

**Modelling and Verifying
Non-blocking Algorithms that
Use Dynamically Allocated
Memory**

by

Simon Doherty

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington

2004

Abstract

This thesis presents techniques for the machine-assisted verification of an important class of concurrent algorithms, called *non-blocking* algorithms. The notion of *linearizability* is used as a correctness condition for concurrent implementations of sequential datatypes and the use of *forward simulation relations* as a proof method for showing linearizability is described. A detailed case study is presented: the attempted verification of a concurrent double-ended queue implementation, the *Snark* algorithm, using the theorem proving system *PVS*. This case study allows the exploration of the difficult problem of verifying an algorithm that uses low-level pointer operations over a dynamic data-structure in the presence of concurrent access by multiple processes.

During the verification attempt, a previously undetected bug was found in the *Snark* algorithm. Two possible corrections to this algorithm are presented and their merits discussed. The verification of one of these corrections would require the use of a *backward simulation relation*. The thesis concludes by describing the reason for this extension to the verification methodology and the use of a *hierarchical* proof structure to simplify verifications that require backward simulations.

Acknowledgements

Thanks to my supervisors Lindsay Groves and Ray Nickson, and to Mark Moir for their advice, encouragement and endless rereading. Thanks also to my parents and my partner Jill for their love and support. Finally, thanks to Sun Microsystems for its financial assistance and to Victoria University Wellington for providing the Postgraduate Research Scholarship.

Contents

1	Introduction	1
1.1	Non-blocking Algorithms and the Snark Algorithm	2
1.2	Formal Verification	2
1.2.1	Mechanical Theorem Provers	4
1.3	Verification of Non-blocking Algorithms	5
1.4	Finding and Fixing Bugs	6
1.5	Mathematical Notation	6
2	Non-Blocking Algorithms and Snark	9
2.1	Non-Blocking Data-structures	9
2.1.1	Non-blocking Progress Properties	10
2.1.2	Synchronisation Primitives	11
2.1.3	Consensus Theory and Universal Constructions	13
2.1.4	Lock-Free Algorithms	14
2.2	The Snark Algorithm	15
2.2.1	Dequeues	16
2.2.2	The Snark Data-structure	17
2.2.3	Push Operations	22
2.2.4	Pop Operations	24
2.2.5	Lock-freedom	25
2.3	System Requirements for Snark	26
2.3.1	Garbage Collection	26

2.3.2	Memory Consistency Models	27
3	I/O Automata, Specification and Simulation	31
3.1	I/O Automata	31
3.2	Describing I/O automata	33
3.3	Sequential Datatypes	36
3.4	Correctness and Atomic Automata	38
3.5	Canonical Automata	44
3.6	Properties of Canonical Automata	46
3.6.1	Atomicity of the Canonical Automaton	47
3.6.2	Completeness of the Canonical Automaton	48
3.7	Simulation Relations and Trace Inclusion	49
3.7.1	General Forward Simulation and Representation	49
3.7.2	Reachability and Forward Simulations	51
3.7.3	Simple Forward Simulation	52
4	Correctness and Representation	55
4.1	Correctness Requirements	55
4.1.1	The Deque Data-type	56
4.1.2	The Canonical Automaton for the Deque	57
4.1.3	A Symmetric Deque Automaton	58
4.2	The Snark Automaton	61
4.2.1	The Signature of the Snark Automaton	62
4.2.2	Modelling the Heap	64
4.2.3	Notation for the Heap Model	67
4.2.4	The States of the Snark Automaton	68
4.2.5	The Transition Relation of the Snark Automaton	70
5	The Attempted Verification	75
5.1	Step Correspondence	77
5.2	Overview of the Simulation Relation	79
5.3	The <i>correspondence_ok?</i> Predicate	81

5.4	The Representation Function and <i>obj_ok?</i>	85
5.5	The <i>rest_ok?</i> Predicate	88
5.5.1	The <i>dead_ok?</i> Predicate	89
5.5.2	The <i>conditions_ok?</i> Predicate	90
5.5.3	The <i>nds_ok?</i> Predicate	93
5.5.4	The <i>distinctness_ok?</i> Predicate	94
5.5.5	The <i>free_ok?</i> Predicate	97
5.6	Invariants of <i>SnarkAut</i>	98
5.7	Verifying the Simulation Relation	100
5.7.1	What Happens During Push Operations?	101
5.7.2	What Happens During Pop Operations?	107
5.8	Using PVS	113
5.8.1	Describing I/O Automata	113
5.8.2	Using the PVS Prover	114
6	The Bug in Snark	117
6.1	The Bug in the Snark Algorithm	117
6.2	Characterising the Bug	124
6.2.1	Approaches to Fixing the Bug	125
7	Corrections for Snark	127
7.1	Optimisations for the Corrections	128
7.2	Algorithm 1 - Version numbering	129
7.2.1	Limitations of Version Numbering	133
7.3	Algorithm 2 - Using CAS	134
7.4	Modelling and Verifying the Corrections	137
7.4.1	Verifying Algorithm 1	138
7.4.2	Verifying Algorithm 2	139
8	Conclusions	143
8.1	Evaluating the Verification Methodology	143
8.1.1	I/O Automata and Simulation Relations	144

8.1.2	The Representation Function	144
8.1.3	Mechanical Proof-checking	147
8.2	Complexity of Verification	147
8.3	The Snark Algorithm	148
8.4	Future Work	149

Chapter 1

Introduction

This thesis describes work on the formal verification of an important class of concurrent algorithms called *non-blocking* algorithms, which provide implementations of shared data-structures without using any form of mutual exclusion. A case study based on the attempted verification of a non-blocking algorithm called *Snark* is presented. The goal of this verification attempt was to define correctness conditions and arguments at a sufficient level of formality and rigour that the arguments could be submitted to a mechanical theorem prover for elaboration and validity checking.

Our effort to verify the Snark algorithm exposed a previously undetected bug. This bug had gone undetected despite the fact that the Snark algorithm had been published with a semi-formal proof of its correctness. This illustrates an important point about non-blocking algorithms: they tend to have very subtle behaviours and are difficult to validate by traditional means. Therefore, the application of formal methodologies to the verification of non-blocking algorithms can greatly increase our confidence in the correctness of these algorithms.

1.1 Non-blocking Algorithms and the Snark Algorithm

The Snark algorithm was originally developed at Sun Microsystems at Burlington, Massachusetts and was first presented in [8]. Snark was designed to implement a *double-ended queue* or *deque*. A deque is a datatype like a queue that supports pushes and pops (or enqueues and dequeues) at *both* ends. Snark exploits dynamic memory allocation and uses a linked structure to represent the deque.

Non-blocking algorithms typically make clever use of strong synchronisation primitives to guarantee that the system remains in a consistent state and so avoid the need for mutual exclusion; Snark follows this pattern. Because they do not rely on mutual-exclusion, non-blocking algorithms are able to guarantee one of several progress properties. These progress properties mean that certain problems associated with mutual exclusion do not occur: for example, all non-blocking algorithms are free from deadlock, most non-blocking progress conditions guarantee freedom from livelock. Chapter 2 provides a brief introduction to theoretical and practical issues relating to non-blocking algorithms and describes the Snark algorithm.

1.2 Formal Verification

A *formal verification* of an algorithm is an attempt to provide a very high level of assurance that the algorithm behaves correctly, using well understood mathematical models and arguments. Any formal verification methodology must provide answers to several questions:

- What are the correctness conditions for an implementation?
- How is the implementation to be modelled?
- What proof techniques are available to show that an implementation meets its specification?

This thesis provides answers to these questions, drawn from the work of several research programmes, particularly the work of Lynch and others at M.I.T [28, 29, 36]. Correctness is defined in terms of a certain mathematical notion of *sequential datatypes* and *linearizability*. The notion of linearizability, originally due to Herlihy and Wing [22], is a popular way of specifying correctness for concurrent systems: it defines a natural way of bridging the gap between concurrent systems and their sequential counterparts.

Labelled transition systems called *I/O automata* are used to model the Snark algorithm. Originally presented in [27], I/O automata make a distinction between transitions (modelling steps in a computation) that are internal and those that are externally visible. Specification of the desired behaviour of an automaton constrains the externally visible behaviour: this property allows us to use *abstract* automata as intermediate specifications between *concrete* automata (like the one that models the Snark algorithm) and their specifications. If the abstract automaton is known to meet its specification (put in terms of linearizability and a sequential datatype) *and* the abstract automaton has all the externally visible behaviour that the concrete automaton does, then the concrete automaton is correct.

In this thesis, we use a *simulation relation* in our attempt to show that the Snark algorithm meets its specification. Simulation relations are relations over the states of two automata: the existence of a simulation relation from one automaton to another guarantees that the externally visible behaviour of the first is shared by the second.

