (state 5) [else]
77. proc 2 terminates
78. proc 0 (init) line 16 "pan-in" (state 4) [((nr.pr==m))|spin: line 16 "pan-in",
Error: assertion violate saw '−2' text of failed assertion: assert ((n>2))

Figure 6.3: Counting with interference simulation result

pan:1: assertion violated (n>2) (at depth 79)
State−vector 24 byte, depth reached 81, errors: 1
48267 states, stored
17207 states, matched
65474 transitions (stored+matched)
1 atomic steps
hash conflicts: 604 (resolved)
Stats on memory usage (in Megabytes):
1.657 equivalent memory usage for states (stored+State−vector + overhead)
1.854 actual memory usage for states (unsuccessful compression: 111.89%)
state−vector as stored = 28 byte + 12 byte overhead
2.000 memory used for hash table (−w19)
0.305 memory used for DFS stack (−m10000)
4.063 total actual memory usage

Figure 6.4: Counting with interference verification result

pan:1: assertion violated (h>2) (at depth 61)
State−vector 24 byte, depth reached 63, errors: 1
35626 states, stored
17207 states, matched
52833 transitions (stored+matched)
1 atomic steps
hash conflicts: 411 (resolved)
Stats on memory usage (in Megabytes):
1.223 equivalent memory usage for states (stored+State−vector + overhead)
1.463 actual memory usage for states (unsuccessful compression: 119.63%)
state−vector as stored = 31 byte + 12 byte overhead
2.000 memory used for hash table (−w19)
0.305 memory used for DFS stack (−m10000)
3.672 total actual memory usage

Figure 6.5: Counting with interference verification of abstract data
Chapter 7

Conclusion And Future Work

7.1 Conclusion

We presented an implementation of a Framework, motivated by the intention to model check concurrent programs by SPIN, a famous model checking tool. The language of SPIN is Promela, so we implemented macros in the Promela language for memory management and verifying correctness properties mainly linearizability. We tried to have a structured, efficient and independent model as much as possible to be able to use it in verification of most concurrent algorithms which use concurrent data structures. This framework helps verifiers to spend short time on modeling and as a result of that, they will have more time to investigate the correctness properties. We applied our framework to the verification of a non blocking algorithm for concurrent access to a double ended queue that supports four operations: pushRight, pushLeft, popRight, popLeft. The algorithm depends on an environment that supports automatic storage reclamation. Widespread use of garbage collection in concurrent programming motivates us to provide SPIN with support for garbage collection.

The framework includes dynamically allocated data structures and garbage collection which are added to the promela model manually. In addition to having organized framework we tried to find if SPIN produces less states in the verification process when using garbage collection, but according to our experiments we found that actually it increases the number of states which SPIN must track. The most important reason is adding more local variables such as local pointers to our data structure.

We presented our experience with checking linearizability of concurrent data structure algorithms. We focused on checking fixed linearization points and modeled checking by one sequential array which imitates the behavior of the original one. Our first implementation takes lots of time. Then we excluded some parts from our model and changed
them to some separate macros. When we started to model the second example we spent around two or three days because most parts were ready. We believe our framework will be instructive to anyone who attempts to specify and model check concurrent algorithms in SPIN.

### 7.2 Future Works

As we mentioned previously, Promela does not support function calls and recursive functions. But we can also work around it and create a macro using Promela’s channel capability. The channels can send values of every inline or proctype to each other. It can be a good facility to implement recursive calls.

We believe using symmetry reduction and partial order methods is quite useful. Some authors have attempted to apply abstraction techniques to nonblocking algorithms [14]. It is very difficult to apply abstractions to these algorithms but it would be valuable. We did some work around it and its result is in Chapter 6. But we couldn’t complete it. We know that some work is needed to find how the original array with local pointers can be abstracted.

Finally, we worked just on modeling linearization checking macros for fixed linearization points. But there would be a possibility for concurrent algorithms to have non fixed linearization points. For example the linearization point of a remove operation that returns false for a key \( k \) is either before a successful addition of the key \( k \) by another concurrent thread or after a successful removal of the key \( k \) by another concurrent thread. It would be especially helpful to create macros for checking non-fixed points, because knowing these points is crucial to understanding how the algorithms work.
Appendix A

