Verification of Work-stealing Deque Implementation

Majid Khorsandi Aghai
Abstract

Verification of Work-stealing Deque Implementation

Majid Khorsandi Aghai

Software verification is the process of checking a software system to make sure it meets its specifications. A software system can be either of sequential or concurrent type. The most important part in a sequential system from the verification point of view is the relationship between the system’s inputs and outputs. More formally, verification of a sequential system can be expressed in the following form: If a program starts in a specific state which satisfies a certain condition (precondition), then it eventually terminates and the program variables at the final state satisfy some given relation with the corresponding values at the beginning of the execution [20]. But the story in verification of concurrent software is different.

Many concurrent software use parallelism in order to make calculations more efficient. Parallelism is used to distribute a large amount of computations between different processing units to finish them in a shorter time. Input and output values are still enough for specifying the behavior of these sequential processes but are absolutely not enough for specifying the behaviors of the concurrent system. This is mainly because of the interactions between processes which can not only be expressed with the help of inputs and outputs.

One important aspect in verification of concurrent programs is to directly verify them against an abstract specification of overall functionality. For example, a concurrent implementation of a familiar data type abstraction, such as a queue, could be verified to conform to a simple abstract specification of such a data type. This has been accomplished for finite-state programs and some verification tools like SPIN [3, 13] are already supporting it, but to our knowledge there are no approaches for handling unbounded data domains in specifications and implementations as is the case for a work-stealing double ended queue (deque) implementation.

In this project we present a technique for automatically verifying that a concurrent workstealing deque conforms to an abstract specification of its functionality and we mathematically prove our technique’s correctness. We also demonstrate its use by applying it to a famous implementation of a work-stealing deque data structure presented by Arora, et al. [2]
Acknowledgements

First I would like to thank Prof. Bengt Jonsson who gave me the honor to work under his supervision in UPMARC group at IT department of Uppsala University. Working in this group with its perfect infrastructure, the nice working atmosphere and the large experience and knowledge I gathered there will serve as a stable basis for my future scientific activities.

I also thank Prof. Parosh Abdulla who agreed to review my master thesis and supported me in my thesis presentation.

Next, I would like to thank Ahmed Rezine for his generous help and guidance. Without his suggestions I would not be able to finish this work. Ahmed specially helped me a lot with mathematical proof I did in this thesis.

I want to thank my parents, Ahmad and Manijeh, without whom I would never have been able to achieve so much in my life. I cordially thank my brother, Amin, for his warm support during my two years of study in Sweden.

Last but not least, I especially wish to express my love for Atefeh, my dear wife, who only knows the real price of this thesis as she walked by my side in each and every step. I thank for her endless love, patience, and understanding.
Contents

List of Figures iv

1 Introduction 1
  1.1 Contribution of This Thesis 1
  1.2 Organization 3

2 Background 3
  2.1 Automatic Verification 4
  2.2 Model Checking 5
  2.3 Automata Theoretic Approach to Model Checking 6
  2.4 Formal Specification 6
  2.5 Data-Independence 7
  2.6 Work-Stealing Deque 8
  2.7 Linearizability 8
  2.8 Summary 9

3 Related Work 10

4 Verification Method 10
  4.1 Work-stealing deque by Arora et.al 10
  4.2 Specification 13
  4.3 Program Semantics and Observers 15
  4.4 Our Data Independence Argument 19

5 Correctness of Verification Method 20

6 Implementation 26
  6.1 State Space Growth 26
  6.2 Modeling Threads-Modeling Methods 26
  6.3 Unbounded Data Domain-Input Values 27
  6.4 Applying Verification Method to Model 28
  6.5 Checking properties 29
    6.5.1 DUPLICATE 29
    6.5.2 FIFO 30
    6.5.3 LIFO 30
    6.5.4 CREATION 31
    6.5.5 LOSS 31
  6.6 Verification Results 32

7 Conclusion and Future Work 33

A 37

ii
1 Instrumented Pseudo-code of Arora et al. Work-stealing Algorithm . . . . . . 38
B AppendixB 39
1 Promel model of Arora Algorithm Implementation . . . . . . . . . . . . . . 39
List of Figures

1  Automata theoretic approach to model checking .......................... 7
2  work-stealing algorithm .................................................. 11
3  The deque object ............................................................ 12
4  Observer for property ”DUPLICATE” .................................... 17
5  Observer for property ”LIFO” .............................................. 17
6  Observer for property ”FIFO” .............................................. 18
7  Observer for property ”LOSS” ............................................. 18
8  Observer for property ”CREATION” ...................................... 18
9  The checkDuplicate inline .................................................. 30
10  The code to check FIFO property ........................................ 30
11  The code to check LIFO property ........................................ 31
12  How we check CREATION property ..................................... 31
13  Checking LOSS property .................................................. 32
1 Introduction

Software verification is the process of checking a software system to make sure it meets its specifications and fulfills its intended purpose. A software system can be either of sequential or concurrent type. The most important part in a sequential system from the verification point of view is the relation between the system’s inputs and outputs. In other words verification of a sequential system can be summarized in checking the relation between input values and output values of the system during its operation. More formally, verification of a sequential system can be expressed in the following form: If a program starts in a specific state which satisfies a certain condition (pre-condition), then it eventually terminates and the program variables at the final state satisfy some given relation with the corresponding values at the beginning of the execution [20]. But the story in verification of concurrent softwares is different.

Many concurrent softwares use parallelism in order to make calculations more efficient. Parallelism is used to distribute a large amount of computations between different processing units to finish them in a shorter time. Concurrent programs are usually constructed from two or more sequential processes interacting with each other. Input and output values are still enough for specifying the behavior of this sequential processes but absolutely are not enough for specifying the behaviors of the concurrent system mainly because of the interactions between processes which can not only be expressed with the help of inputs and outputs. further information about verification of concurrent programs and its problems is available at [1, 16, 20, 22]

1.1 Contribution of This Thesis

Verification of concurrent programs mainly involves efforts for detecting general concurrency bugs, such as race conditions, atomicity violations, or deadlocks [8, 18, 19]. However, it is also important to directly verify concurrent programs against an abstract specification of overall functionality. For instance, a concurrent implementation of a familiar data type abstraction, such as a queue or set, could be verified to conform to a simple abstract specification of such a data type, which is typically sequential. This has been accomplished for finite-state programs and some verification tools like SPIN are already supporting it, but to our knowledge there are no approaches for handling unbounded data domains in specifications and implementations as is the case for a work-stealing double ended queue (deque) implementation.
In this project we present a technique for automatically verifying that a concurrent work-stealing deque conforms to an abstract specification of its functionality. This technique is based on the automata-theoretic approach to automated verification of finite-state programs [24]. In this approach one will specify a program by finite-state automata which we call observer. Each observer can specify one or more properties of a program. The observer automaton observes the execution of a program and accepts precisely those executions that violate the intended property. Therefore to verify a program in this approach, all the possible executions of a program are checked by the observer and if any of them are accepted, we will say the program is violating the property which is specified by that observer.

In verification terminology, property normally means an expected and correct behavior of a program. For example consider a program that implements a queue data structure. Queues usually have two main operations, namely add and remove, where add inserts a new element to the end of queue, and remove deletes one element from the front of the queue and returns it. In such a queue, a correct behavior can be that the sequence of items added to the queue must be exactly equal to the sequence of items removed from it. This is the famous FIFO (First In, First Out) property. Expressing interesting properties of a computer program as a finite-state automaton (observer) is always desirable, because automata are easily understood by computers and can be easily translated to computer language.

In our verification method we faced two problems with using the automata theoretic approach. First problem was that observer automata are finite-state which means they are not suitable for covering specifications concerning unbounded data domains. To solve this problem we extend the class of automata to automata with a finite-state control structure and a finite set of variables. But variables here can assume values form an unbounded domain.

The other problem with the extended class of automata is that they are sufficient for specifying some simple concurrent data-structures like sets, intuitively because they can hold at most one copy of each data value. But for specifying a work-stealing deque we want to keep an unbounded number of copies of each data value. For example when data value $d$ is pushed five times into the deque, we want to check and make sure that it is not popped six times or more. In order to overcome this problem we extend a data-independence argument originated from Wolper [26]. Our argument implies that for our work-stealing deque implementation (or any other data-independent program), the specification need not consider executions where a data value is pushed more than once.
By combining the above extension to the automata-theoretic approach, we obtain a very simple automated technique which is able to directly specify and verify conformance to specifications of behavioral properties over both control and data, for a work-stealing deque implementation that uses an arbitrary large data domain. Our verification in fact establishes that the implementation is linearizable [12] (the term linearizability is explained in section 2.8). In concurrent programming, linearizability is a widely accepted correctness condition for concurrent objects. It guarantees that a concurrent data structure appears to the programmer as a sequential data structure. Informally, a concurrent data-structure is linearizable if each concurrent execution of operations on the data-structure is equivalent to some permitted sequential executions. The only manual intervention needed here is the instrumentation by abstract events, which should be done at linearization points.