Simulation relations have a very useful property for showing that an implementation is correct: they reduce reasoning about all possible behaviours of the automaton to reasoning about the individual transitions. In this respect they are akin to proofs relying on invariants, which reduce reasoning about all possible states of an automaton to reasoning about transitions. This *locality* of proof obligations makes reasoning about a large set of possibilities tractable.

This thesis does *not* discuss the verification of progress properties. It is much easier to see that the Snark algorithm is non-blocking than to see that it is correct. This is true of many non-blocking algorithms: published algorithms are presented

with much more space devoted to correctness arguments than to progress arguments (see [8, 2, 31] for examples). Also, the techniques used in the verification of progress properties are very different to the techniques discussed here [35].

Chapters 3 and 4 describe the issues of specification, modelling and verification in more detail. Chapter 5 defines the simulation relation used in the attempted verification of the Snark algorithm.

1.2.1 Mechanical Theorem Provers

One of the advantages of using rigorous mathematical models and specifications is that proof obligations can be submitted to a mechanical theorem prover. A mechanical theorem prover is an application capable, at least, of checking proofs of theorems expressed in some kind of formal notation. Most provers have some ability to *construct* proofs, using heuristic-driven, automated proof search and decision procedures. The attempted verification of the Snark algorithm used the theorem prover PVS developed at SRI [7, 37].

The use of a mechanical theorem prover offers several advantages over the construction of proofs by hand. Automated proof search relieves the human of much of the responsibility for carrying out tedious, mechanical reasoning. The PVS system can carry out simple quantifier instantiation and propositional reasoning automatically, as well as applying lemmas based on well-designed heuristics. PVS also has sophisticated decision procedures for equational logic and pure boolean expressions. In combination, these features mean that a user of the PVS system can submit most simple proof goals to the PVS prover, with good reason to hope that they can be proved without any human intervention.

Mechanical theorem provers present the possibility of recording and re-using frequently applied modes of reasoning. They do this by providing a *strategy language*: PVS supports a LISP-like functional language for this purpose. Even when this language is used in a simple way, it greatly increases the level of automation available for the construction of proofs.

Proofs are checked with mechanical rigour. In the ideal case, steps in an argument are matched against the rules of the logic that the prover supports. However,

the use of decision procedures in a theorem proving system complicates this issue somewhat: the mechanically checked proof may rely on the correctness of decision procedures that do *not* explicitly represent applications of proof rules. Still, in the PVS system, these decision procedures are implementations of well-understood algorithms and may be assumed to be trustworthy.

1.3 Verification of Non-blocking Algorithms with Dynamic Memory

In addition to the general issues related to the description and verification of concurrent systems, there are important issues to be dealt with related to non-blocking algorithms that work with dynamically allocated memory. These algorithms typically use low-level operations for communicating with one another and operating on shared resources: reads and writes from and to shared memory. In particular, Snark uses a pointer-based linked structure and depends on an instruction that operates on addresses. This is unlike many concurrent algorithms, where processes have high-level operations for communicating with each other, and the low-level nature of these operations complicates proofs.

The low-level nature of the Snark algorithm, and the fact that it deals with operations on pointers in dynamic memory, raises several questions:

- How do we describe dynamic memory mathematically?
- How do we describe unbounded linked structures in dynamic memory and manage this information in a proof?
- How do we handle the fact that updates to shared memory by one process may affect the states and executions of other processes in unexpected ways?

Chapter 4 describes our answer to the first question; Chapter 5 answers the second and third. At the heart of Chapter 5 is the use a *representation function*. This representation function does not carry states of the concrete automaton to states of the abstract automaton, as in other verification strategies (for example, the standard

approach used in the Z community [39, Chapter 18]). Instead, it carries indexes of a sequence in the abstract state into pointers into the heap of the concrete state. This allows a concise treatment of the properties of the unbounded data-structure used to represent the deque in the concrete automaton. The use of this function elaborates an approach first applied in a verification appearing in [2]. We believe that the use of functions of this kind constitutes a generalisable approach to the verification of concurrent algorithms that use dynamic memory and low-level operations.

1.4 Finding and Fixing Bugs

As mentioned above, the attempted verification of the Snark algorithm led to the discovery of a bug in what was believed to be a correct non-blocking implementation of a deque. In lieu of a complete verification, this provides evidence that the proof approach described here is useful: an analysis of failed proof obligations lead to the construction of an incorrect execution of the Snark algorithm. Chapter 6 describes this process and the bug.

In response to the discovery that the Snark algorithm is incorrect, two possible fixes have been developed. These fixes allow the exploration of techniques used in the design of non-blocking algorithms. Moreover, one of these fixes has an unusual property: values can be popped from the data-structure representing the deque *before* it is determined which process will return the popped value. This creates the need for an extension to the methodology used in the attempted verification of the original Snark algorithm: we would need to use a *backward simulation*. The implications of this extension are discussed in Chapter 7.

1.5 Mathematical Notation

This section describes the mathematical notation used in this thesis. We use the standard logical connectives, listed here in order of increasing binding power: \forall for ‘for all’; \exists for ‘there exists’; \Rightarrow for implies; \vee for ‘or’; \wedge for ‘and’; \neg for ‘not’.

These binding conventions are consistent with those used in PVS [7]. The scope of bound variables extends to the end of the expression following the quantifier, and we use a dot notation to separate quantifier and predicate. Thus, in

$$\forall x \bullet P \Rightarrow Q(x)$$

x is bound in the predicate Q .

We use \mathbb{N} to denote the natural numbers, \mathbb{Z} to denote the integers and *bool* to denote the booleans $\{true, false\}$.

$S \times T$ is the Cartesian product of sets S and T . The projections π_1 and π_2 access the first and second members of these products, respectively. We also use $\prod_{s \in S} e$ where e is some set expression that may involve s , which yields the product of the sets e across the index set S . For products like this, we use the projections π_s for each $s \in S$.

For complex products, we often use mnemonic access names with a dot syntax. For example, for some tuple $t \in X \times (Y \times Z)$ we might stipulate that $t.x = \pi_1(t)$, $t.y = \pi_1(\pi_2(t))$ and $t.z = \pi_2(\pi_2(t))$.

Given a relation $R : S \times T$ and $s \in S$, $R[s]$ is the relational image of s onto T :

$$R[s] = \{t \in T \mid R(s, t)\}$$

We often need to modify the value of a function at a certain point: given a function $f : S \rightarrow T$, $s \in S$ and $t \in T$, $f[s := t]$ be the function exactly like f , but with $f(s) = t$, ie. for every $s' \in S$:

$$f[s := t](s') = \begin{cases} f(s') & \text{if } s' \neq s \\ t & \text{if } s' = s \end{cases}$$

Chapter 2

Non-Blocking Algorithms and Snark

This chapter discusses non-blocking algorithms in general and the Snark algorithm in particular. Section 2.1 presents brief discussions of certain topics relating to non-blocking algorithms: their advantages over traditional techniques for the implementation of shared data-structures; the operations on which they are based; an important theoretical problem called the *consensus problem*; and a brief survey of known non-blocking implementations. This provides background for Section 2.2, which describes the Snark algorithm in detail. Section 2.3 discusses the system requirements of the Snark algorithm and how these requirements relate to other non-blocking algorithms and to verification.

2.1 Shared Data-Structures and Non-Blocking Implementations

As discussed in the Chapter 1, non-blocking algorithms implement data-structures shared by several processes without using any form of mutual exclusion. Mutual exclusion creates several well-known problems: Greenwald's thesis [15] provides a good introduction to some of these problems and the advantages which non-blocking algorithms provide.

Perhaps most notorious of the problems with mutual exclusion is the issue of

deadlock. Deadlock occurs when two or more processes compete to obtain locks on the same set of resources. If each process holds a lock on one resource from the set (of course, these will be distinct) and does not release it, then no process can obtain the locks it needs to continue computation. Techniques exist to solve this problem, but it continues to be an issue in the development of multi-process systems. Non-blocking implementations do not use locks, and so do not contribute to deadlock.

Another advantage of non-blocking algorithms is that they often scale well to systems where dozens or hundreds of processors can access shared memory simultaneously. In cases like these, an algorithm that uses locks can perform very poorly: if a process holding a lock on a shared data-structure is pre-empted by some other process, then all other processes on all other processors must wait for the pre-empted process to run again and complete its operation. If a data-structure is non-blocking, other processes are free to execute operations without having to wait for stalled processes.