AppendixA

A.1 Promela Model of The Snark Algorithm

```plaintext
#define MemSize 4
#define SeqSize 3 //MemSize-1
#define Undef MemSize+1
#define pointer byte
#define no_result 0
#define USEGC 0

#define FOR(i, l, h) i = l; do :: i <= h ->
#define ROF(i, l, h) ; i++ : : i > h -> break od

#define IF if
#define FI :: else -> skip ; fi;

typedef Node {
  pointer R;
  pointer L;
  byte V;
} Node
node mem [MemSize];
pointer RightHat;
pointer LeftHat;
pointer Dummy;
bit used [MemSize];
#if USEGC
byte RC[MemSize];
bit toBeCollected [MemSize];
#endif
byte Seqset [SeqSize];
pointer endpointer;
mtype = {emp , fu ,OK};
chan NodeValue = [0] of {mtype.int};
byte i ;

#define incRC (ptr) RC[ptr]++
#define decRC(ptr) assert (RC[ptr] > 0); RC[ptr]--;
def garbageGenerated (ptr)

#define garbageGenerated (ptr)
atomic{
  IF :: (RC[ptr] == 0) -> toBeCollected [ptr] = 1; gc();
  :: ((RC[ptr] == 1) && ((node_mem[ptr].R == ptr) || (node_mem[ptr].L == ptr))) ->
    toBeCollected [ptr] = 1; gc()
  :: ((RC[ptr] == 2) && ((node_mem[ptr].R == ptr) && (node_mem[ptr].L == ptr))) ->
    toBeCollected [ptr] = 1; gc()

37
```


```c
#define check_if_Garbage_Generated_Again(ptr)
atomic{
  IF
  :: (RC[ptr] == 0) -> toBeCollected[ptr] = 1; newGarbageGenerated = 1;
  :: ((RC[ptr] == 1) && ((node_mem[ptr].R == ptr) || (node_mem[ptr].L == ptr))) ->
    toBeCollected[ptr] = 1;
    newGarbageGenerated = 1
  :: ((RC[ptr] == 2) && ((node_mem[ptr].R == ptr) && (node_mem[ptr].L == ptr))) ->
    toBeCollected[ptr] = 1;
    newGarbageGenerated = 1
  FI
};

#define gc()
atomic{
  FOR(i, 0, MemSize - 1)
  IF
  :: toBeCollected[i] == 1 ->
    used[i] = 0;
    RC[node_mem[i].R]--;
    check_if_Garbage_Generated_Again(node_mem[i].R);
    node_mem[i].R = Undefined;
    RC[node_mem[i].L]--;
    check_if_Garbage_Generated_Again(node_mem[i].L);
    node_mem[i].L = Undefined;
    toBeCollected[i] = 0;
  FI
  ROF(i, 0, MemSize - 1);
  i = 0;
  newGarbageGenerated = 0;
};

#define newNode(nd)
atomic {
  node_c = 0;
  do
  :: (node_c >= MemSize) -> break
  :: else ->
    if
      :: (used[node_c] == 0) ->
        used[node_c] = 1;
        set_ptr(nd, node_c);
        break
      :: else -> node_c++
    fi;
  od;
  node_c = 0
};

#if USE_GC
inline set_ptr(ptr, value){
  atomic{
    IF
    :: (ptr != Undefined) -> decRC(ptr);
    FI
    ptr = value;
    IF
    :: (ptr != Undefined) -> incRC(ptr);
    FI
  }
}
#else
#define set_ptr(ptr, value) ptr = value;
#endif
```
#define DCAS(addr1, addr2, old1, old2, new1, new2, resultDcas)
    atomic{
        if ((addr1 == old1) && (addr2 == old2)) {
            set_ptr(addr1, new1);
            set_ptr(addr2, new2);
            resultDcas = true;
        } else {
            resultDcas = false;
        }
    }
    #define return_result(state, result) NodeValue ! state, result;
    #define linearizability
    inline seq_pushR(v) {
        atomic{
            IF {
                if (endpointer < SeqSize) {
                    Seqset[endpointer] = v; endpointer++;
                } FI
            }
        }
    }
    inline seq_popR(resultSeq) {
        atomic{
            assert(endpointer != 0); resultSeq = Seqset[endpointer - 1];
            Seqset[endpointer - 1] = 0; endpointer--;
        }
    }
    inline seq_pushL(v) {
        atomic{
            i = endpointer;
            IF {
                if (endpointer < SeqSize) {
                    do {
                        i > 0 -> Seqset[i] = Seqset[i - 1]; i--;
                        i == 0 -> break;
                    } od;
                    endpointer++;
                    Seqset[0] = v;
                } FI
            i = 0;
        }
    }
    inline seq_popL(resultSeq) {
        atomic{
            assert(endpointer != 0);
            assert(Seqset[0] != 0);
            resultSeq = Seqset[0];
            Seqset[0] = 0;
            FOR(i, 0, endpointer - 2)
                Seqset[i] = Seqset[i + 1];
            ROF(i, 0, endpointer - 2);
            i = 0;
            endpointer--;
            Seqset[endpointer] = 0;
        }
    }
    inline seq_popCheckIfEmpty(was_empty) {
        atomic{
            if {
                if (endpointer == 0) {
                    was_empty = true;
                } else {
                    was_empty = false;
                }
            } FI
        }
    }
FOR \( i \), \( 0 \), MemSize - 1
\[ \text{node}_\text{mem}[i].R = \text{Undef}; \]
\[ \text{node}_\text{mem}[i].L = \text{Undef}; \]
ROF \( i \), \( 0 \), MemSize - 1;
i = 0;
}