In order to show application of our method, we will apply it to a famous implementation of the work-stealing deque by Arora, et al [2]. We will first generate a model of this implementation. Then we will express all the desired properties of this model as observer automata. Finally we will verify the model against these properties and report the verification results.

1.2 Organization

In Section two we provide background information about the concepts we are using in the rest of this report. In Section three we talk about the related works that has been done in this area. We explain our verification method by applying it to one famous implementation of a work-stealing deque in Section four. In Section five we show the correctness of our verification method using a mathematical proof. The application of our method is presented in Section six where we apply it to the implementation of a work-stealing deque and report the results. Finally in Section seven we will write the conclusion and future work.

2 Background

This section contains background information about various concepts we are using throughout this thesis. The concepts we are explaining in this section are automatic verification, model
checking, automata theoretic approach to model checking, data independence, formal specification and work-stealing deque. At the end of this section we explain where and how these concepts are used.

2.1 Automatic Verification

The correctness of a software system is being checked by two processes. One of the processes is to check if the software is what the customer wants which is called validation. The other process is to check if the software is bug free and it matches the specification of the software which is called verification. 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. Formal verification is an alternative method for verifying a system. This method assumes access to a so called specification of the software, i.e., a description of the correct behavior of the system. It compares the specification to the actual behavior of the system [23]. There are several verification methods, and most of them are useful, but some have more advantages. Code and design reviews is one of these methods. It is good at finding (some classes of) problems but needs organization and people. Static program analysis is another one. It is an approach to analyze the source code by tools. It is completely automatic but can verify a limited set of properties (type-correctness, absence of some run-time errors, etc.) and unfortunately tools are available only for limited number of languages and properties. Testing and model checking are among the most used and famous methods for verification.

Therefore there are different approaches to verification each of which have their own limitations and problems. But one thing always desirable with verification process is automation. In other words we always like to have a tool that accepts a specific program along with its expected correct behavior (specification) as input and the tool checks automatically whether the program is satisfying that correct behavior or not. But according to computability theory [6] building such a tool for a broad range of programs is impossible. What we can do to overcome this limitations we have but are not limited to one of the following options:

- Focus on verification of smaller classes of programs which, in terms of number of states they generate, are small enough to be verified automatically using a verification tool.
• Instead of verifying the whole program, focusing on smaller yet more crucial parts of it. For example in another work of us [15] we are verifying underlying "Direct Task Stack” algorithm of Wool [7] parallel library.

• Using abstraction we can remove some of the details of the program or algorithm, resulting an an algorithm which is simple and small enough to be verified automatically by verification tools.

One way of abstraction is modeling which means generating a model from the actual program or algorithm. There are different verification tools that verify a program by checking an abstract model of it. But in order to use such model checkers one needs to do an abstraction process to generate the model. Unfortunately the abstraction process is generally a manual process because again, according to computability theory doing abstraction for all class of programs is impossible.

2.2 Model Checking

In model checking, a model of a system which describes the system’s behavior is algorithmically checked against the specification of the system. The model is usually expressed as a directed graph consisting of nodes and edges. The nodes represent the state of the program and the edges represent the possible execution which changes the program state. Usually, a set of properties is associated with each node. The properties represent the condition that should hold in a particular state of the program. Model checking is important for both validation and verification of a software system. The model of the system can be compared to customers needs for validating the system. It is also one of the techniques for model-based verification. By checking the formal model of a system against its specification, the correctness of the system can be verified. Model checking, like other verification methods has its own strengths and weaknesses. For example some of its strengths are that it can be used early in the design cycle of a software system, it is automated and it can check different kinds of properties. The weaknesses are that, model checking does not scale to very large models, constructing a model at a suitable level of abstraction is not always easy and the model must be maintained as the original system evolves.
2.3 Automata Theoretic Approach to Model Checking

The automata theoretic approach to model checking was first introduced by Moshe Y. Vardi and Pierre Wolper in 1985 [24]. In this approach one first builds a high-level and abstract model of the system which describes its behavior. Then the formal specification of the system is written. A formal specification is normally a mathematical description of the system which describes what the system is supposed to do but not necessarily how the system is supposed to do it. It is usually expressed in linear time temporal logic (PTL) [21] and is used as a measure to verify the system by checking the system’s behavior against it. If all behaviors (executions) of the system satisfy the specification, one can say that the system is working correctly, otherwise we say that the system violates its specification. Having both the model and the formal specification, the next step in this approach is to convert the formal specification to a Büchi automaton that accepts exactly the behaviors that violate this specifications. Büchi automaton is an extended version of finite-state automaton that accepts infinite input sequences [24]. Finally we provide the automatic model checker with both the model and the Büchi automaton. It will then generate all the behaviors of the model and if any of them are accepted by the automaton, the model checker will stop running and report the violation and also give a counterexample that shows how the model can violate the specification. Otherwise the model checker will verify the correctness of the model. Figure 1 shows the above explanation. In this figure $\mathcal{M}$ is the system’s model, $\Psi$ is a PTL property, $\mathcal{K}$ is the underlying transition system of $\mathcal{M}$ and $L(\Psi) = \{W | W \models \Psi \}$ is the language of all the words that satisfy the property $\Psi$. In the same way $L(\neg \Psi)$ is the language of all the words that do not satisfy $\Psi$ and $L(\mathcal{A}\neg \Psi)$ is the language of the Büchi automaton that exactly accepts those words that does not satisfy $\Psi$. Finally $L(\mathcal{K})$ means the language of all the words being generated by the underlying transition system of the model.

2.4 Formal Specification

In software verification the main goal is to answer the question: "is this program correct?". But we are only able to answer this question with respect to a given specification. Thus verification is always connected to a given specification. The Specification can even be used as a contract between the software’s customer and developer. The customer can compare the delivered software with the specification and complain if it does not satisfy the specification. The term "formal" refers to a precise and unique specification. That is because the developer and the customer need to agree on the interpretation of the specification, otherwise the customer may claim the software
is not satisfying the specification according to their understanding and the developer may deny this claim by having another interpretation of the specification. Therefore, a formalized and agreed specification is required to have a well defined syntax and precise semantics. In this way, a specification is unambiguous, and the question "Does the system satisfy the specification?" has a unique answer [20].

2.5 Data-Independence

A program is called data-independent when its behavior is independent from its input data. So when we change the input data of a data-independent program, the only thing that might change is the corresponding output data. In other words a program is data independent if the control part of the program is not making any decisions based on its input data. As a simple example suppose a program contains an \( \text{if} \) statement that checks the value of an input variable and then based on this value generates different outputs, then we say the program is not data-independent.

As we already discussed in previous sections, one successful approach to model checking of concurrent programs is the automata theoretic approach. In this approach we express the desired properties of the program in propositional temporal logic (PTL) and then create observer automata based on this expressions.

One problem with using PTL in this approach is its limited expressiveness i.e, it can only
describe finite-state structures. Most of the concurrent programs which are interesting for verification are those that operate on an infinite number of data values. Given the data-independence assumption we can say that many of the properties of a program which are stated over an infinite number of data values are equivalent to the same properties stated over a small finite set of data values. Since the data-structure that we want to verify in this thesis also has properties that are stated over an infinite number of data values, using the data-independence argument, we will be able to express them over not more than 2 or 3 data values. This is discussed in detail later in section 4.4.

2.6 Work-Stealing Deque

As mentioned in the introduction, in this project we want to create an automatic verification method for a work-stealing deque data structure. A work stealing deque is usually used for scheduling threads in a multiprogrammed multiprocessor environment [2]. In a work-stealing algorithm implementation we normally have 2 or more working threads sharing a large amount of computations with each other. Each thread in this implementation owns a deque data structure. Each thread may create small executable tasks and push them into the bottom of the deque. When a thread finishes pushing tasks into its deque it may start to pop the pushed tasks and execute them. Meanwhile other idle threads who do not have any task in their deque may become thieves and want to steal a task from this thread. In any work-stealing deque implementation stealing usually happens from one end while popping a task by the owner happens from the opposite end. In other words, the oldest task should be stolen first and the youngest task should be popped by the owner first. Therefore we can look at a work stealing deque as a stack from the popping end and as a simple queue from the stealing end.