2.1.1 Non-blocking Progress Properties

There are several non-blocking progress properties which have been the subject of research. Currently, the most important are *wait-freedom*¹ and *lock-freedom*. These two properties form a hierarchy. That is, every lock-free algorithm is wait-free. Their definitions, taken from [21], follow. Underlying these definitions are notions of executing high-level *operations* and low-level *steps*. These correspond to executing procedures in the interface to some datatype, and executing individual statements or instructions within those procedures, respectively. These definitions also use a notion of *execution* which is the sequence of steps which processes take during all the operations executed on a data-structure during its lifetime.

Definition 2.1 (Wait-freedom) *An implementation of an operation is wait-free*

¹There is ambiguity in the literature between the terms *lock-free* and *non-blocking*. Until recently, they have been used synonymously. However, a convention is developing whereby *non-blocking* describes the whole family of algorithms which do not rely on mutual exclusion, and *lock-free* describes a class within that family. This is the convention used here.

if, after a finite number of steps of any execution of that operation, that operation completes.

Definition 2.2 (Lock-freedom) *An implementation of an operation is lock-free if, after a finite number of steps of any execution, some operation has completed.*

Lock-freedom is the weaker condition: lock-freedom allows the possibility that some processes *never* complete their operations. So long as some processes are completing, the others may be prevented from making progress. Wait-freedom precludes this property: every process is guaranteed to complete. The Snark algorithm is lock-free but not wait-free (see Section 2.2).

Lock-freedom precludes the use of locks for the following reason. A process which has a lock on a given resource stops any other operations on that resource from completing. If this process does not complete any steps in its execution (if the process fails), then there will be no finite number of steps which any other process can take to complete their operations.

The advantage that lock-freedom does have is that implementations of lock-free algorithms tend to be much simpler and more efficient than wait-free implementations [15].

2.1.2 Synchronisation Primitives

The construction of non-blocking algorithms and analysis of their properties constitutes a large research area. These algorithms normally make substantial use of powerful synchronisation primitives. The *compare-and-swap* (CAS) instruction is a very popular synchronisation operation. The CAS operation takes three arguments, an address `addr`, and two value arguments, `old` and `new`, and returns a boolean value. The value currently at `addr` is tested against `old`. If they are equal, then the value at `addr` is updated to `new` and the CAS returns `true`; otherwise, no update is made and the CAS returns `false`. These comparisons and updates happen atomically. That is, no other operation on memory appears to occur between the invocation and response of the CAS operation. Figure 2.1 contains C-style pseudo-code representing the semantics of CAS.

```

boolean CAS(val *addr,
            val old,
            val new) {
    atomically {
        if (*addr == old){
            *addr = new;
            return true;
        } else return false;
    }
}

```

Figure 2.1: Semantics of the CAS instruction

```

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

```

Figure 2.2: Semantics of the DCAS instruction

The *double-compare-and-swap* (DCAS) instruction is a generalisation of CAS. It operates like CAS, but compares and updates two addresses instead of one. Figure 2.2 describes its semantics. Note that an update occurs only if *both* addresses contain the old values. CAS is available on many multiprocessor systems [15], whereas almost no systems offer DCAS at the hardware level.² For this reason, the algorithms which use DCAS currently have a very limited range of uses and their development serves mainly to assess the utility of providing DCAS as a primitive instruction [15, 14, 8, 2, 30].

²To my knowledge, the only system offering such a DCAS is the Motorola 68040 [2].

There are several other strong synchronisation primitives which have been implemented on many machines, among them *test-and-set* and *fetch-and-add*. Semantics for these instructions are given in Figures 2.3 and 2.4. Both these operations atomically read a value at some address, and modify that address, returning the value originally read. For this reason they are sometimes called *read-modify-write* (RMW) instructions.

2.1.3 Consensus Theory and Universal Constructions

RMW instructions are less powerful than CAS and DCAS instructions because they unconditionally update memory, and the updated value is constrained to be a (sometimes constant) function of the old value. This difference in power can be formalised in terms of *consensus theory*. The consensus problem is a very important one in distributed-systems theory. Briefly, the consensus problem is the problem of making some set of processes agree on an output value, taken from a set of input values, each input value being assigned to one process.

In “Wait-free Synchronisation” [17], Herlihy showed that CAS can be used to construct a wait-free solution to the consensus problem for any number of processes; RMW instructions are only able to solve the consensus problem for sets of two processes. Herlihy presents a series of results of this form for several different synchronisation primitives and shows that for any natural number n , there is a synchronisation primitive which can solve the consensus problem for up to, but no more than n processes.

```
boolean test_and_set(boolean *addr)
    atomically {
        boolean b = *addr;
        *addr = true;
        return b;
    }
}
```

Figure 2.3: Semantics for test-and-set.

```
int fetch_and_add(int *addr, int k)
  atomically {
    int i = *addr;
    *addr = i+k;
    return i;
  }
```

Figure 2.4: Semantics for fetch-and-add.

Herlihy also showed that any operation which can solve the consensus problem for any number of processes can be used in a *universal construction*. A universal construction is a mechanical method of transforming sequential code for some data-type into a (in this case, wait-free) concurrent implementation. Because they are mechanical, universal constructions offer the possibility of developing wait-free algorithms from their much simpler, sequential counterparts. Unfortunately, the implementations so constructed tend to be very inefficient in most circumstances and are very rarely used in practical applications[15].

2.1.4 Lock-Free Algorithms

Numerous published lock-free algorithms exist; there are several lock-free implementations of deques alone. The deque implementations given in [2] and [30] were developed, like Snark, at Sun Microsystems Laboratories, Burlington, Massachusetts and all rely on the DCAS operation. Deques have received substantial attention for several reasons [33]. They generalise the two most commonly used concurrent data-structures, stacks and queues. Also, they offer developers an opportunity to explore issues in the implementation of concurrent data-structures: implementations must detect and handle complex boundary cases properly; and they benefit greatly from making use of parallelism in access to opposite ends of a non-empty deque.

Arora, Blumfoe and Plaxton [4] present a practical implementation of a non-blocking deque, designed for use in a *work-stealing* algorithm. Work-stealing is a technique used in thread scheduling for multiprocessor systems. The work-

stealing deque uses the CAS instruction rather than DCAS but does not have all the functionality of the general deques in [2, 30] and Snark. Only one end supports both pushes and pops, the other end offering only pops. Also, the end with both operations can only be accessed by one designated process; and the pop operation on the opposite end is allowed to *abort* if another pop operation completes successfully during its execution. These restrictions allow the work-stealing deque algorithm to be simple, efficient, and correct for its specialised purpose all without relying on the DCAS operation.

Other common data-structures have been successfully implemented and used in practical applications. Michael and Scott [31] present a practical implementation of a lock-free queue object which uses the CAS operation. Treiber [38] presents a simple and efficient lock-free stack implementation, based on a linked list and using the CAS operation.

2.2 The Snark Algorithm

Snark [8] was designed to be a lock-free implementation of a deque, using DCAS to provide lock-freedom. It was believed to be an advance on other DCAS based deque algorithms for several reasons:

- It exploits the natural parallelism of deques. Processes popping from different ends when there is more than one element in the deque are able to complete their operations without interfering with each other.
- No ‘spare bits’ are needed in the addresses containing pointers to deque nodes. Non-blocking algorithms often allow processes to signal various conditions to other processes by setting a bit (or several) in an operand of a CAS or DCAS which simultaneously contains a pointer value. This is possible because in many multiprocessors, CAS (and hypothetically, DCAS) operates on a word wider than what is needed to represent a pointer. [16, 32] provide examples; [2] is a deque algorithm predating Snark that uses this kind of technique. However, especially in cases where several spare bits are

needed, storing extra information in pointers reduces the applicability of the algorithm.

- The Snark algorithm uses only one DCAS per operation. This is beneficial because hardware implementations of DCAS are likely to be very much slower than normal reads and writes.
- The algorithm uses a dynamically allocated data-structure, rather than a statically allocated array, and so can adapt to fluctuations in the number of items currently stored.

The remainder of this section describes the Snark algorithm.

2.2.1 Deques

As mentioned in Section 1.1, a deque is a datatype like a queue but which supports operations to add and remove elements at both ends. In more detail³, a deque is a container type which offers the interface described in Figure 2.5. A deque maintains a sequence of elements storing the values which have been put into the deque, but not yet popped: invocations of the form `pushLeft(v)` place `v` onto the left end of this sequence; invocations of the form `pushRight(v)` place `v` onto the right; `popLeft` removes an element from the left end of the sequence and returns it; `popRight` does this on the right end of the sequence. `popRight` and `popLeft` return a special value "empty" if they find the deque empty: "empty" should not be pushed onto the deque.

In the Snark algorithm, `pushLeft(v)` and `pushRight(v)` are allowed to not add `v` to the sequence, when there is no room in dynamic memory. The type `rtype` is used to indicate when this has happened: "ok" is returned when the value has been successfully pushed; "full" is returned if allocation failed.