**inline** init_deque()

\[
\text{RightHat} = \text{Undef};
\text{LeftHat} = \text{Undef};
\text{Dummy} = \text{Undef};
\text{newNode(Dummy)};
\text{set_ptr(node}\_\text{mem}[\text{Dummy}].\text{L}, \text{Dummy});
\text{set_ptr(node}\_\text{mem}[\text{Dummy}].\text{R}, \text{Dummy});
\text{set_ptr(LeftHat, Dummy)};
\text{set_ptr(RightHat, Dummy)};
\]

**inline** define_pushR_locals()

\[
\text{pointer rh;}
\text{pointer lh;}
\text{pointer rhR;}
\text{pointer nd;}
\text{bit resultDcas;}
\text{byte node_c;}
\text{bit newGarbageGenerated;}
\text{rh} = \text{Undef;}
\text{lh} = \text{Undef;}
\text{rhR} = \text{Undef;}
\text{nd} = \text{Undef;}
\text{resultDcas} = 0;
\text{newGarbageGenerated} = 0;
\]

**inline** reset_pushR_locals()

\[
\text{d_step}(
\text{set_ptr(nd, Undef)};
\text{set_ptr(lh, Undef)};
\text{set_ptr(rh, Undef)};
\text{set_ptr(rhR, Undef)};
\text{resultDcas} = 0;
\text{node_c} = 0;
\text{newGarbageGenerated} = 0;
\)

**inline** define_pushL_locals()

\[
\text{pointer rh;}
\text{pointer lh;}
\text{pointer lhL;}
\text{pointer nd;}
\text{bit resultDcas;}
\text{byte node_c;}
\text{bit newGarbageGenerated;}
\text{rh} = \text{Undef;}
\text{lh} = \text{Undef;}
\text{lhL} = \text{Undef;}
\text{nd} = \text{Undef;}
\text{resultDcas} = 0;
\text{newGarbageGenerated} = 0;
\]

**inline** reset_pushL_locals()

\[
\text{d_step}(
\text{set_ptr(nd, Undef)};
\text{set_ptr(lh, Undef)};
\text{set_ptr(rh, Undef)};
\text{set_ptr(lhL, Undef)};
\text{resultDcas} = 0;
\)
node c = 0;
newGarbageGenerated = 0;
}

inline define_popR_locals(){
    pointer rh;
    pointer lh;
    pointer rhL;
    bit resultDcas;
    byte resultSeq;
    int result;
    bool was_empty;
    bit newGarbageGenerated;
    rh = Undef;
    lh = Undef;
    rhL = Undef;
    resultDcas = 0;
    resultSeq = 0;
    result = 0;
    was_empty = false;
    newGarbageGenerated = 0;
}

inline reset_popR_locals(){
    d_step{
        set_ptr(rh, Undef);
        set_ptr(lh, Undef);
        set_ptr(rhL, Undef);
        resultDcas = 0;
        resultSeq = 0;
        result = 0;
        was_empty = false;
        newGarbageGenerated = 0;
    }
}

inline define_popL_locals(){
    pointer rh;
    pointer lh;
    pointer lhR;
    bit resultDcas;
    byte resultSeq;
    int result;
    bool was_empty;
    bit newGarbageGenerated;
    rh = Undef;
    lh = Undef;
    lhR = Undef;
    resultDcas = 0;
    resultSeq = 0;
    result = 0;
    was_empty = false;
    newGarbageGenerated = 0;
}

inline reset_popL_locals(){
    d_step{
        set_ptr(rh, Undef);
        set_ptr(lh, Undef);
        set_ptr(lhR, Undef);
        resultDcas = 0;
        resultSeq = 0;
        result = 0;
        was_empty = false;
        newGarbageGenerated = 0;
    }
}

/***************************************************************************/
/**START OF ALGORITHM***************************************************************************/

proctype pushRight(int v){
define_pushR_locals();
newNode(nd);
IF :: (nd >= MemSize) -> return_result(fu, no_result); goto end;
FI
set_ptr(node_mem[nd]. R, Dummy);
node_mem[nd]. V = v;
start:
do :: true ->
set_ptr(rh, RightHat);
set_ptr(rhR, node_mem[rh]. R);
if :: (rhR == rh) ->
set_ptr(node_mem[nd]. L, Dummy);
set_ptr(lh, LeftHat);
atomic{
    DCAS(RightHat, LeftHat, rh, lh, nd, nd, resultDcas);
    if :: resultDcas -> seq_pushR(v);
    else -> goto start;
fi;
return_result(OK, v);
} break;
else ->
set_ptr(node_mem[nd]. L, rh);
atomic{
    DCAS(RightHat, node_mem[rh]. R, rh, rhR, nd, nd, resultDcas);
    if :: resultDcas -> seq_pushR(v);
    else -> goto start;
fi;
return_result(OK, v);
} break;
fi;
od;
end : reset_pushR_locals();