In section 4 where we introduce our verification method, we explain the work-stealing algorithm in more detail with the help of a famous example implementation.

2.7 Linearizability

In concurrent programing, linearizability is a widely accepted correctness property. Intuitively, linearizability guarantees that the behavior of a concurrent data-structure exactly conforms to the behavior of a sequential version of the same data-structure. We illustrate this definition more by the help of an example. Consider a concurrent queue data-structure that only has
two methods: \texttt{enqueue()} and \texttt{dequeue()}. The \texttt{enqueue()} method accepts an element as an argument and inserts it at the end of the queue and the method \texttt{dequeue()} removes the first item in the front of queue and returns it. Since the queue is concurrent, it can be accessed by a different number of threads at the same time and these threads may call its methods by any order. The correct behavior of a sequential queue necessitates that all the elements are added to and removed from the queue in FIFO order. Therefore the sequence of methods \texttt{enqueue(x) enqueue(y) dequeue(x) dequeue(y)} shows a correct behavior of queue while \texttt{enqueue(x) enqueue(y) dequeue(y) dequeue(x)} shows an incorrect one (x in \texttt{dequeue(x)} is the return value of method). Now suppose two different threads are sharing a concurrent queue data-structure. The first thread runs the sequence \texttt{enqueue(x) dequeue(x)} and the second thread runs \texttt{enqueue(y) dequeue(y)}. Since these threads are scheduled by a scheduler, different interlivings between these sequences may occur. For example the first thread may start executing \texttt{enqueue(x)}, finish it but then get delayed while the second thread starts working. It executes \texttt{enqueue(y) dequeue(y)} completely and then the first thread gets preempted and continues running by executing \texttt{dequeue(x)}. The overall sequences of operations executed by these two thread will become as follows:

\[
\texttt{enqueue(x) enqueue(y) dequeue(y) dequeue(x)}
\]

which is obviously an incorrect behavior because y is removed from the queue before x. Therefore one can say that this concurrent queue is not linearizable. In a linearizable concurrent queue for each final sequence of operations, regardless of execution order, there must be an equal correct sequential sequence of operations.

2.8 Summary

In this section, we presented the basic concepts from automatic verification, model checking, automata-theoretic approach to model checking, formal specification, data-independence, work-stealing deque and linearizability.

We will use the concept of data independence in Section 4.4, where we extend our own data independence argument. In section 4.3 where we create observer automata for specifying a work-stealing deque implementation, we use concepts of automata theoretic approach to model checking, formal specification and work-stealing deque.
3 Related Work

In this section we introduce other attempts to verify a concurrent deque in implementation of a work stealing algorithm.

- First, in paper [4] the authors introduce the term ”synchronizability” as a new correctness criteria for a concurrent data structure which is weaker than the more traditional notion of serializability [17]. Then they prove that the concurrent deque implementation given in [2] is synchronizable.

- Second, Hendler et al in [9] prove that their implementation of deque in their dynamic-sized nonblocking work stealing deque is linearizable by first specifying the sequential semantics of the implemented deque and then showing that the deque is linearizable to this specification.

4 Verification Method

In this section we introduce our method by illustrating its application to an implementation of the work stealing algorithm presented by Arora et al. [2]. We will first explain how this implementation works. Then we discuss the specification of the work-stealing deque data-structure. After specification we talk about semantics and how we have created the observer automata for this verification. Finally we introduce our data independence argument.

4.1 Work-stealing deque by Arora et.al

In 1998 Arora et al. presented their implementation of the work-stealing algorithm which uses work-stealing deque data-structure. Since then, several attempts by different group of people were made to improve the efficiency of this algorithm [5, 9, 10]. Since the first version is simple and both easy to understand and implement, we used it in our work to illustrate the application of our verification method. Figure 2 is a Pseudo-code description of the main operations in this implementation. The operations are:
void pushBottom(Task* tsk)
{
    localBot = bot;
    deq[localBot] = tsk;
    localBot++;
    bot = localBot;
}

Task* popTop()
{
    oldAge = age;
    localBot = bot;
    if (localBot <= oldAge.top)
        return NULL;
    tsk = deq[oldAge.top];
    newAge = oldAge;
    newAge.top++;
    cas(age, oldAge, newAge)
    if (localBot == oldAge.top) {
        return tsk;
    }
    age = newAge;
    return NULL;
}

pushBottom(), popBottom() and popTop(). In a work-stealing environment, each thread has its own work-stealing deque data structure. In a normal run, each thread can generate new working tasks and push them into its deque. This is done using the private pushBottom operation. After a thread is done with pushing new tasks into its deque, it will retrieve them in a LIFO (“Last in First Out”) manner and execute them using the popBottom operation which is also private to the thread. Please note that by a private operation we mean an operation which can only be accessed and executed by the owner thread.

When a task is pushed into a work-stealing deque, it also becomes available to be stolen by other threads. So an idle thread which does not have any task to execute can steal a task from others by calling their popTop operation. The popTop operation in each thread is public meaning that all the other threads in the environment can call it. The popTop operation removes tasks from the victim’s deque in a FIFO manner. If the thief succeeds in stealing the task, then it will remove it from the victim’s deque and execute it. It is possible that a thief would start a popTop (stealing) operation but then get delayed and the victim itself would execute the task. In this case when the thief resumes with stealing, it will notice that the task is not on the deque anymore and therefore the popTop operation will return NULL. Also a similar situation may

1 In the original code of Arora et al. instead of Task*, they are using Thread*. In our context, the words Task and Thread can be used interchangeably.
As we can see as soon as a thread pushes a new task into the deque, that task can be accessed both by the owner and all the other threads in the environment. So there is always a risk that an owner and a thief or two thieves want to access one task at the same time. In order to avoid problems in these situations, different threads in a concurrent program use synchronization. Intuitively, synchronization means locking. In synchronization a thread first locks a shared resource to get exclusive access to it and then uses it to avoid concurrency problems. The implementation by Arora et al. uses compare-and-swap (CAS) operation for synchronization purposes. The cas instruction operates as follows:

It takes three operands: a register addr that holds an address and two other registers, old and new, holding arbitrary values. The instruction cas(addr, old, new) compares the value stored in memory location addr with old, and if they are equal, the value stored in memory location addr is swapped with new. In this case, we say the cas succeeds. Otherwise, it loads the value stored in memory location addr into new, without modifying the memory location addr. In this case, we say the cas fails. The whole operation is performed atomically with respect to all other memory operations [2]. Figure 3 shows a deque object with its variables and data structures.

As we can see in the picture, the deque object is using an extra variable tag which along with top pointer are encapsulated in age. This variable is used to avoid one famous problem.
of concurrent data structures called ABA. More information about this problem and how it is solved using tag is presented at section 3.3 of [2]. The top pointer keeps track of the head of deque and can be accessed and changed by thieves while the bot pointer keeps track of the bottom of deque and can only be accessed by the owner thread.

4.2 Specification

In our verification method we state the specification of the work-stealing deque data structure as following:

"A sequence of pushBottom, popBottom and popTop operations including their input parameters and return values which are performed by any execution of the program in Figure 2 should conform to a valid and correct sequential behavior of a work-stealing deque."

To specify, we first instrument the program to generate abstract events. By abstract events we simply mean the event name along with its input and output parameters. So for the work-stealing program, we use the abstract events pushBottom(d), popBottom(d), popTop(d), popBottom(empty) and popTop(empty) where d is a data value in D and D is the data domain of our program. Data value d in the abstract event pushBottom(d) is the input parameter while in popBottom(d) or popTop(d) it is the output value of operation. popBottom and popTop will return empty whenever a thread wants to synchronize with its own task or steal a task from another thread, where it finds that the deque is empty, the task is already stolen or it is already executed. We let the statement, at which the operation performed by the method takes global effect (linearization point of operation), generate a corresponding abstract event. As we already explained in Section 2.7, linearization point of an operation or function is some instant between its invocation and its response when the operation appear to occur instantly [11]. Typically, these linearization points are at program locations that update a globally visible shared variable [25]. In the code of figure 2 the linearization point of pushBottom is where localBot is assigned to bot, we replace line 6 of method pushBottom by atomic{bot=localBot; emit(pushBottom(d))}. Here emit simply means we record the abstract event.