³Section 4.1 describes the deque datatype in formal terms, based on material presented in Chapter 3.

```

type rtype = {"ok", "full"}

rtype pushLeft(val v);
rtype pushRight(val v);
val popLeft();
val popRight();

```

Figure 2.5: The deque interface.

2.2.2 The Snark Data-structure

Snark uses a doubly-linked-list data structure to represent the deque, which is accessed through the `LeftHat` and `RightHat` pointers. Figure 2.6 specifies the node structure which makes up the list, and the initialisation code for the deque. Nodes in the list are never explicitly deallocated: Snark assumes the availability of a garbage collector, which allows memory to be recycled.

Figures 2.7 and 2.8 contain C-style pseudo-code for the right-hand side operations, `pushRight` and `popRight`. The left-hand side operations are entirely symmetric but have been included in Figures 2.9 and 2.10 for completeness. Flow-charts representing the right-side operations appear at the end of the chapter in Figures 2.14 and 2.15.

In the following discussion, it is convenient to call the node pointed to by the pointer variable `LeftHat` simply the `LeftHat`. Likewise for the `RightHat`.

The use of clever sentinel nodes marking each end of the deque is critical to the way Snark handles pops and detection of an empty deque. A node is *left-dead* if its `L` field contains a self-pointer; likewise, it is *right-dead* if its `R` field contains a self-pointer. The Snark deque maintains two important properties:

1. structure Node {	1. Node Dummy = new Node();
2. Node *L;	2. Dummy->L = Dummy;
3. Node *R;	3. Dummy->R = Dummy;
4. val V;	4. Node *LeftHat = Dummy;
5. }	5. Node *RightHat = Dummy;

Figure 2.6: Node structure and deque initialisation.

```
1. rtype pushRight(val v) {
2.     nd = new Node();
3.     if (nd == null) return "full";
4.     nd->R = Dummy;
5.     nd->V = v;
6.     while (true) {
7.         rh = RightHat;
8.         rhR = rh->R;
9.         if (rhR == rh) {
10.            nd->L = Dummy;
11.            lh = LeftHat;
12.            if (DCAS(&RightHat, &LeftHat,
13.                    rh, lh, nd, nd))
14.                return "ok";
15.        } else {
16.            nd->L = rh;
17.            if (DCAS(&RightHat, &rh->R,
18.                    rh, rhR, nd, nd))
19.                return "ok";
20.        }
21.    }
```

Figure 2.7: Snark deque - right push.

```

1.  val popRight() {
2.      while (true) {
3.          rh = RightHat;
4.          lh = LeftHat;
5.          if (rh->R == rh) {
6.              if (DCAS(&RightHat, &rh->R,
7.                      rh, rh, rh, rh))
8.                  return "empty";
9.          } else if (rh == lh) {
10.             if (DCAS(&RightHat, &LeftHat,
11.                    rh, lh, Dummy, Dummy))
12.                 return rh->V;
13.          } else {
14.              rhL = rh->L;
15.              if (DCAS(&RightHat, &rh->L,
16.                      rh, rhL, rhL, rh)) {
17.                  result = rh->V;
18.                  rh->R = Dummy;
19.                  rh->V = null; /* Optional */
20.                  return result;
21.              }
22.          }
23.      }
24.  }

```

Figure 2.8: Snark deque - right pop.

```
1. rtype pushLeft(val v) {
2.     nd = new Node();
3.     if (nd == null) return "full";
4.     nd->L = Dummy;
5.     nd->V = v;
6.     while (true) {
7.         lh = LeftHat;
8.         lhL = lh->L;
9.         if (lhL == lh) {
10.            nd->R = Dummy;
11.            rh = RightHat;
12.            if (DCAS(&LeftHat, &RightHat,
13.                    lh, rh, nd, nd))
14.                return "ok";
15.        } else {
16.            nd->R = lh;
17.            if (DCAS(&LeftHat, &lh->L,
18.                    lh, lhL, nd, nd))
19.                return "ok";
20.        }
21.    }
```

Figure 2.9: Snark deque - left push.

```

1.  val popLeft() {
2.      while (true) {
3.          lh = LeftHat;
4.          rh = RightHat;
5.          if (lh->L == lh) {
6.              if (DCAS(&LeftHat, &lh->L,
7.                      lh, lh, lh, lh))
8.                  return "empty";
9.          } else if (lh == rh) {
10.             if (DCAS(&LeftHat, &RightHat,
11.                    lh, rh, Dummy, Dummy))
12.                 return lh->V;
13.          } else {
14.             lhR = lh->R;
15.             if (DCAS(&LeftHat, &lh->R,
16.                    lh, lhR, lhR, lh)) {
17.                 result = lh->V;
18.                 lh->L = Dummy;
19.                 lh->V = null; /* Optional */
20.                 return result;
21.             }
22.         }
23.     }
24. }

```

Figure 2.10: Snark deque - left pop.

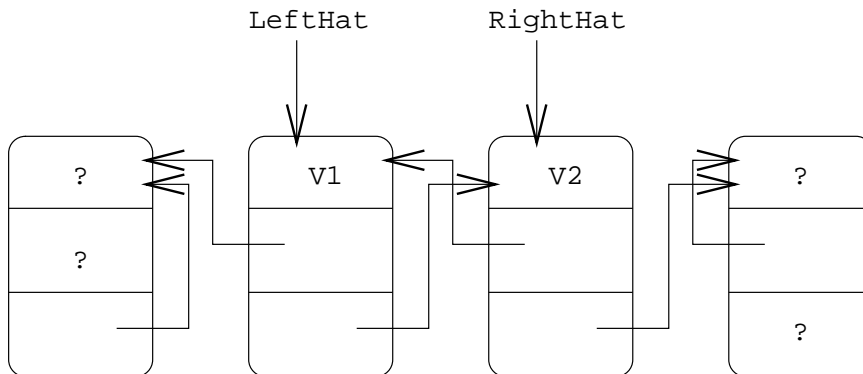


Figure 2.11: Typical non-empty deque.

1. When the deque is not empty, the node immediately to the left of the `LeftHat` is right-dead and the node to the right of the `RightHat` is left-dead. See Figure 2.11.
2. When the deque is empty, the `LeftHat` is left-dead and the `RightHat` is right-dead. See Figure 2.12.

The sentinel nodes are not necessarily distinct. In fact, Snark uses a global constant `Dummy`, which points to a node which is both left-dead and right-dead, as the initial value of both `LeftHat` and `RightHat`: this initial state represents an empty deque (see Figure 2.13). No node within the deque (ie., that holds a value that has been pushed onto the deque but not popped) is ever left- or right-dead.

The sentinel nodes are not treated as containing actual values: the fields containing question marks in figures 2.11 and 2.12 are irrelevant to the representation of the deque and can contain any value.

2.2.3 Push Operations

The `pushRight` routine functions as follows. Line numbers mentioned in this discussion refer to the line numbers of Figure 2.7. First, a process executing `pushRight` attempts to allocate a new node called `nd`: if this allocation fails, the

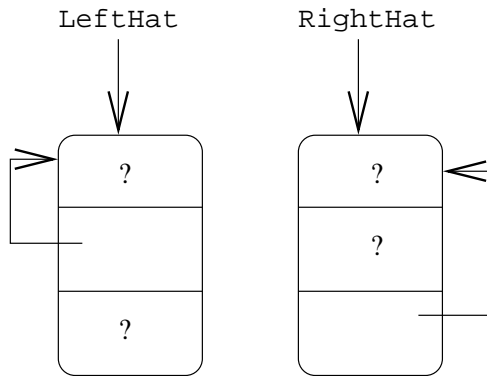


Figure 2.12: The form of an empty deque.

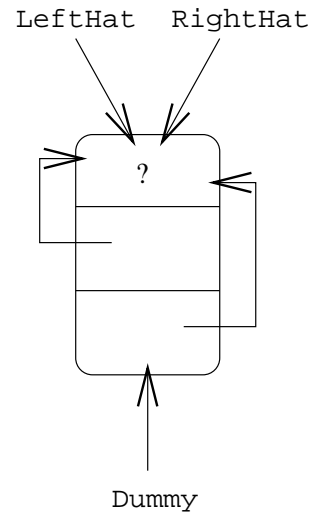


Figure 2.13: Initial empty deque.

push returns indicating that the heap was full; otherwise $nd \rightarrow V$ is set to the value that is being pushed; and $nd \rightarrow R$ is set to point to Dummy. Recall that Dummy is always both left- and right-dead: since nd will become the rightmost node in the deque, we need its R field to point to a left-dead node. Now, the current value of RightHat is loaded into the local variable rh . If the deque is empty, as ascertained by the test at line 9, then `pushRight` sets $nd \rightarrow L$ to Dummy. This is because $nd \rightarrow L$ must point to a right-dead node if nd is successfully pushed onto the deque while it is empty. After loading the current LeftHat and using a DCAS, `pushRight` attempts to set both the LeftHat and RightHat to the new node. If this succeeds, because $nd \rightarrow L == nd \rightarrow R == \text{Dummy}$ and Dummy is both left- and right-dead, property (2) above will be fulfilled. If the DCAS fails, it must be that some other process has updated the deque since the current process loaded its hats. In this case, `pushRight` will re-load the RightHat and attempt the push operation again. This loop (like the similar loops in the pop routines) may fail to terminate and it is this property which means Snark is not wait-free.