/*PUSH LEFT*/
proctype pushleft(int v){
    define_pushL_locals();
    newNode(nd);
    IF :: (nd >= MemSize) -> return_result(fu, no_result); goto end;
    FI
set_ptr(node_mem[nd]. L, Dummy);
node_mem[nd]. V = v;
start:
do :: true ->
set_ptr(lh, LeftHat);
set_ptr(lhL, node_mem[lh]. L);
if :: (lhL == lh) ->
set_ptr(node_mem[nd]. R, Dummy);
set_ptr(rh, RightHat);
atomic{
    DCAS(LeftHat, RightHat, lh, rh, nd, nd, resultDcas);
    if :: resultDcas -> seq_pushL(v);
    else -> goto start;
fi;
return_result(OK, v);
} break;
else ->
set_ptr(node_mem[nd]. R, lh);
atomic{
    DCAS(LeftHat, node_mem[lh]. L, lh, lhL, lhL,
nd, nd, resultDcas);
if
:: resultDcas -> seq.pushL(v);
:: else -> goto start;
f1;
return_result(OK, v);
}
break;
f1;
od;
end: reset_pushLlocals();
}

// POP RIGHT
proctype popRight (){
define_popR_locals();
start:
do:
:: true ->
atomic(set_ptr(rh,RightHat); seq_popCheckIfEmpty(was_empty);)
set_ptr(lh,LeftHat);
atomic{
IF
:: node_mem[rh].R == rh ->
return_result(emp, no_result);
assert(was_empty);
break;
FI
};
if
:: rh == lh ->
atomic{
DCAS(RightHat,LeftHat,rh,lh,
Dummy,Dummy,resultDcas);
if
:: resultDcas -> seq_popR(resultSeq);
:: else -> goto start;
f1;
}
result = node_mem[rh].V;
return_result(OK, result);
assert(result == resultSeq);
break;
:: else -> set_ptr(rhL,node_mem[rh].L);
atomic{
DCAS(RightHat,node_mem[rh].L,rh,
lh, rhL, rh, resultDcas);
if
:: resultDcas -> seq_popR(resultSeq);
:: else -> goto start;
f1;
}
result = node_mem[rh].V;
set_ptr(node_mem[rh].R,Dummy);
node_mem[rh].V = 0;
return_result(OK, result);
assert(result == resultSeq);
break;
f1;
}
reset_popRlocals();
}

// POP LEFT
proctype popLeft (){
define_popL_locals();
start:
do:
:: true ->
atomic(set_ptr(lh,LeftHat); seq_popCheckIfEmpty(was_empty);)
set_ptr(rh,RightHat);


atomic{
  if
    if
      node_mem[lh].L == lh ->
        return_result(emp, no_result);
        assert(was_empty);
        break;
    fi
  }
  if
    if
      node_mem[lh].R == rh ->
        atomic{
          DCAS(RightHat, LeftHat, lh, rh,
               Dummy, Dummy, resultDcas);
          if
            resultDcas -> seq_popL(resultSeq);
            else -> goto start;
          fi;
          result = node_mem[lh].V;
          return_result(OK, node_mem[lh].V);
          assert(result == resultSeq);
          break;
        };
        else -> set_ptr(lhR, node_mem[lh].R);
        atomic{
          DCAS(LeftHat, node_mem[lh].R, lh,
               lhR, lhR, lhR, resultDcas);
          if
            resultDcas -> seq_popL(resultSeq);
            else -> goto start;
          fi;
          result = node_mem[lh].V;
          set_ptr(node_mem[lh].L, Dummy);
          node_mem[lh].V = 0;
          return_result(OK, result);
          assert(result == resultSeq);
          break;
        };
      fi;
  od;
  reset_popL_locals();
}

proctype collect(){
  end:
  do
    NodeValue ? _, _
  od
}

init {
  bit newGarbageGenerated;
  byte node_c;
  init_memory();
  init_deque();
  run collect();
  run pushRight(9);
  run popRight();
  run pushRight(20);
  run popLeft();
}
Appendix B

B.1 Promela Model of DCAS-Based Concurrent Deques

```c
#define MemSize 5
#define SeqSize 3 /*MemSize-2(separate two sentinel)*/
#define Undef MemSize+1
#define n_result 0
#define null 0
#define sentL 200
#define sentR 200

#define FOR( i , l , h) i = l ; do :: i <= h ->
#define ROF( i , l , h) ; i++ : : i > h -> break od

#define IF if
#define FI : : else -> skip ; fi;

typedef pointer{ byte ptr ;
 bool deleted ;
}
typedef Node { pointer R; 
 pointer L ;
 byte Value ;
}
Node node_mem[MemSize] ;
p pointer SL ;
p pointer SR ;
b bit used[MemSize] ;
byte Seqset[SeqSize] ;
p pointer endpointer ;
mytype = {emp, fu, OK};
chan NodeValue = [0] of (mytype, int); 
byte i ;

#define newNode(nd)
atomic {
 node_c = 0 ;
do :: (node_c >= MemSize) -> break
 : : else ->
if :: (used[node_c] == 0) ->
 used[node_c] = 1 ;
```
set_ptr(nd, node_c)
    break
    :: else => node_c++
fi;
node_c = 0
}