Successful popBottom operation linearizes when localBot decrements and after that is assigned to bot or when bot is set to 0. Therefore we change line 7 in this operation to the following lines of code:
Automatic verification of work-stealing deque implementation

atomic{
    bot = localBot;
    emit(popBottom(deq[localBot]))
}

and we also change line 12 to the following block of code:

atomic {
    bot = 0;
    emit(popBottom(tsk))
}

Unsuccessful popBottom is linearized when it detects that the deque is empty. This is line 4 and line 17 of figure 2. Therefore we change line 4 to

```c
if(localBot == 0)
    emit(popBottom(empty)); return NULL;
```

And line 17 to

```c
atomic{
    if (oldAge == newAge)
        return tsk;
    else
        emit(popBottom(empty));}
```

A successful popTop() call is linearized at the point when a successful CAS() took place. Therefore we replace line 10 with the following lines of code.

```c
atomic{ cas(age, oldAge, newAge);
    if (oldAge == newAge)
        emit(popTop(tsk))
}
```

Each unsuccessful popTop call is linearized at the point where the method detects that the deque is empty. We will explain this case in more detail in Section 6.5.

With this instrumentation, each execution of the program generates a sequence of abstract events. The complete instrumented version of the code in figure 2 is available in Appendix A. As we already mentioned in the introduction section, this is the only manual part of the verification method and that is because finding the linearization points of program or algorithm is not easy to automate. It requires good knowledge and understanding of the algorithm and experience in detecting the linearization points.
After instrumentation, each execution of the program will generate a sequence of abstract events. These sequences are called the traces of the program. Having the traces generated, the next step is to specify them by an observer automaton. Each observer automaton will be responsible for checking the correctness of one or more properties. For example we can build an observer automaton that checks (observes) the FIFO property of a queue. Then we generate traces from a queue program and feed the automaton with these traces. The automaton will observe the traces and move to an accepting location if any trace violates the property. But if the automaton does not go to an accepting state for any of the traces, then we can say the program preserves the FIFO property. We will do exactly the same thing with the work-stealing program. First we define the desirable correctness properties of the work-stealing algorithm and then for each property we create an observer automaton.

The variables of the observer can initially assume any values in the data domain. This allows observers to specify properties that are universally quantified over data values: by checking whether there are some values of the variables that violate the property. As we already explained in the background section observers are able to directly specify simple concurrent data structures like sets. However, the allowed sequences of operations of work-stealing deque and many other data structures like queues and stacks, cannot be characterized by an observer with a finite set of data variables, e.g., the observer must be able to count the number of copies of a data value that have been inserted but not removed. Later in subsection 4.3 we will show how to remedy this problem by extending a data independence argument.

4.3 Program Semantics and Observers

In this project we concisely describe program syntax by a set of control locations \( Q \) and by representing the method body by a set of transitions \( q \xrightarrow{s} q' \), which denote that a thread can move from control location \( q \) to \( q' \) by executing the command or atomic statement \( s \). Here we briefly sketch how operational semantics for programs can be described in a natural way. We use the term configurations to show the different states of a program. In a concurrent program, each configuration is defined by the value of global variables, local state of all threads, state of the data structure which is shared between threads, etc. We use \( \gamma \) to denote configurations and \( \gamma_{\text{Init}} \) to denote the initial configuration of the program. Program behavior is formalized by a labeled transition relation on the set of configurations. Elements of the transition relation are either of form \( \gamma \xrightarrow{t(d)} \gamma' \), denoting that the configuration can change from \( \gamma \) to \( \gamma' \) while emitting
the abstract symbol \( l(d) \), or of form \( \gamma \xrightarrow{\tau} \gamma' \) in case where no abstract symbol is emitted. An execution of a program is a sequence of transitions \( \gamma_{\text{init}} \xrightarrow{l_1} \gamma_1 \xrightarrow{l_2} \ldots \xrightarrow{l_n} \gamma_n \). A trace of a program is the sequence of non-\( \tau \) labels on transitions in some execution of the program. Traces capture an abstract view of program executions. A program is specified in terms of properties of its traces. We now present how to use observers for specifying safety properties of programs.

Assume a set of event types. An observer is a tuple \( A = \langle S, s_0, F, Z, \rightarrow_A \rangle \) where \( S \) is a finite set of observer locations, \( s_0 \in S \) is the initial location, \( F \) is the set of accepting locations, \( Z \) is a finite set of observer variables, and \( \rightarrow_A \) is a finite set of transitions. The observer variables assume values in \( \mathbb{D} \) (the data domain of the program). Initially, they can assume any value in \( \mathbb{D} \). Each transition is a tuple of form \( s \xrightarrow{l(\overline{p};g)} s' \) where \( s, s' \in S \) are locations, \( l(\overline{p}) \) is a parameterized event, and the guard \( g \) is a Boolean combination of equalities over formal parameters \( \overline{p} \) and observer variables \( Z \). An observer configuration is a pair \( \langle s, \vartheta \rangle \), where \( s \in S \) is an observer location, and \( \vartheta : Z \mapsto \mathbb{D} \) maps each observer variable to a value in the data domain \( \mathbb{D} \). An observer step is a triple \( \langle s, \vartheta \rangle \xrightarrow{\overline{d}} \langle s', \vartheta \rangle \) such that there is a transition \( s \xrightarrow{l(\overline{p};g)} s' \) for which \( g[\overline{d}/\overline{p}] \) is true. A run of the observer on a trace \( \delta = l_1(\overline{d}_1)l_2(\overline{d}_2)\ldots l_n(\overline{d}_n) \) is a sequence of observer steps \( \langle s_0, \vartheta \rangle \xrightarrow{l_1(\overline{d}_1)} \ldots \xrightarrow{l_n(\overline{d}_n)} \langle s_n, \vartheta \rangle \) where \( s_0 \) is the initial observer location. The run is accepting if \( s_n \) is accepting. A trace \( \delta \) is accepted by \( A \) if \( A \) has an accepting run on \( \delta \). Thus, a program satisfies the property represented by observer \( A \) if no trace of the program is accepted by \( A \). To verify that no trace of the program is accepted by \( A \), we form, as in the automata-theoretic approach \([24]\), the cross product of the program and the observer, synchronizing on abstract events, and check that this cross product cannot reach a configuration where the observer accepts.

In this project we first define 5 safety properties for a work-stealing deque data structure which each of them can be specified by an observer automaton.

Then after we have presented the properties and their corresponding observer automata we can present and prove a claim that says “a work-stealing deque implementation is a correct implementation based on the definition of a work-stealing deque, iff all differentiated traces of that implementation satisfy all these properties i.e, the cross product of the program and the observer automatons can not reach a configuration where the observer accepts”. A trace is said to be differentiated if all its input occurrences are pairwise different. For example the trace \( \text{push}(1)\text{push}(2)\text{pop}(2)\text{push}(3) \) is differentiated while the trace \( \text{push}(1)\text{push}(2)\text{pop}(2)\text{push}(1) \) is not as input data value 1 is repeated.
We will express and prove this claim after explaining the safety properties of a work-stealing deque and their corresponding observer automata.

Here we present the correctness properties of a work-stealing deque. For convenience we give each property a name:

**First property:** "DUPLICATE" d can not be stolen twice, popped twice or both stolen and popped.

This property ensures that in a differentiated trace of work-stealing deque implementation a task can be only stolen or popped once and it can not be stolen twice, popped twice or both stolen and popped. Figure 4 shows the observer automaton for specifying this property.

**Second property:** "LIFO" If \(d_2\) is pushed after \(d_1\), then \(d_2\) is popped before \(d_1\).

This property ensures the stack behavior of a work-stealing deque. As we already mentioned a work-stealing deque from one end behaves like a stack and from the other end behaves like a queue. Figure 5 shows the observer automaton for this property.

**Third property:** "FIFO" If \(d_2\) is pushed after \(d_1\), then \(d_1\) is stolen before \(d_2\).
This property ensures the queue behavior of a work-stealing deque. Figure 6 shows the observer automaton for this property.

\[
g = (p \neq d1 \land p \neq d2 \land p \neq empty) \
g' = (p \neq d1 \land p \neq d2)
\]

**Forth property:** "LOSS" A pop or steal operation must not return empty if \(d\) has been pushed but not yet been popped or stolen. The observer automaton for specifying this property is shown in figure 7.

\[
g = (p \neq d \land p \neq empty) 
\]

**Fifth property:** "CREATION" \(d\) can not be popped or stolen before it is pushed. This property ensures that a task must be pushed into the deque before it can be popped or stolen. Figure 8 shows the observer automaton for this property.

\[
g = (p \neq d \land p \neq empty)
\]
4.4 Our Data Independence Argument

Many data structures like queues, stacks, etc. can not be specified by observers with a finite set of states. That is because for some correctness properties of these data structures we need to count the number of times a data variable is visited. For example in a stack implementation as we know, we can not push data value \( d \) ten times and then pop it eleven times. So this means the observer for checking this property should be able to count the number of times a data value has been pushed. Generally this is not possible for observers with a finite set of data variables. In this section, we explain a data-independence argument, originating from Wolper [26], that will allow stacks, queues, work-stealing deques and other structures to be specified by observers.