If the test at line 9 determines that the deque is not empty, `pushRight` attempts to splice the node onto the right end of the deque. $nd \rightarrow L$ is set to the value

that the pushing process saw in `RightHat` at line 7, and the DCAS at line 16 is attempted. If it succeeds, this DCAS swings the `RightHat` variable to point to `nd` and sets `rh->R` (the rightwards field of the old `RightHat`) to `nd`. Because `nd->R == Dummy`, property (1) above is preserved and because the DCAS succeeded we know that `nd->L` points to the old value of `RightHat`. If the DCAS fails, the loop is retried.

2.2.4 Pop Operations

This section describes the operation of the `popRight` routine. Line numbers here refer to the line numbers of Figure 2.8. The `popRight` routine begins by loading both hats into local variables. Then, it tries to determine if the deque is currently empty. First it tests whether `rh->R` is right-dead (at line 5). Then it uses a DCAS to check that `rh == RightHat` and that `rh` is still right-dead. If this DCAS succeeds `RightHat` is right-dead and, by property (2), the deque is empty. If the DCAS fails, the process retries. As with the `pushRight` routine, this fail and retry pattern can continue indefinitely.

If the test at line 5 fails, `popRight` attempts to tell if the deque has exactly one element. If so, it uses a DCAS to try to set both `RightHat` and `LeftHat` to `Dummy`, thus creating an empty deque. If this DCAS succeeds, the popping process is free to return the value contained in the node `rh`. If the DCAS fails, the loop is retried.

If the test at line 8 fails, then `popRight` attempts to pop a node from the right side of the deque. Using a DCAS it attempts to swing `RightHat` inwards to point to `RightHat->L` and make `rh` (the old `RightHat`) left-dead (preserving property (2)). The conditions on a successful DCAS ensure that if this operation succeeds then `rh == RightHat` and `rhL == RightHat->L`. After a successful DCAS, the node which was to be popped now contains a self-pointer and has become the new sentinel. The value to be returned is then loaded into `result`, and the field pointing *outwards* from the deque is set to `Dummy`. This allows the old sentinel (if it is not `Dummy`) to be reclaimed by the garbage collector. The line marked `/* Optional */`, if applied, would improve the interaction

of Snark with the garbage collector: if it is not applied and the deque is being used to contain pointer values then the value in the popped node cannot be collected until after the next push or pop. This is because there is a path from either hat, through the deque to the sentinel node which contains the value. Setting this value field to `null` solves this problem.⁴

While the DCAS at line 13 is meant to be used when the deque contains more than one element, it is possible for it to be executed successfully when the deque contains exactly one element. [8] presents a scenario, dubbed ‘hats crossing’ where two processes popping from opposite ends of the deque begin pop operations on a deque with two elements, and then both successfully execute the DCAS at line 13 one after the other. The second process to execute its DCAS will be removing the last node from the deque. This scenario does *not* result in incorrect behaviour. However, it turns out that it is possible for the DCAS at line 13 to be attempted and succeed, even when the deque is actually empty: this is the source of the bug in Snark, which is described fully in Chapter 6.

The Snark algorithm as described here differs in an important respect from that presented in [8]. In that presentation, line 6 of the pop routines did not appear: it was believed that the test at line 5 (`rh->R == rh` in the `popRight` routine) was enough for a process to tell that the deque was empty. However, it is possible for the test `rh->R == rh` to succeed, but for the deque not to be empty: this can happen when `rh` is no longer the `RightHat` and has been popped from the deque *from the left*. Applying the DCAS at line 6 allows the process to test that both `rh->R == rh` and `rh == RightHat`, which is enough to guarantee that `RightHat` is right-dead. The discovery of this bug and the fix presented here are due to Moir [33].

2.2.5 Lock-freedom

It is fairly easy to see why the Snark algorithm is lock-free. The DCASs which update the deque in the push and pop routines will fail only if some other process

⁴The version of Snark which we attempted to verify does not model this step.

successfully executes a DCAS on the deque. Once a DCAS is successfully executed by a process, there are only a finite number of steps which that process must take before completing its operation all of which take place independently of any other process.

2.3 System Requirements for Snark

The Snark algorithm requires that any system on which it runs have certain properties: as mentioned above, there must be a garbage collector if memory is to be recycled; also, as the algorithm is presented here, the memory system must support a strong consistency model [8]. Subsections 2.3.1 and 2.3.2 explain these requirements and briefly describe work in these areas as related to non-blocking algorithms and program verification.

2.3.1 Garbage Collection

Snark does not use any explicit deallocation of memory and so needs a garbage collector to allow memory to be recycled. Although a process which has completed `popRight` makes the old sentinel node redundant to the representation of the deque, there is no way to determine if some other process still has a pointer to it. It may be that another process executing `popRight` loaded a pointer to one of these nodes into `rh` and was then interrupted, during which time several nodes were popped from the right.

The inability to reliably use explicit deallocation is a very common problem in non-blocking algorithm design [24] and several approaches have been used. One is to simply assume the presence of a garbage collector, as Snark does. This has the drawback that the non-blocking properties of the algorithm will not be preserved if a non-blocking garbage collector is not used and the construction of a non-blocking garbage collector is a very difficult problem. Part of this difficulty comes from the fact that a non-blocking garbage collector cannot rely on the presence of a garbage collector to help implement the data-structures it uses.

Detlefs *et al.* [9] present a way to transform a lock-free algorithm which depends on a garbage collector into one which uses DCAS and reference counting to enable explicit deallocation. Herlihy, Luchangco and Moir [24] introduce the *repeat offender problem* and provide a lock-free solution: solutions to the repeat offender problem can be used to help design dynamic sized non-blocking data-structures which do not require a garbage collector. Herlihy *et al.* [18] give an example of how this technique can be used to transform a program that cannot safely free memory into one that can.

2.3.2 Memory Consistency Models

The Snark algorithm has been designed to work with memories that are *linearizable* (see [8, Section 2]). Roughly speaking, a memory is linearizable if it appears to each process that each operation on the memory (in this case, a read, write or DCAS) begins and ends without any other operation beginning or ending.⁵ Many multiprocessor systems do not have this property. These non-atomic memories support much weaker guarantees about the order in which operations occur and when changes to memory become visible to other processes. The guarantees which a multiprocessor does provide are called its *memory consistency model*. Adve and Garachorloo [1] present 11 consistency models, all non-atomic, which have been used in commercial multi-processors. Typical relaxations include one or more of the following: a processor may write a value to a location and read that value back before any other processor is able to read the new value; a processor can read the value written by *another* processor before that write becomes visible to the remaining processors; in some models it is even possible for a processor to write a value and *then* read back the value which was overwritten. This list is not exhaustive.

Assuming linearizability of memory during design is one way of tackling the fact that there are many different consistency models which an algorithm may have to deal with. Most multiprocessor systems offer instructions which alleviate

⁵Linearizability will be described fully in Chapter 3.

the weakness in the consistency model [1]. As an example, some systems with the CAS instruction have the following property: when a process executes CAS, all the writes to shared memory that process has executed are guaranteed to be visible to every other process once the CAS is complete. Some systems also offer specialised instructions which have similar effects: the STBAR instruction available on the SPARC V8 system is an example. Once an algorithm has been developed under the assumption of linearizable memory, that algorithm can be augmented with instructions of this kind to achieve an implementation for a particular machine.

The attempted verification of the Snark algorithm assumes a very strong memory consistency model that guarantees linearizability: all writes become visible to all processes instantly. There are two very good reasons for this: most obviously, Snark has been designed under the assumption of linearizable memory; secondly, modelling and reasoning about non-atomic memories is a very difficult problem which has attracted a significant amount of research.

One approach is to reduce a model of a program running on a non-atomic memory to one which is atomic, but more complex. Choy and Singh [6] present a strategy for achieving this by introducing processor-local auxiliary variables to the program and stipulating that updates to shared memory in the non-atomic model become updates to these auxiliary variables. Then the method by which the values of these variables are translated to globally accessible updates is specified according to the consistency model being represented.