#define set_ptr(ptr, value) ptr = value;
#define DCAS(addr1, addr11, addr2, addr22, old1, old11, old2, old22, new1, new11, new2, new22, resultDcas)
atomic{
    if
      :: ((addr1 == old1) && (addr2 == old2) && (addr11 == old11) && (addr22 == old22)) ->
        set_ptr(addr1, new1);
        set_ptr(addr11, new11);
        set_ptr(addr2, new2);
        set_ptr(addr22, new22);
        resultDcas = true;
    else -> resultDcas = false;
    fi;
}

#define return_result(state, result) NodeValue ! state, result;

STRUCTOR

inline seq_pushR(v)
atomic{
    IF
      :: (endpointer_ptr < SeqSize) -> Seqset[endpointer_ptr] = v; endpointer_ptr++; 
    FI
}

inline seq_popR(resultSeq)
atomic{
    assert(endpointer_ptr != 0); resultSeq = Seqset[endpointer_ptr - 1];
    Seqset[endpointer_ptr - 1] = 0; endpointer_ptr--; 
}

inline seq_pushL(v)
atomic{
    i = endpointer_ptr;
    IF
      :: (endpointer_ptr < SeqSize ) ->
        do
          :: i > 0 -> Seqset[i] = Seqset[i-1]; i--;
          :: i == 0 -> break;
        od;
        endpointer_ptr++;
        Seqset[0] = v;
    FI
    i = 0;
}

inline seq_popL(resultSeq)
atomic{
    assert(endpointer_ptr != 0);
    resultSeq = Seqset[0];
    Seqset[0] = 0;
    FOR(i, 0, endpointer_ptr - 2)
      Seqset[i] = Seqset[i+1];
    ROF(i, 0, endpointer_ptr - 2);
    i = 0;
    endpointer_ptr--;
    Seqset[endpointer_ptr] = 0;
}

inline seq_popCheckIfEmpty(was_empty)
atomic{
    IF

:: endpointer.ptr == 0 -> was_empty = true;
:: else -> was_empty = false;
fi;
}

/******************************************************************************
* INITIALIZATION******************************************************************************/

inline init_memory(){
FOR(i, 0, MemSize-1)
node_mem[i].R.ptr = Undef;
node_mem[i].L.ptr = Undef;
ROF(i, 0, MemSize-1);
i = 0;
}

inline init_deque(){
SL.ptr = Undef;
SR.ptr = Undef;
newNode(SL.ptr);
newNode(SR.ptr);
set_ptr(node_mem[SR.ptr].L.ptr,SL.ptr);
set_ptr(node_mem[SL.ptr].R.ptr,SR.ptr);
SR.deleted = 0;
SL.deleted = 0;
node_mem[SL.ptr].Value = sentL;
node_mem[SR.ptr].Value = sentR;
}

inline define_popR_locals(){
pointer oldL;
poiner newL;
poiner oldLL;
poiner oldLLR;
poiner newR;
poiner oldR;
byte v;
bit resultDcas;
byte resultSeq;
bool was_empty;
oldL.ptr = Undef;
newL.ptr = Undef;
oldLL.ptr = Undef;
oldLLR.ptr = Undef;
newR.ptr = Undef;
oldR.ptr = Undef;
oldL.deleted = 0;
newL.deleted = 0;
oldLL.deleted = 0;
oldLLR.deleted = 0;
newR.ptr = 0;
oldR.deleted = 0;
resultDcas = 0;
resultSeq = 0;
v = 0;
was_empty = false;
bool noDeletedBit;
noDeletedBit = false;
}

inline reset_popR_locals(){
set_ptr(oldL.ptr, Undef);
set_ptr(newL.ptr, Undef);
set_ptr(oldLL ptr, Undef);
set_ptr(oldLLR.ptr, Undef);
set_ptr(newR.ptr, Undef);
set_ptr(oldR.ptr, Undef);
}

inline define_popL_locals(){
poiner oldR;
pointer newR;
pointer oldRR;
pointer oldRRL;
pointer newL;
pointer oldL;
byte v;
bit resultDCas;
byte resultSeq;
bool was_empty;
oldR.ptr = Undef;
newR.ptr = Undef;
oldRR.ptr = Undef;
oldRRL.ptr = Undef;
newL.ptr = Undef;
oldL.ptr = Undef;
oldR.deleted = 0;
newR.deleted = 0;
oldRR.deleted = 0;
oldRRL.deleted = 0;
newL.ptr = 0;
oldL.deleted = 0;
resultDCas = 0;
resultSeq = 0;
v = 0;
was_empty = false;
bool noDeletedBit;
noDeletedBit = false;
}