In this argument we divide the occurrences of data values in traces into input occurrences and output occurrences. For example in the work-stealing deque program the parameters of the \texttt{pushBottom}(d) operation are input occurrences and return values from \texttt{popBottom()} and \texttt{popTop()} operations are output occurrences. Note that the same data value can appear both in input occurrences and output occurrences. A renaming is any function \( f : D \mapsto D \) that renames data-values to data-values. The renaming of a trace \( \sigma \) with some renaming \( f \), denoted \( f(\sigma) \), is then obtained by replacing each data value \( d \) in \( \sigma \) by \( f(d) \).

We say that a set of traces \( \xi \) is data-independent if for any trace \( \sigma \in \xi \) both of the following conditions hold:

1. \( f(\sigma) \in \xi \) for any renaming \( f \)
2. there exists at least one differentiated trace \( \sigma_d \in \xi \) such that \( f(\sigma_d) = \sigma \) for some renaming \( f \)

We say that a program is data-independent if the set of its traces is data-independent. In a similar manner, a correctness property is data-independent if the set of allowed traces is data-independent. It follows from these definitions that a data-independent program satisfies a data independent property if its differentiated traces satisfy the property. An observer for a data-independent property therefore needs only consider differentiated traces: whenever a data value is input twice in a trace, the observer can stop checking, since the trace will anyway be ignored. By having this argument now we can say that differentiated traces of a work-stealing deque can be completely specified by observers with a small number of variables as long as the implementation we want to verify is data-independent.
5 Correctness of Verification Method

In this section we will show by mathematical proof the correctness of our verification method.

According to our data-independence argument now we are able to show that all differentiated traces of a work-stealing deque can be specified by observers. This is expressed in Theorem 1. From here and onward, for convention, we use push instead of pushBottom, pop instead of popBottom and steal instead of popTop operations. Furthermore we use $p_1$ for property "DUPLICATE", $p_2$ for "LIFO", $p_3$ for "FIFO", $p_4$ for "CREATION" and $p_5$ for "LOSS".

Each of the properties $p_1 \ldots p_5$ can be specified by an observer with a finite set of control locations and one or two local variables which are non-deterministically initialized to arbitrary values. If the property is violated by some specific data values $d_1$ and $d_2$, then there is some run of the observer, in which the initial values of the variables are $d_1$ and $d_2$, which leads to an accepting state.

Before stating theorem 1 we need to present the following two definitions and four lemmas which later we will use in our proof.

**Definition 1. Lexicographical Ordering $\prec$ over traces of a work-stealing deque**
Suppose $\sigma_a$ and $\sigma_b$ are traces of a work-stealing deque. Then we say $\sigma_a$ is smaller than $\sigma_b$ iff:

1. $|\sigma_a| < |\sigma_b|$ or
2. $|\sigma_a| = |\sigma_b|$ and the number of pushes before a specific steal operation in $\sigma_a$ is less than the number of pushes before the same steal operation in $\sigma_b$, where $|\sigma|$ means the length of trace $\sigma$.

In the next definition we define inductively the correct trace(behavior) of a work-stealing deque.

**Definition 2. Inductive Definition of a Correct Trace of a Work-stealing Deque**

*Base:* A trace of length zero is always a correct trace of a work-stealing deque.

The trace $\sigma$ is a correct trace of a work-stealing deque iff it is in one of the following forms:

- $\text{pop}(\text{empty})\sigma_n$ iff $\sigma_n$ is correct
- $\text{steal}(\text{empty})\sigma_n$ iff $\sigma_n$ is correct
- $\sigma_n\text{push}(d)$ iff $\sigma_n$ is correct
- $\sigma_i\text{push}(d)\text{pop}(d)\sigma_j$ iff $\sigma_i\sigma_j$ is correct
- $\text{push}(d)\text{steal}(d)\sigma_{n-1}$ iff $\sigma_{n-1}$ is correct
-σ_i\push(a)\push(b)\steal(c)σ_j \iff \sigma_i\push(a)\steal(c)\push(b)σ_j \text{ is correct.}

**Lemma 1.** If the trace σ of length n(σ_n) satisfies $\bigwedge_{i=1}^{5} p_i$ then any prefix trace of σ of any length will also satisfy $\bigwedge_{i=1}^{5} p_i$.

**Proof.** We will prove this lemma by induction on difference between the size of prefix trace and n which is the length of σ. We show this difference by k.

**Base:** If k=0 then $\sigma_{n-k} = \sigma_n$ and we know that $\sigma_n \models \bigwedge_{i=1}^{5} p_i$ satisfies the properties.

**Assumption:** If $0 < k < n$ then $\sigma_{n-k} \models \bigwedge_{i=1}^{5} p_i$.

Now we want to show that $\sigma_{n-k+1} \models \bigwedge_{i=1}^{5} p_i$.

Suppose $\sigma_{n-k}$ does not satisfy $\bigwedge_{i=1}^{5} p_i$. We can write $\sigma_{n-k}$ as $\sigma_{n-k+1}\alpha$ where α can be any of the following possible operations of a work-stealing deque:

- push(d)
- pop(d)
- steal(d)
- pop(empty)
- steal(empty)

If $\sigma_{n-k+1}$ does not satisfy $\bigwedge_{i=1}^{5} p_i$ then adding any one of the above operations to the end of trace will not fix the problem and hence $\sigma_{n-k}$ will also violate $\bigwedge_{i=1}^{5} p_i$ which is a contradiction to proof assumption □.

**Lemma 2.** If $\sigma_n \models \bigwedge_{i=1}^{5} p_i$, where $\sigma_n$ is a differentiated trace of length n of a work-stealing deque, then any trace of size $k < n$ which is extracted from $\sigma$ by removing any number of consequent push(d) pop(d) will also satisfy $\bigwedge_{i=1}^{5} p_i$.

**Proof.** We prove this by induction on the number of push(d) pop(d) pairs removed from the trace. We name this number $\alpha$:

**Base:** If $\alpha = 0$ then $\sigma = \sigma_n \models \bigwedge_{i=1}^{5} p_i$. 
Assumption: \( \alpha = k - 1 \) and \( \sigma \) is the trace \( \sigma_n \) after removing \( k - 1 \) number of \( \text{push}(d) \text{pop}(d) \) pairs. \( \sigma = \sigma_a \text{push()} \text{pop()} \sigma_b \models \bigwedge_{i=1}^{5} p_i \).

Now when \( \alpha = k \) we remove one more pair of \( \text{push()} \text{pop()} \) from \( \sigma \) so \( \sigma = \sigma_a \sigma_b \).

Suppose \( \sigma = \sigma_a \sigma_b \) does not satisfy \( \bigwedge_{i=1}^{5} p_i \). Then if we add a new pair of \( \text{push()} \text{pop()} \) to \( \sigma \) so that again \( \alpha \) becomes \( k - 1 \) and \( \sigma = \sigma_a \text{push()} \text{pop()} \sigma_b \) but obviously adding a pair of \( \text{push}(d) \text{pop}(d) \) operations will not help \( \sigma \) to satisfy the properties and hence it will violate the property which is in contradiction to the proof’s assumption. \( \square \)

**Lemma 3.** If the differentiated trace \( \sigma = \sigma_i \text{push}(d) \text{push}(d') \text{steal}(d'') \sigma_j \) satisfies \( \bigwedge_{i=1}^{5} p_i \) then the trace \( \sigma' = \sigma_i \text{push}(d) \text{steal}(d'') \text{push}(d') \sigma_j \) also satisfies \( \bigwedge_{i=1}^{5} p_i \).

**Proof.** We can easily prove this lemma using contradiction.

Suppose \( \sigma' \not\models \bigwedge_{i=1}^{5} p_i \). By swapping the places of \( \text{steal}(d'') \) and \( \text{push}(d') \) we will reach \( \sigma \) but this change will not affect the violation of properties by \( \sigma' \). i.e if \( \sigma' \not\models \bigwedge_{i=1}^{5} p_i \), then swapping \( \text{steal}(d'') \) and \( \text{push}(d') \) will not help \( \sigma' \) to satisfy the properties and hence \( \sigma \) also will not satisfy them and that is contradiction to the lemma’s assumption. \( \square \)

**Lemma 4.** If the trace \( \sigma_n \models \bigwedge_{i=1}^{5} p_i \), the trace \( \sigma \) which is a postfix trace of \( \sigma_n \) will also satisfy \( \bigwedge_{i=1}^{5} p_i \).