Anderson and Gouda [3] present a similar transformation-based approach for a more abstract kind of non-atomic memory. They make the interesting observation that the invariants which are of use in verifying a program under the *assumption* of atomicity can be modified to help verify the program for non-atomic memory. Lamport [25] describes a proof system which allows a program to be verified in a way which exposes the ordering relations which must exist between non-atomic memory operations. Once these orderings have been obtained, they can be used as the basis of a modification of the program to work with a given memory consistency model.

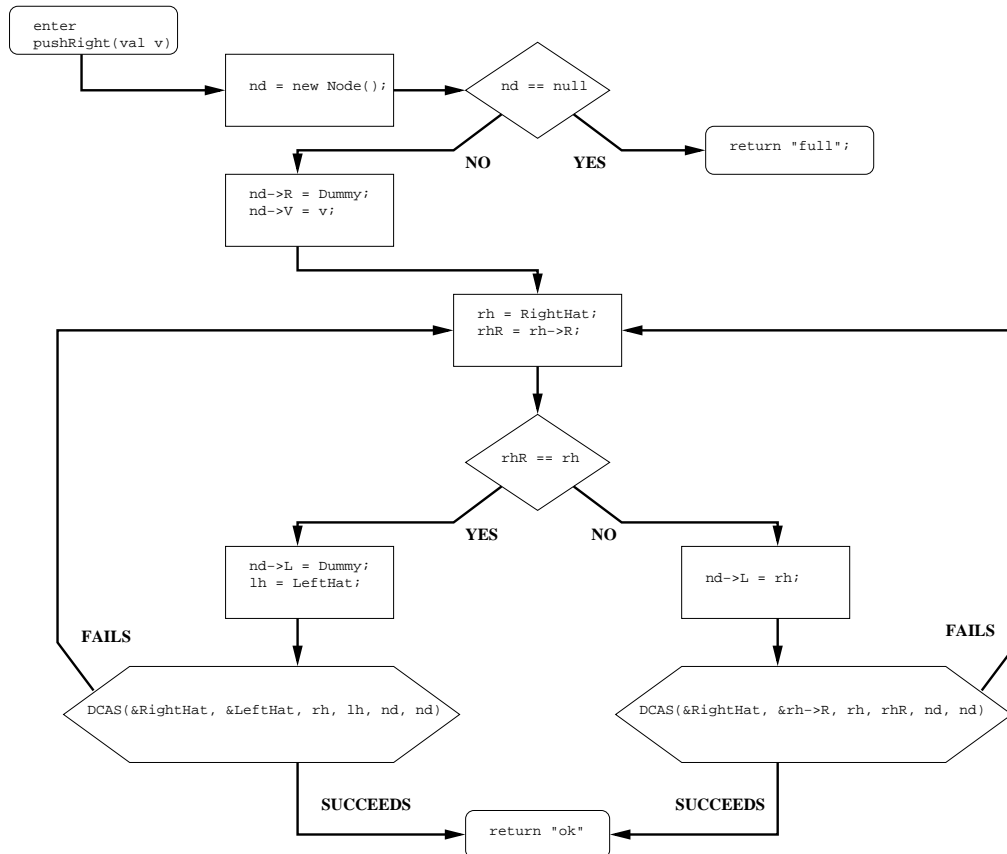


Figure 2.14: Flow chart - pushRight

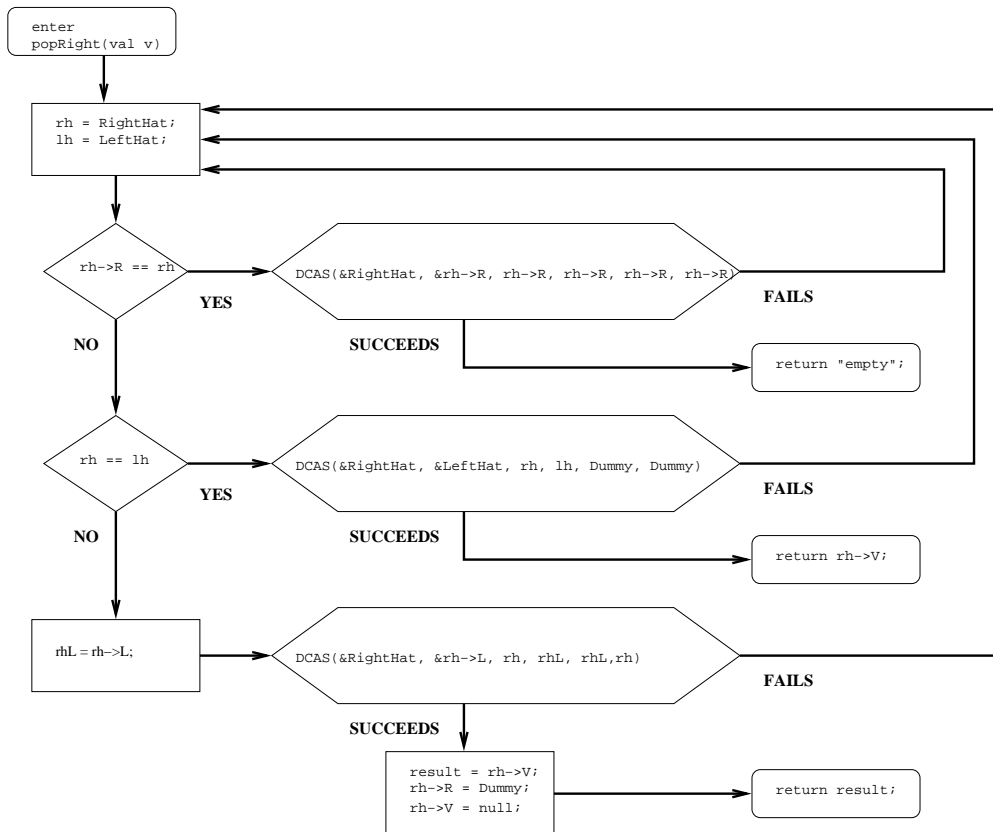


Figure 2.15: Flow chart - popRight

Chapter 3

I/O Automata, Specification and Simulation

This chapter answers three questions which are critical to the formal verification of concurrent algorithms: how are the correctness requirements of a concurrent algorithm to be described mathematically; how are the algorithms themselves to be modelled mathematically; and how should a proof that an algorithm meets its requirements be constructed? Correctness is defined in terms of a certain mathematical notion of *datatype*; the algorithms themselves are modelled using *I/O automata*; and the proofs are based on *simulation relations*.

3.1 I/O Automata

I/O automata originated in the work of Nancy Lynch and Mark Tuttle at M.I.T [27] and are essentially labelled-transition systems in which transitions are classified according to whether they are ‘visible’ to the external world. Externally visible transitions are further classified according to whether they model invocation (input) or response (output) events. I/O automata have become very popular for modelling concurrent systems [27, 17, 12, 36].

What follows is adapted from material in [27] and [29]. The I/O automata used here differ from those described in [27] and [29]: in both those treatments,

I/O automata must be *input-enabled*; here we do not require that automata be input-enabled. Input-enabledness is discussed in Section 3.5.

Definition 3.1 (LTS, signature, I/O automaton, external action) A labelled transition system (LTS) is a quadruple $(states, start, acts, trans)$, where $states$ is a set of states (over some set or type); $start \subseteq states$ is a nonempty set of start states; $acts$ is the label alphabet (called actions); and $trans \subseteq states \times acts \times states$ is the transition relation. Given an LTS $T = (states, start, acts, trans)$, a signature of T is a triple of subsets of $acts$ called *Input*, *Output* and *Internal*, where each member of $acts$ is contained in exactly one component of S . An I/O automaton is a pair (T, S) , where T is an LTS, and S is a signature of T . For any automaton $A = (T, S)$, $external(A)$, is the union of the sets *Input* and *Output* defined by the signature S .

We need some notation to access the components of an I/O automaton. Given an I/O automaton $A = (T, S)$ where $T = (states, start, acts, trans)$ and $S = (Input, Output, Internal)$, let $states(A) = states$, $start(A) = start$, $acts(A) = acts$ and $trans(A) = trans$; also, let $Input(A) = Input$, $Internal(A) = Internal$ and $Output(A) = Output$. When $(s, a, s') \in trans(A)$ we write $s \xrightarrow{a}_A s'$, or $s \xrightarrow{a} s'$ when no confusion is possible. If $s \xrightarrow{a}_A s'$ we may refer to s as the *pre-state* of the transition, and s' as the *post-state*.

Accessing components of I/O automata in this way is slightly at odds with the dot notation described in Section 1.5 and is used here for consistency with notation used in the literature (for example, [28, 29]).