inline reset_popL_locals() {
set_ptr(oldR.ptr, Undef);
set_ptr(newR.ptr, Undef);
set_ptr(oldRR.ptr, Undef);
set_ptr(oldRRL.ptr, Undef);
set_ptr(newL.ptr, Undef);
set_ptr(oldL.ptr, Undef);
}

inline define_pushR_locals() {
pointer newL;
pointer oldL;
pointer oldLR;
pointer nd;
pointer oldLL;
pointer oldLLR;
pointer newR;
pointer oldR;
byte node.c;
newL.ptr = Undef;
oldL.ptr = Undef;
oldLR.ptr = Undef;
nd.ptr = Undef;
oldLL.ptr = Undef;
oldLLR.ptr = Undef;
newR.ptr = Undef;
oldR.ptr = Undef;
newL.deleted = 0;
oldL.deleted = 0;
oldLR.deleted = 0;
oldLL.deleted = 0;
oldLLR.deleted = 0;
newR.ptr = 0;
nd.deleted = 0;
oldR.deleted = 0;
resultDCas = 0;
}

inline reset_pushR_locals() {
d_step(
set_ptr(md.ptr, Undef);
set_ptr(newL.ptr, Undef);
)
inline define_pushLLocals() {
    pointer newR;
    pointer oldR;
    pointer oldRL;
    pointer nd;
    pointer oldRR;
    pointer oldRRL;
    pointer newL;
    pointer oldL;
    bit resultDcas;
    byte node_c;
    bit newGarbageGenerated;
    newR.ptr = Undef;
    oldR.ptr = Undef;
    oldRL.ptr = Undef;
    nd.ptr = Undef;
    oldRR.ptr = Undef;
    oldRRL.ptr = Undef;
    newL.ptr = Undef;
    oldL.ptr = Undef;
    newR.deleted = 0;
    oldR.deleted = 0;
    oldRL.deleted = 0;
    oldRR.deleted = 0;
    oldRRL.deleted = 0;
    newL.ptr = 0;
    nd.deleted = 0;
    oldL.deleted = 0;
    resultDcas = 0;
}

inline reset_pushLLocals() {
    d_step{
        set_ptr(md.ptr,Undef);
        set_ptr(newR.ptr,Undef);
        set_ptr(oldR.ptr,Undef);
        set_ptr(oldRL.ptr,Undef);
        set_ptr(oldRR.ptr,Undef);
        set_ptr(oldRRL.ptr,Undef);
        set_ptr(newL.ptr,Undef);
        set_ptr(oldL.ptr,Undef);
    }
}

/**************************************************************************************/
/* START OF ALGORITHM **************************************************************************************/
inline deleteRight() {
    again:
    do :: true ->
        d_step{
            set_ptr(oldL.ptr,node_mem[SR.ptr].L.ptr);
            oldL.deleted = node_mem[SR.ptr].L.deleted;
        };
    if :: oldL.deleted == false -> break;
    else -> skip;
    fi;
    set_ptr(oldL.ptr,node_mem[oldL.ptr].L.ptr);
    if :: (node_mem[oldL.ptr].Value != null) ->
        d_step{
            set_ptr(oldLLR.ptr,node_mem[oldLL.ptr].R.ptr);
            oldLLR.deleted = node_mem[oldLL.ptr].R.deleted;
        };
    if
:: (oldL.ptr == oldLLR.ptr) -> set_ptr(newR.ptr,SR.ptr); newR.deleted = false;

atomic{
    DCAS(node_mem[SR.ptr].L.ptr,node_mem[SR.ptr].L.deleted,
         node_mem[oldLL.ptr].R.ptr,
         node_mem[oldLL.ptr].R.deleted,
         oldL.ptr,oldL.deleted,oldLLR.ptr,
         oldLLR.deleted,oldLL.ptr,oldLL.deleted,
         newR.ptr,newR.deleted,resultDCas);
    if :: resultDCas -> break;
    :: else -> skip;
    fi;
}: :: else -> skip;

}: :: else -> d_step{
    set_ptr(oldR.ptr,node_mem[SL.ptr].R.ptr);
    oldR.deleted = node_mem[SL.ptr].R.deleted;
};

if :: oldR.deleted ->
    set_ptr(newL.ptr,SL.ptr);
    newL.deleted = false;
    set_ptr(newR.ptr,SR.ptr);
    newR.deleted = false;

atomic{
    DCAS(node_mem[SR.ptr].L.ptr,
         node_mem[SL.ptr].R.ptr,
         node_mem[SL.ptr].R.deleted,
         node_mem[node_mem[newL.ptr].L.ptr].R.ptr,
         node_mem[node_mem[newL.ptr].L.delete].oldL.ptr,
         oldL.ptr,oldL.deleted,oldR.ptr,
         oldR.deleted,newL.ptr,newL.deleted,
         newR.ptr,newR.deleted,resultDCas);
    if :: resultDCas -> break;
    :: else -> skip;
    fi;
}: :: else -> skip;

fi;