Since the proof of this lemma is also simple and exactly similar to proof of lemma 1 we will not explain it here.

Now that we have all the lemmas we need we will explain and prove the following theorem.

**Theorem 1.** All differentiated traces of a work-stealing deque program satisfy \( \bigwedge_{i=1}^{5} p_i \) iff they belong to definition 2.

**Proof.** We will do the proof by induction on length of trace \( \sigma \) which is a differentiated trace of a work-stealing deque implementation. In this proof \( i \) in \( \sigma_i \) means the length of trace \( \sigma \).

**Base:** \( \sigma_0 \models \bigwedge_{i=1}^{5} p_i \) then \( \sigma_0 \in \text{Definition} 1 \)

**Assumption:** \( \sigma_{n-1} \models \bigwedge_{i=1}^{5} p_i \) then \( \sigma_{n-1} \in \text{Definition} 1 \)
We want to show that if $\sigma_n \models \bigwedge_{i=1}^{5} p_i$ then $\sigma_{n-} \in \text{Definition}1$ so we write $\sigma$ as $\sigma_{n-1} \alpha$, where $\alpha$ is one of the following operations of a work-stealing deque:

- push(d)
- pop(d)
- steal(d)
- pop(empty)
- steal(empty)
if $\alpha = \text{push}(d) \Rightarrow \sigma_n = \sigma_{n-1}\text{push}(d) \Rightarrow \{\text{lemma1}\} \sigma_n \in \text{definition1}$

if $\alpha = \text{pop}(d) \Rightarrow \sigma_n = \sigma_{n-1}\text{pop}(d) = \{\text{lemma1, definition1}\}$

1- $\sigma_{n-2}\text{push}(d')\text{pop}(d)$

or

2- $\text{pop(\text{empty})}\sigma_{n-2}\text{push}(d)$

or

3- $\text{steal(\text{empty})}\sigma_{n-2}\text{push}(d)$

or

4- $\sigma_i\text{push}(d')\text{pop}(d')\sigma_j\text{pop}(d); i + j = n - 3$

or

5- $\sigma_i\text{push}(a)\text{push}(b)\text{steal}(c)\sigma_j\text{pop}(d); i + j = n - 4$

or

6- $\text{push}(d')\text{steal}(d')\sigma_{n-3}\text{pop}(d)$

Now we will study the above cases one by one:

In case 1: $\sigma_{n-2}\text{push}(d')\text{pop}(d) = \{\text{assumption}\} \sigma_n = \bigwedge_{i=1}^{5} p_i$

$\sigma_{n-2}\text{push}(d)\text{pop}(d) = \sigma_{n-2}\text{push}(d)\text{pop}(d)\sigma_0 \in \text{definition1}$

In case 2: $\sigma_n = \text{pop(\text{empty})}\sigma_n - 2\text{pop}(d) = \{\text{lemma4}\} \text{pop(\text{empty})}\sigma_{n-1} \in \text{definition1}$

In case 3: $\sigma_n = \text{steal(\text{empty})}\sigma_n - 2\text{pop}(d) = \{\text{lemma4}\} \text{steal(\text{empty})}\sigma_{n-1} \in \text{definition1}$

In case 4: $\sigma_n = \sigma_i\text{push}(d')\text{pop}(d')\sigma_j\text{pop}(d)$

$\{\text{from lemma2 we have}\} \sigma_i\sigma_j\text{pop}(d) = \bigwedge_{i=1}^{5} p_i \Rightarrow \sigma_i\sigma_j\text{pop}(d) \in \text{definition1} \Rightarrow \{\text{definition1}\} \sigma_n \in \text{definition1}$

In case 5: $\sigma_n = \sigma_i\text{push}(a)\text{push}(b)\text{steal}(c)\sigma_j\text{pop}(d)$

$\{\text{from lemma3 we have}\} \sigma_i\text{push}(a)\text{steal}(c)\text{push}(b)\sigma_j\text{pop}(d) = \bigwedge_{i=1}^{5} p_i \Rightarrow$

$\{\text{inductive assumption}\} \sigma_i\text{push}(a)\text{steal}(c)\text{push}(b)\sigma_j\text{pop}(d) \in \text{definition1} \Rightarrow \{\text{definition1}\} \sigma_n \in \text{definition1}$

In case 6: $\sigma_n = \text{push}(d')\text{steal}(d')\sigma_{n-3}\text{pop}(d)$. $\{\text{From lemma4 we have}\}$

$\sigma_{n-3}\text{pop}(d) = \bigwedge_{i=1}^{5} p_i \Rightarrow \{\text{inductive assumption}\}$
$\sigma_{n-3} \text{pop}(d) \in \text{definition1}$

$\Rightarrow \{\text{definition1}\} \sigma_n \in \text{definition1}$.

In a similar way we can continue the proof when $\alpha$ is steal($d$), pop(empty) or steal(empty).

$\Box$. 

$\Box$
6 Implementation

In this work we have modeled the implementation of the Arora algorithm in the Spin model checker to apply our verification method and observe the results. We tried to build an accurate Promela model which is as close to the algorithm’s implementation as possible and at the same time as summarized and abstract as possible to avoid explosion of state space.

In this section we will first explain the issues we faced in modeling the implementation of the algorithm. Then we will explain the instrumentation we have done in each method to be able to verify the correctness properties. Finally we will explain how we have checked each correctness property.

6.1 State Space Growth

In the main implementation we have a number of threads each with its own deque of tasks. Each thread might have a task to execute or it might have no executable task in which case it tries to steal tasks from other threads. So a thread can be a victim or a thief at different periods of time during one execution. So if we want to model threads exactly according to implementation, we need to define a deque data structure and $\text{top}$ and $\text{bot}$ pointers for each thread which results in a huge state space. Furthermore we need mechanisms for each thread to decide when the thread becomes a thief and when it is a victim and also a mechanism to decide from which victim a thief should steal a task.

In our model we have decided to have only one victim thread and only one deque which is accessible by all threads. We also have a number of thief threads. The victim thread tries to push and then pop tasks into/from the deque while thieves are trying to steal tasks before they are popped by the victim. This way we only have one deque data structure and only one bot and top pointer which will help to save state space.

6.2 Modeling Threads-Modeling Methods

Another issue we had was how to model threads and methods. For the threads the only way to model them in Promela is by using processes. Each process is a piece of code that can be executed independently. Processes like methods or functions in regular programming languages
Automatic verification of work-stealing deque implementation

...can have input arguments but they don’t have return values (though we overcame this limitation by using the method presented in [14]).

For modeling methods, Spin gives us two options: Processes again or inline calls. Inline calls are nothing but a part of code that can be written separately outside the main code, but when it is executed, it will be inlined exactly after the line where it is being called [3, 13]. Inline calls in Promela can not have a return value or can not have local variables. Because of this limitations we decided to use processes for modeling methods. So in total we have four processes: pushBottom, popBottom, popTop and victim.

We have defined a process called victim to execute pushBottom and popBottom on the deque. We also defined pushBottom and popBottom as two independent processes. For each thief thread we have defined a process called popTop which inside it we execute the code of the popTop method. That is because in our model popTop is the only method a thief can execute.

6.3 Unbounded Data Domain-Input Values

Another issue that we had to consider in our model was how to model unbounded number of input data items that can assume a value from an unbounded data domain. We already solved this problem using our data independence argument (section 2.5). According to our data independence argument we will only need three different data values to check the correctness properties. In this model we defined three colors to be used as data values that can be pushed into or popped from deque. The colors are Red, Blue and White. So in the model we push Red, Blue or white instead of real tasks.

In order to cover all possible executions of the algorithm and on the other hand avoid state-space explosion, we limit the number of ways these three different colors can be pushed into the deque. We use one of the following patterns:

- \textit{white}*
- \textit{white}^* \textit{red}^1 \textit{white}^*
- \textit{white}^* \textit{red}^1 \textit{white}^* \textit{blue}^1 \textit{white}^*
The white color here does not have any value for us from the verification point of view. It is only used to show the other possible values other than red and blue. The complete code of our model is presented in Appendix B.