The external actions are meant to be actions which are visible to the outside world. The sequences of these actions that an automaton can generate represent its externally visible behaviour.

Definition 3.2 (Execution fragment, execution, trace, traces) For any I/O automaton A :

- An execution fragment from s_m to s_n is a (possibly empty) sequence $\alpha = \langle s_m, a_m, s_{m+1}, \dots, a_{n-1}, s_n \rangle$ made up of alternating states and actions of A , where

$s_i \xrightarrow{a_i} s_{i+1}$ for each $m \leq i < n$. If such an execution fragment exists, we write $s_m \xrightarrow{\alpha}_A s_n$, or $s_m \xrightarrow{\alpha} s_n$ when no confusion is possible.¹

- An execution of A is a finite or infinite sequence $\langle s_0, a_0, s_1, a_1, \dots \rangle$ of alternating states and actions of A , where $s_i \xrightarrow{a_i} s_{i+1}$ for each i and $s_0 \in \text{start}(A)$.
- For an execution $\alpha = \langle s_0, a_0, s_1, a_1, \dots \rangle$ of A , $\text{trace}(\alpha)$ is the sequence α restricted to actions of the automaton which are in $\text{external}(A)$.
- The set $\text{traces}(A)$ is the set of traces of executions of A .

Definition 3.3 (Trace inclusion) The trace inclusion pre-order \leq_T , is defined thus: for any I/O automata A and B , $A \leq_T B$ iff $\text{traces}(A) \subseteq \text{traces}(B)$.

Note that trace inclusion is a *pre-order*: that is, it is reflexive and transitive. Other trace pre-orders can be defined, relating to strictly finite or infinite traces [29].

The set of traces of an automaton defines its externally observable behaviour. If, for any automata A and B , $A \leq_T B$, then any behaviour exhibited by A could also be exhibited by B . Trace inclusion can be used to allow one I/O automaton to specify the desired behaviour of another automaton. Sections 3.4, 3.5 and 3.7 describe a verification technique based on trace inclusion between a specification automaton and an implementation automaton.

3.2 Describing I/O automata

It will be useful to have some notation to describe the states and transition relations of I/O automata. The notation we describe here is modelled closely on the IOA language which has been used for describing I/O automata [13, 12]. This section presents a simple I/O automaton that is used to illustrate this notation and provides an example of the modelling style used in this thesis.

¹Note that this notation differs from that used in [29]. There, the arrow notation is used to represent a *move*: the notion of *move* is not used here.

Our example is an automaton that models a very simple mutual-exclusion protocol for any number of processes. Given a set of processes $PROC$ and a set $COUNTER = \{idle, has_lock, acquiring, releasing\}$, define the automaton $Mutex$ as follows:

$$\begin{aligned} states(Mutex) &= bool \times \prod_{p \in PROC} COUNTER \\ Input(Mutex) &= \{acquire_inv_p, release_inv_p \mid p \in PROC\} \\ Internal(Mutex) &= \{\} \\ Output(Mutex) &= \{acquire_resp_p, release_resp \mid p \in PROC\} \end{aligned}$$

The interpretation is that $Mutex$ models $|PROC|$ processes running in parallel: an invocation of $acquire$ by p is modelled by a transition labelled by the p -indexed action $acquire_inv_p$; a $release$ invocation is modelled by a $release_inv_p$. Transitions labelled by the response (output) actions $acquire_resp_p$ and $release_resp_p$ model the process p returning from $acquire$ or $release$ operations. The component of the state $\prod_{p \in PROC} COUNTER$ associates a value from $COUNTER$ with each process: this value records whether that process is executing an operation and if so, what point in that operation it is up to. The $bool$ component of the state serves as a flag, indicating when the lock can be acquired.

It is very common for I/O automata that the set of states is a Cartesian product, so it is useful to introduce *state variables* to access each element of the state of an automaton. These state variables are just access names for the state type of the automaton. We introduce the state variables pc_p for each $p \in PROC$ and $flag$ where, for any $s \in states(Mutex)$, $s.flag = \pi_1(s)$ and, $pc_p = \pi_p(\pi_2(s))$. Now, it is easy to define the start states of $Mutex$:

$$\begin{aligned} start(Mutex) &= \{s \in states(Mutex) \mid s.flag = false \wedge \\ &\quad \forall p \in PROC \bullet pc_p = idle\} \end{aligned}$$

So the start states of $Mutex$ are those where no process has acquired the lock ($flag$ is false) and no process is attempting to acquire or release the lock (all the pc_p variables are *idle*).

Several of the automata presented in this thesis have process-indexed variables: these variables always represent the local state of each process, so some-

```

acquire_invp :
pre   $pc_p = idle$ 
eff   $pc_p := acquiring$ 

acquire_respp :
pre   $pc_p = acquiring \wedge flag = false$ 
eff   $pc_p := has\_lock; flag := true$ 

release_invp :
pre   $pc_p = has\_lock$ 
eff   $pc_p := releasing$ 

release_respp :
pre   $pc_p = releasing$ 
eff   $pc_p := idle; flag := false$ 

```

Figure 3.1: Transition relation of the *Mutex* automaton.

times we refer to them as *local* variables. We also refer to un-indexed variables as *global* variables.

Now we define the transition relation of *Mutex*. To do this, we will associate each action with a *precondition* and an *effect* that together specify the transitions labelled by that action. Figure 3.1 presents this association for *Mutex*.

The precondition of each action acts as a guard for the action. The precondition constrains the values taken by state variables in pre-states of transitions labelled by the action. The effect of each action is a set of *parallel assignments*, where the post-state value of the variable on the left-hand side is taken to be the value of the right-hand side expression. Variables not mentioned on the left-hand side of any assignment keep the same value. For example, the precondition and effect associated with the action *release_resp_p* entail that

$$s \xrightarrow{\text{release_resp}_p} s' \Leftrightarrow s.pc_p = \text{releasing} \wedge s'.pc_p = \text{idle} \wedge s'.flag = \text{false} \wedge \forall q \neq p \bullet s'.pc_q = s.pc_q$$

When state variables appear on the right-hand side of assignments, their values are taken to be the *pre-state* values of those variables.

Note that, given a pre-state and action there is only one possible post-state: every transition relation discussed in this thesis has this property. The parallel assignment notation used here is simpler and clearer than a more general relational notation that would be needed to specify less deterministic systems.

3.3 Sequential Datatypes

In order to describe formally the relationship between a sequential datatype and its concurrent implementations, we need a formal description of the notion of a datatype. Informally, a datatype defines two things: an interface between its members and the outside world; and some kind of specification of its responses to invocations on that interface. What follows is a simple way to describe these aspects of a datatype formally, adapted from [28, Section 9.4].

A datatype \mathcal{D} is a tuple (V, v_0, I, R, f) ; each component of this tuple is described below:

- V is the set of values of the datatype.
- $v_0 \in V$ is the datatype's initial value.
- The sets I of invocations and R of responses constitute the interface to the datatype. These invocations and responses may take parameters and are used to represent the operations the datatype supports.
- The function $f : V \times I \rightarrow V \times R$ is called the *update function*. f takes an invocation on a given value, and provides the suitable response while updating the value if necessary. It defines the behaviour of the datatype.

The primary goal of this formulation is to give a general way of describing the behaviour of automata that implement a given type. Note that the invocations and responses, which model operations, are explicit components of the datatype. This is slightly at odds with a specification style that treats operations directly as functions (or relations) over the datatype (for example [11]). However, this

approach allows us to easily define *traces* of the datatype, in close analogy with the way we have done for I/O automata. Definitions 3.4 and 3.5 are adapted from [28, Section 13.1.1]. First we define executions:

Definition 3.4 (Execution of datatype) *An execution of a datatype \mathcal{D} is a sequence beginning with $v_0 \in V$, the initial value of the datatype, and continuing with zero or more subsequences made up of an invocation, a response and a value. (So sequences have the form $\langle v_0, i_1, r_1, v_1, i_2, r_2, v_2 \dots \rangle$.) Each such subsequence i, r, v' , occurring immediately after some value v , is required to satisfy $f(i, v) = (r, v')$.*

This leads to a notion of traces:

Definition 3.5 (Trace of datatype) *A trace of a datatype \mathcal{D} is any sequence that can be obtained from an execution of \mathcal{D} by removing all the occurrences of values. Note that a trace is a sequence of alternating invocations and responses, beginning with an invocation.*

As an example of this specification style, consider the *stack* datatype. The stack contains elements of some non-empty set T . It is modelled as a sequence of elements from that set. Let the stack datatype be $Stack = (V, v_0, I, R, f)$ where

- V is the set of sequences of elements from T .
- $v_0 = \langle \rangle$, the empty sequence.
- $I = \{push(t) \mid t \in T\} \cup \{pop\}$ and $R = \{push_resp, empty\} \cup \{pop_resp(t) \mid t \in T\}$. Expressions like $push(t)$ and $pop_resp(t)$ can be interpreted as injections over the set T into the sets I or R , whose ranges are distinct from the other elements of I and R . $push(t)$ represents an invocation with the parameter t ; $pop_resp(t)$ represents the response to a previous pop invocation, with the return value t ; $push_resp$ signals that a push operation has been completed; $empty$ signals that an attempted pop operation found the stack empty.