} od;

proctype pushRight(int v){
define_pushR_locals();
newNode(nd.ptr);

IF :: (nd.ptr >= MemSize) -> return_result(fu,no_result); goto end;
FI

set_ptr(newL.ptr,nd.ptr); newL.deleted = false;
start:
do :: true ->
    d_step{
        set_ptr(oldL.ptr,node_mem[SR.ptr].L.ptr);
        oldL.deleted = node_mem[SR.ptr].L.deleted;
    };
    if :: (oldL.deleted) -> deleteRight();
    :: else -> set_ptr(node_mem[newL.ptr].R.ptr,SR.ptr);
        node_mem[newL.ptr].R.deleted = false;
        set_ptr(node_mem[newL.ptr].L.ptr,oldL.ptr);
        node_mem[newL.ptr].L.deleted = oldL.deleted;
        node_mem[newL.ptr].Value = v;
        set_ptr(oldLR.ptr,SR.ptr);
        oldLR.deleted = false;

atomic{
    DCAS(node_mem[SR.ptr].L.ptr,
         node_mem[SR.ptr].L.deleted,
         node_mem[oldL.ptr].R.ptr,
         node_mem[oldL.ptr].R.deleted,oldL.ptr,
         oldL.deleted,oldLR.ptr,oldLR.deleted,
         newL.ptr,newL.deleted,newL.ptr,newL.deleted,
         resultDCas);
    if :: resultDCas -> break;
    :: else -> skip;
    fi;
}: :: else -> skip;

fi;
Appendix B. AppendixB

51

if
:: resultDcas -> seq_pushR(v);
:: else -> goto start;
fi;
};
return_result(OK,v);
break;
fi;
}
end: reset_pushR_locals();
}

//<POP RIGHT>//
proctype popRight (){
define_popR_locals();
start:

if :: true ->
atomic{
  set_ptr(oldL.ptr,node_mem[SR.ptr].L.ptr);
  oldL.deleted = node_mem[SR.ptr].L.deleted;
  seq_popCheckIfEmpty(was_empty);
};
  v = node_mem[oldL.ptr].Value;
atomic{
  IF :: v == sentL ->
    return_result(emp,no_result);
    assert(was_empty);
    break;
  FI
  }
};
if :: oldL.deleted -> deleteRight();
:: else ->
if :: (v == null) ->
atomic{
  DCAS(node_mem[SR.ptr].L.ptr,
       node_mem[SR.ptr].L.deleted,
       node_mem[oldL.ptr].Value,noDeletedBit,oldL.ptr,
       oldL.deleted,v,noDeletedBit,oldL.ptr,oldL.deleted,noDeletedBit,false,
       resultDcas);
  if :: resultDcas -> return_result(emp,no_result);
    assert(was_empty);
    break;
  else -> goto start;
  fi;
};
:: else ->
set_ptr(newL.ptr,oldL.ptr);
newL.deleted = true;
atomic{
  DCAS(node_mem[SR.ptr].L.ptr,
       node_mem[SR.ptr].L.deleted,
       node_mem[oldL.ptr].Value,noDeletedBit,
       oldL.ptr,oldL.deleted,v,noDeletedBit,
       newL.ptr,newL.deleted,null,noDeletedBit,
       resultDcas);
  if :: resultDcas -> seq_popR(resultSeq);
  else -> goto start;
  fi;
}
assert(v == resultSeq);
return_result(OK,v);
break;
fi;
fi;
}
reset_popR_locals();
}
inline deleteLeft ()
  again:
  do :: true ->
    d_step{
      set_ptr(oldR.ptr, node_mem[SL.ptr].R.ptr);
      oldR.deleted = node_mem[SL.ptr].R.deleted;
    };
    if :: oldR.deleted == false -> break;
    else -> skip;
    fi;
    set_ptr(oldRR.ptr, node_mem[oldR.ptr].R.ptr);
    if :: (node_mem[oldRR.ptr].Value != null) 
      d_step{
        set_ptr(oldRRL.ptr, node_mem[oldRR.ptr].L.ptr);
        oldRRL.deleted = node_mem[oldRR.ptr].L.deleted;
      }
    