### 6.4 Applying Verification Method to Model

When we defined different processes that we had in our model and the data structures we needed, we had to apply our verification method to model-check the correctness properties. According to our verification method we had to first specify the linearization points of the algorithm in the model. Since we modeled the three main operations of the algorithm exactly as they are in the implementation, the linearization points of the model are also exactly according to linearization points of the algorithm.

Having the linearization points specified in the model, the next step in the verification method was to emit an abstract event at the linearization point of each method to create a trace of abstract events and then create our Büchi automata to verify each property. In Promela Büchi automata are defined using never claims [3]. What we do when we define a never claim is that we define our correctness property in LTL logic. Spin will automatically generate the corresponding never claim for us. Expressing the correctness properties in LTL logic is not always straight-forward. It needs good knowledge and experience in LTL logic. One also needs to ensure that the defined LTL formula is exactly checking the desired property. Furthermore the method needs to define new data structures to save traces of abstract events which will be used by never claims to verify the properties.

In order to avoid these difficulties we decided to implement our method in an easier way. What we did instead was that first we defined four auxiliary variables which were used only for verification purposes. The variables are \texttt{pushRed}, \texttt{pushBlue}, \texttt{popRed} and \texttt{popBlue}. \texttt{pushRed} and \texttt{pushBlue} are boolean variables that are used to show whether the data value red or blue are pushed yet or not. We set these variables at the linearization point of \texttt{pushBottom}. \texttt{popBlue} and \texttt{popRed} are counters which are used to count the number of times red or blue is popped. We increment these counters at the linearization point of \texttt{popBottom} and \texttt{popTop} and we use them for checking properties like ”LIFO” and ”DUPLICATE”.

Finally the way we verify the properties using our model is that at the linearization point of each method, we check the related correctness properties. We also update the auxiliary variables
at the appropriate places in the code. For popTop this is at the linearization point of successful operation. For popBottom this is where the operation returns a task either after it detects that the deque is not empty or after a successful CAS operation. Please note that we do not update auxiliary variables at the linearization points of successful popBottom. This is because the result of operation is not known at those linearization points. When we decrement bot, result of operation depends on the check that follows (line 10 of figure 2) and when we set bot to 0, result of operation depends on the CAS operation that follows in line 16. In the upcoming sections we will explain for each method how we have instrumented the code and how we have performed our verification.

6.5 Checking properties

In the following sections we will explain how we have checked different properties in our model. Our verification is performed inside our methods. Inside pushBottom method at its linearization point, we set the pushRed and pushBlue flags to true. In the next sections we will explain what we did in the popBottom and popTop methods to verify each correctness property. In general what we do in these methods is that at the linearization point of the method, we check the related correctness property (ies). We also update the auxiliary variables at the appropriate places inside the code.

6.5.1 DUPLICATE

The DUPLICATE property is to make sure that no data value is popped twice neither by popBottom nor by popTop or both. To check this property, we have defined two counter variables; popRed and popBlue. We increment these counters at the completion (return) point of popBottom and popTop and check them using the inline method checkDuplicate. The code for checkDuplicate is presented at figure 9. Later when the popBottom or popTop operations are completed we increment the counters and assert the result of checkDuplicate inline.

As we can see in this inline method we first check what data value (color) is being popped. Based on the color we check the corresponding counter and if the counter at this point has a higher value than 1 this means that the property is violated.
inline checkDuplicate(task)
{
  if ::task == red -> assert(popRed <= 1);
  ::task == blue -> assert(popBlue <= 1);
  ::else -> skip;
  fi;
}

FIGURE 9: The checkDuplicate inline

6.5.2 FIFO

According to the specification of Arora algorithm, this property states that the first task that is pushed to the work stealing deque must be the first task that is being stolen by a thief thread. Therefore we need to check this property somewhere inside the popTop process. We check this property at the linearization point of a successful popTop using the inline method checkFifo().

inline checkFifo(task)
{
  if task == blue -> assert(popRed == 1);
  fi;
}

FIGURE 10: The code to check FIFO property

As we mentioned at the beginning of the implementation section, we push red data value before blue. So when we are in popTop method and we have pushed both red and blue, we must first popTop red and then popTop blue. What we are checking in the code of figure 10 is that when we are popping blue we make sure that red is already popped by assertion (popRed == 1). If this assertion fails, it means the property is violated.

6.5.3 LIFO

This property states that the last task that is pushed to the work stealing deque must be the first task that is being popped by the owner (victim) thread. Therefore we need to check this property somewhere inside the popBottom process. The LIFO property should be only checked in case of successful popBottom which is when popBottom returns a data value and not when the deque is empty and popBottom returns NULL. In order to check this property, we have written an inline method called checkLifo. Since in case of successful popBottom we have two linearization points, we need to call this inline check in both of them. The code for checkLifo is presented in figure 11.
**Automatic verification of work-stealing deque implementation**

```c
inline checkLifo(task){
    if (task == red -> IF pushBlue -> assert(popBlue == 1) FI; FI; 
}
```

**FIGURE 11:** The code to check LIFO property

In the `checkLifo` inline, we first check what data value is being popped. If it is blue we don’t care because we know that blue can only be pushed after red (this is explained at beginning of current section) and hence the order is preserved. However if the popped data value is red, then there are two scenarios based on our model; either blue is not pushed at all in which case we don’t care and the property is preserved or blue is pushed which means it must be popped already otherwise the tasks are being popped out of order and the LIFO property is being violated. For complete code of `popBottom` method in our model please refer to appendix section.

### 6.5.4 CREATION

The CREATION property simply states that a task can not be popped if it is not already pushed. This property must be checked both in a successful `popBottom` and successful `popTop` processes because in both, this property might be violated. This property is checked using the inline method `checkCreation`. The code for `checkCreation` is shown in figure 12.

```c
inline checkCreation(task){
    if (::task == red -> assert(pushRed ); ::task == blue -> assert(pushBlue); ::else -> skip; fi 
}
```

**FIGURE 12:** How we check CREATION property

### 6.5.5 LOSS

The last property we need to check in our model is LOSS. This property states that if a data value is pushed to the deque but not already popped, then neither `popBottom` nor `popTop` can not return NULL. In other words a data value should not be lost during the operation of the algorithm. We need to check this property at the linearization point of unsuccessful `popBottom` or `popTop` because that is where the operations are returning NULL and we want to make sure that when the operation returns NULL there is actually no data value remaining in the deque. In
Automatic verification of work-stealing deque implementation

The popBottom method, the linearization point for returning NULL (unsuccessful popBottom) is where it detects that the deque is empty. So we check the LOSS property using inline method checkLoss once at the point where the operation detects that the deque is empty (line 10 of figure 2) and another time after unsuccessful CAS operation. The code for checkLoss is shown in figure 13. Checking LOSS property inside popTop method is not as easy as checking it for popBottom. This is because the linearization point of unsuccessful popTop is non-fixed [25]. A non-fixed linearization point means that the linearization point of an operation might lay inside the code of another operation being executed in parallel. In case of popTop, the linearization point of returning NULL is either before a successful pushBottom operation or after a successful pushBottom operation followed by a successful popBottom operation. When the linearization point is non-fixed it needs manual instrumentation of the code and extra effort to check the property. Since checking properties that are related to an event that has non-fixed linearization point is not the main purpose of this thesis, we leave checking LOSS property for popTop operation as work to be done in the future.

```c
inline checkLoss(){
    IF (pushRed) -> assert(popRed >= 1) FI;
    IF (pushBlue) -> assert(popBlue >= 1) FI;
}
```

FIGURE 13: Checking LOSS property

6.6 Verification Results

After adding all the assertions, we verified the model using a random number of pushBottom and popBottom processes executed by the victim process followed by up to three popTop (thief) processes. We run our verification on a computer with 3GB of RAM memory and 1GB of memory available for Spin. The maximum search depth was set to 1000 and we used compression to minimize the number of states. For the mentioned settings, Spin was able to verify our model in approximately one minute and none of the properties were violated. Furthermore we increased the number of thieves to four but Spin was not able to verify the model exhaustively. However in Supertrace/Bitstate mode it was able to verify the model again in approximately one minute and all the properties were preserved.

In order to check our model and see if it can find violation of properties in case of a mistake in the algorithm, as an example we altered the algorithm and observed the result.
In the example we removed the statement "bot = 0" (line 12 of popBottom code in 4.1) to see how the model behaves when bot is not set to zero in the right place. As a result Spin reported a violation of the Duplicate property where red was pushed once but was popped twice!