    if :: (oldR.ptr == oldRRL.ptr) 
      set_ptr(newL.ptr, SL.ptr);
      newL.deleted = false;
      atomic{
        DCAS(node_mem[SL.ptr].R.ptr, node_mem[SL.ptr].R.deleted, 
              node_mem[oldRR.ptr].L.ptr, node_mem[oldRR.ptr].L.deleted, 
              oldR.ptr, oldR.deleted, oldRRL.ptr, 
              oldRRL.deleted, oldRR.ptr, oldRR.deleted, 
              newL.ptr, newL.deleted, resultDcas);
        if :: resultDcas -> break;
        else -> skip;
        fi;
      };
    :: else -> skip;
    fi;
    :: else ->
    d_step{
      set_ptr(oldL.ptr, node_mem[SR.ptr].L.ptr);
      oldL.deleted = node_mem[SR.ptr].L.deleted;
    };
    set_ptr(newR.ptr, SR.ptr);
    newR.deleted = false; set_ptr(newL.ptr, SL.ptr);
    newL.deleted = false;
    if :: oldL.deleted ->
      atomic{
        DCAS(node_mem[SL.ptr].R.ptr, node_mem[SL.ptr].R.deleted, 
              node_mem[SR.ptr].L.ptr, node_mem[SR.ptr].L.deleted, 
              oldR.ptr, oldR.deleted, oldL.ptr, 
              oldL.deleted, newR.ptr, newR_deleted, 
              newL.ptr, newL.deleted, resultDcas);
        if :: resultDcas -> break;
        else -> skip;
        fi;
      };
    fi;
  };
}

proctype pushLeft(int v){
  define_pushL_locals ();
  newNode(nd.ptr);
  IF :: (nd.ptr >= MemSize) -> return_result(fu, no_result); goto end;
  FI
  set_ptr(newR.ptr, nd.ptr); newR.deleted = false;
  start:
  do
:: true ->
  d_step(
    set_ptr(oldR.ptr, node_mem[SL.ptr].R.ptr);
    oldR.deleted = node_mem[SL.ptr].R.deleted;
  );
  if
    (oldR.deleted) -> deleteLeft();
  else -> set_ptr(node_mem[newR.ptr].L.ptr, SL.ptr);
    node_mem[newR.ptr].L.deleted = false;
    set_ptr(node_mem[newR.ptr].R.ptr, oldR.ptr);
    node_mem[newR.ptr].R.deleted = oldR.deleted;
    node_mem[newR.ptr].Value = v;
    set_ptr(oldRL.ptr, SL.ptr);
    oldRL.deleted = false;
  atomic{
    DCAS(node_mem[SL.ptr].R.ptr, node_mem[SL.ptr].R.deleted,
      node_mem[oldR.ptr].L.ptr, node_mem[oldR.ptr].L.deleted,
      oldR.deleted, oldRL.ptr, oldRLdeleted, newR.ptr,
      newR.deleted, newR.ptr, newR.deleted, resultDcas);
    if
      resultDcas -> seq.pushL(v);
      else -> goto start;
    fi;
    return_result(OK, v);
    break;
  fi;
  od;
end: reset_pushLlocals();
}

//POP LEFT*/
proctype popLeft (){
  define_popLlocals();
  start:
  do
    :: true ->
      atomic{
        set_ptr(oldR.ptr, node_mem[SL.ptr].R.ptr);
        oldR.deleted = node_mem[SL.ptr].R.deleted;
        seq_popCheckIfEmpty(was_empty);
      };
      v = node_mem[oldR.ptr].Value;
      atomic{
        if
          v == sentR ->
            return_result(emp, no_result);
            assert(was_empty);
            break;
          FI
        fi;
        if
          oldR.deleted -> deleteLeft();
        else ->
          if
            (v == null) ->
              atomic{
                DCAS(node_mem[SL.ptr].R.ptr, node_mem[SL.ptr].R.deleted,
                  node_mem[oldR.ptr].L.ptr, node_mem[oldR.ptr].L.deleted,
                  oldR.deleted, v, noDeletedBit, oldR.ptr,
                  oldR.deleted, v, noDeletedBit, resultDcas);
                if
                  resultDcas -> return_result(emp, no_result);
                  assert(was_empty);
                  break;
                else -> goto start;
                fi;
              };
            else ->
              set_ptr(newR.ptr, oldR.ptr);
              newR.deleted = true;
              atomic{
                if
                  resultDcas -> return_result(emp, no_result);
                  assert(was_empty);
                  break;
                else -> goto start;
                fi;
              };
          else ->
            set_ptr(newR.ptr, oldR.ptr);
            newR.deleted = true;
            atomic{
              if
                resultDcas -> return_result(emp, no_result);
                assert(was_empty);
                break;
              else -> goto start;
              fi;
            };
        fi;
      fi;
  fi;
}

DCAS(node_mem[SL.ptr].R.ptr,
node_mem[SL.ptr].R.deleted,
node_mem[oldR.ptr].Value,noDeletedBit,
oldR.ptr,oldR.deleted,v,noDeletedBit,
newR.ptr,newR.deleted,null,noDeletedBit,
resultDcas);
if :
  resultDcas => seq_popL(resultSeq);
  else => goto start;
fi;
assert(v == resultSeq);
return_result(OK,v);
break;
fi;
reset_popL_locals();
}

proctype collect()
end:
do :
  NodeValue ? - , -
od
}
init {
byte node.c;
init_memory();
init_deque();
run collect();
run popRight();
run pushRight(10);
run pushRight(20);
run pushLeft(30);
}
Bibliography