7 Conclusion and Future Work

In this thesis we have presented a new verification method that claims; for a specific implementation of an algorithm, given that the algorithm is data independent, our method can show that the algorithm is correct with respect to an abstract specification of its overall functionality. With the help of our data independence argument we have also shown that with a small number of data values and therefore an automata with finite number of states we can verify the implementation of the given data-independent algorithm.

Our method currently is limited to parallel programs with bounded number of threads. So a possible future work is to extend this method to be able to handle unbounded number of threads. Furthermore the method can be extended to support heap manipulating programs (e.g., programs that are using linked list data structure) using techniques like shape analysis.
Bibliography


Automatic verification of work-stealing deque implementation


Appendix A

1 Instrumented Pseudo-code of Arora et al. Work-stealing Algorithm

1 void pushBottom(Task* tsk)
2 {
3    localBot = bot;
4    deq[localBot] = tsk;
5    localBot++;
6    atomic{bot=localBot;
7        emit(pushBottom(tsk))};
8 }

1 Task* popTop()
2 {
3    oldAge = age;
4    localBot = bot;
5    if (localBot <= oldAge.top)
6        return NULL;
7    tsk = deq[oldAge.top];
8    newAge = oldAge;
9    newAge.top++;
10   atomic{ cas(age, oldAge, newAge);
11      if (oldAge == newAge)
12         emit(popTop(tsk));
13      return tsk;
14    }
15   return ABORT;
16 }

1 Task* popBottom()
2 {
3    localBot = bot;
4    atomic{
5      if(localBot == 0)
6        emit(popBottom(0));
7        return NULL;
8    }
9    localBot--;
10   atomic{
11      bot = localBot;
12      emit(popBottom(deq[localBot]));
13    }
14    tsk = deq[localBot];
15    oldAge = age;
16    if (localBot > oldAge.top)
17        return tsk;
18   atomic {
19      bot = 0;
20      emit(popBottom(tsk));
21    }
22    newAge.top = 0;
23    newAge.tag = oldAge.tag + 1;
24    if (localBot == oldAge.top) {
25       cas(age, oldAge, newAge);
26      atomic{
27        if (oldAge == newAge)
28           return tsk;
29        else
30           emit(popBottom(empty));
31    }
32    age = newAge;
33    return NULL;
34 }
1 Promel model of Arora Algorithm Implementation

// Definitions -------------------------
#define stackSize 6
#define Undef StackSize + 1
#define pointer byte
#define NULL -6
#define red 1
#define blue 2
#define white 3
#define empty 0

// For loop -----------------------------
#define FOR(i,l,u) i = l; do ::i <= u ->
#define ROF(i,l,u) ; i++ ::i > u -> break od

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

// CAS --------------------------------
#define CAS(age, oldAge, newAge) atomic{
  if :: ((age.top == oldAge.top) && (age.tag == oldAge.tag)) -> 
    tmp.top = age.top; tmp.tag = age.tag; 
  newAge.top = tmp.top; newAge.tag = tmp.tag; 
  :: else -> 
    newAge.top = age.top; newAge.tag = age.tag; 
  fi; 
}

// START OF SHARED DATA -------------------------

typedef AgeBlock{
  pointer top;
  byte tag;
}

// model data structures --------------------------
pointer bot;
AgeBlock age;
byte deq[stackSize];

.getOrElse flags and counters to be used in assertions
/
bit pushRed;
bit pushBlue;
byte popRed;
byte popBlue;
bit isInit = false;
/
/channel for communication between processes
/
chan receive = [0] of {bool}
/

Initiate variables

inline initVariables(){
  bot = 0;
age.top = 0;
age.tag = 0;
pushRed = false;
pushBlue = false;
popRed= 0;
popBlue = 0;
init_deq();
    isInit = true;
}

Initiate deque

inline init_deq(){
byte i;
i = 0;
FOR(i,0,stackSize-1) 
  deq[i] = empty;
RDF(i,0,stackSize-1)
}

check LOSS property

inline checkLoss(){
  IF (pushRed) -> assert(popRed >= 1) FI;
  IF (pushBlue) -> assert(popBlue >= 1) FI;
}

check LIFO property

inline checkLifo(task){
  IF task == red -> IF pushBlue -> assert(popBlue >= 1) FI;
    FI;
}

check FIFO property

inline checkFifo(task){
  IF task == blue -> assert(popRed == 1);
    FI;
}

check CREATION property

inline checkCreation(task){
  if 
::task == red -> assert(pushRed);
::task == blue -> assert(pushBlue);
::else -> skip;
  fi
}
/----------------------check DUPLICATE property----------------------*/
inline checkDuplicate(task){
  if ::task == red -> assert(popRed <= 1);
  ::task == blue -> assert(popBlue <= 1);
  ::else -> skip;
  fi;
}

/*------------------------- pushBottom inline ---------------------*/
proctype pushBottom(byte color){
  pointer localBot;
  localBot = bot;
  deq[localBot] = color;
  localBot++;
  /* at linearization point of pushBottom, we update push flags. */
  atomic{
    bot = localBot;
    if ::color == red -> pushRed = true;
    ::color == blue -> pushBlue = true;
    ::else -> skip;
    fi;
  }
  receive ! 1;
}

/*------------------------- popBottom inline ----------------------*/
proctype popBottom(){
  pointer localBot;
  byte tsk;
  AgeBlock tmp;
  AgeBlock oldAge;
  AgeBlock newAge;
  localBot = bot;
  /* We check LOSS where popBottom returns NULL */
  atomic{
    IF (localBot == 0) ->
      checkLoss();
    goto end;
  }
  localBot--;
  /* First linearization point */
  atomic{
    bot=localBot;
    checkLifo(deq[localBot]);checkCreation(deq[localBot]);checkDuplicate(deq[localBot]);
  }
  tsk = deq[localBot];
  oldAge.top = age.top;
  oldAge.tag = age.tag;
  /* We update pop counters at the point operation returns a value */
  atomic{
    IF (localBot > oldAge.top) ->
      if ::tsk == red -> popRed++;
        ::tsk == blue -> popBlue++;
        ::else -> skip;
      fi;
  }
}
goto end;
::else -> skip;
fi;
}

/* Second linearization point. */
atomic{
bot = 0;
checkLifo(tsk);checkCreation(tsk);checkDuplicate(tsk);
}
newAge.top = 0;
newAge.tag = oldAge.tag + 1;
IF (localBot == oldAge.top) ->
CAS(age, oldAge, newAge)
/* We update pop counters at the point operation returns a value */
atomic{
if :: ((oldAge.top == newAge.top) && (oldAge.tag == newAge.tag)) ->
if :: tsk == red -> popRed++;
:: tsk == blue -> popBlue++;
::else -> skip;
fi;
goto end;
/* We check LOSS where popBottom returns NULL */
:: else -> checkLoss(); skip;
fi;
FI;
atomic{age.top = newAge.top; age.tag = newAge.tag;}
end:
receive ! 1;
skip;
}

/***************************** popTop ******************************/
proctype popTop(){
pointer localBot;
byte tsk;
AgeBlock tmp;
AgeBlock oldAge;
AgeBlock newAge;

atomic{
oldAge.top = age.top;
oldAge.tag = age.tag;
}
localBot = bot;
if ::(localBot <= oldAge.top) -> goto end;
::else -> skip;
fi;
tsk = deq[oldAge.top];
newAge.top = oldAge.top;
newAge.tag = oldAge.tag;
newAge.top++;
/* Linearization point and return point of popTop*/
We update pop flags and check properties at this point */
atomic{
CAS(age, oldAge, newAge);
if ((oldAge.top == newAge.top) && (oldAge.tag == newAge.tag))
    checkFifo(tsk); checkCreation(tsk); checkDuplicate(tsk);
if :: tsk == red -> popRed++;
:: tsk == blue -> popBlue++;
:: else -> skip;
fi;
:: else -> skip;
fi;
end;
/* Operation is aborted! */
skip;
}

/***************************** Victim process **********************/
proctype victim(){
/* Test Harness: */
begin:
if :: (bot < stackSize) ->
    if :: true -> run pushBottom(white); receive ? _; goto begin;
:: !pushBlue && !pushRed -> run pushBottom(red); receive ? _; goto begin;
:: !pushBlue && pushRed -> run pushBottom(blue); receive ? _; goto begin;
:: bot > 0 -> run popBottom(); receive ? _;
fi;
:: else -> skip;
fi;
}

/*************************** Initialization process *****************/
init{
atomic{initVariables();
if /* Test Harness: */ :: isInit ->
    run victim();
    run popTop();
    run popTop();
    run popTop();
:: else -> skip;
fi;
}