- We will treat the left side of the sequence as the top of the stack so that in response to a push, we want to concatenate the pushed value onto the left side of the sequence. For a pop, unless the stack is empty, we want to remove and return the leftmost value in the sequence; if the stack is empty, we should do nothing to its state, but return *empty* as a response. Hence, for any $v \in V, t \in T$, define f as follows:

$$f(v, \text{push}(t)) = (\langle t \rangle \frown v, \text{push_resp})$$

$$f(\langle \rangle, \text{pop}) = (\langle \rangle, \text{empty})$$

$$f(\langle t \rangle \frown v, \text{pop}) = (v, \text{pop_resp}(t))$$

Stacks have executions like

$$\langle \langle \rangle, \text{push}(t_1), \text{push_resp}, \langle t_1 \rangle, \text{pop}, \text{pop_resp}(t_1), \langle \rangle, \dots \rangle$$

which has the following trace

$$\langle \text{push}(t_1), \text{push_resp}, \text{pop}, \text{pop_resp}(t_1), \dots \rangle$$

This specification is similar in style to the one which is used later to specify dequeues. However, there is a major difference: rather than using sequences to represent the state of a deque, we will use functions over a finite range of integers. This issue will be discussed in Chapters 4 and 5.

3.4 Correctness and Atomic Automata

Lynch [28] defines a notion of *atomic object*. In the current context, an *object* is understood to be an I/O automaton. An atomic automaton represents an implementation of a datatype that can be accessed concurrently by multiple processes. An automaton must fulfil three properties to be a concurrent implementation of a given datatype: it must have a signature which allows it to model the operations

of the given datatype; its traces must be *well-formed*; and it must be *linearizable* [22]. When an automaton A is atomic for a datatype \mathcal{D} , we say A *implements* \mathcal{D} .

The datatype an automaton implements determines the interface the automaton must provide. The automaton's input actions must be the invocations of the datatype, indexed by processes. Likewise, its external actions are the datatype's responses, indexed by processes. The interpretation is that a process indexing an invocation or response is the process invoking or returning from the operation, respectively. An automaton A implementing a type \mathcal{D} with invocations I and responses R for use by processes from a set $PROC$, will have the following external signature

$$Input(A) = \{inv_p \mid inv \in I, p \in PROC\}$$

$$Output(A) = \{resp_p \mid resp \in R, p \in PROC\}$$

When A implements some datatype, we call $Input(A)$ its invocations, and $Output(A)$ its responses.

A trace is *well-formed* if for each process, invocations and responses alternate, beginning with an invocation. To put this formally, it is useful to have a notion of *process sub-trace*. The following two definitions are adapted from [22]:

Definition 3.6 (Process sub-trace) A process sub-trace of a trace μ for $p \in PROC$, written $\mu \mid p$, is the subsequence of μ consisting of the elements of μ that are indexed by p .

Definition 3.7 (Well-formed trace) A trace μ is well-formed if, for every $p \in PROC$ the following is true:

1. $\mu \mid p$ begins with an invocation.
2. Each invocation in $\mu \mid p$, except possibly the last, is immediately followed by a response.
3. Each response in $\mu \mid p$, except possibly the last, is immediately followed by an invocation.

The *linearizability* property constrains what these invocation-response pairs can be. Linearizability is due originally to Herlihy and Wing [22, 23], and has become a very common correctness condition for concurrent objects. The idea is to make it look to each process (and the observer) as though each operation in a trace is occurring one at a time in some particular order consistent with a sequential specification of the datatype. The formal definition presented here is adapted from [22] and [28]. It is standard among developers who use linearizability as a correctness condition to present the definition found in [22] (for example [17, 8, 2, 16, 14]). [28] has the advantage of providing a tight notion of sequential specification, using sequential datatypes.

The definition of linearizability depends on the following preliminary definitions:

Definition 3.8 (Complete and incomplete operation) *Given a trace μ of some automaton A , a complete operation of μ is a pair $(inv_p, resp_p)$ with $inv_p \in Input(A)$ and $resp_p \in Output(A)$ where inv_p occurs before $resp_p$ in μ with no intervening p -indexed actions. Given a complete operation $\mathcal{O} = (inv_p, resp_p)$, the invocation of \mathcal{O} is inv_p and the response of \mathcal{O} is $resp_p$. An incomplete operation of μ is an invocation $inv_p \in Input(A)$ appearing in μ with no p -indexed actions occurring after inv_p in μ .*

Definition 3.9 *Given a trace μ of some automaton, $complete(\mu)$ is the sequence μ with all incomplete operations of μ removed.*

Definition 3.10 (Irreflexive partial order of trace, $<_\mu$) *Given a well-formed trace μ of some automaton, $<_\mu$ is the irreflexive partial order over the operations of μ defined by*

$\mathcal{O}_1 <_\mu \mathcal{O}_2$ if and only if the response of \mathcal{O}_1 occurs before the invocation of \mathcal{O}_2 in μ .

Observe that an irreflexive *total* order $<$, on complete operations that has a least element (ie., there is an operation \mathcal{O} with the property that there is no operation

\mathcal{O}' such that $\mathcal{O}' < \mathcal{O}$) or is defined over the empty set of operations, induces a sequence of invocations and responses. If $<$ has a least element, this sequence is constructed by laying out the invocation and response of each operation, in the order determined by $<$; if $<$ is defined over the empty set, then the sequence is $\langle \rangle$. Also, note that given a well-formed trace μ with some complete operation, any total order over the complete operations of that trace has a least element if it contains the irreflexive partial order $<_{\mu}$ (the set of operations with invocations occurring before the first response is finite, since the first response occurs at a finite index, and every element of this set is less than every operation not in this set) and so induces a sequence of invocations and responses.

Definition 3.11 (Linearizability of traces) *A trace μ of an automaton A is linearizable with respect to a datatype \mathcal{D} if it can be extended to a trace μ' by appending actions of A , such that there exists an irreflexive total order $<$ over the operations of $\text{complete}(\mu')$ satisfying the following conditions:*

1. *A trace of \mathcal{D} is obtained by removing the process indices from the sequence induced by $<$.*
2. *For every pair of operations $\mathcal{O}_1, \mathcal{O}_2$ of μ' , $\mathcal{O}_1 <_{\mu'} \mathcal{O}_2$ implies $\mathcal{O}_1 < \mathcal{O}_2$.*

There are two sets of decisions which must be made to show that a given trace μ can be linearized: the choice of the extension μ' and the construction of the order $<$. The trace μ' should be extended by providing responses for incomplete operations whose ‘effects’ have been ‘seen’ by other operations. The construction of $<$ can be achieved by choosing, for each operation, a *linearization point* between the invocation and response of that operation: the linearization point of an operation is a point in the trace where we think of the operation as taking effect; $<$ is the order induced by the relative positions of these linearization points. These issues will be made clear with an example.

With reference to the datatype *Stack* introduced in the previous section, suppose we are given a trace μ of some automaton meant to implement the stack

datatype where

$$\mu = \langle \text{push}_{p_1}(t_1), \text{push}_{p_2}(t_2), \text{pop}_{p_3}, \text{push_resp}_{p_2}, \\ \text{pop_resp}_{p_3}(t_2), \text{pop}_{p_3}, \text{pop_resp}_{p_3}(t_1) \rangle$$

Figure 3.2 illustrates this trace.

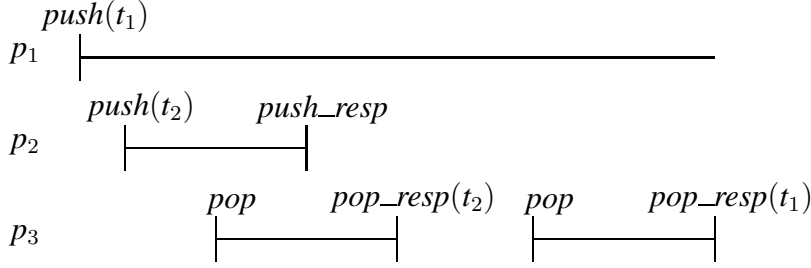


Figure 3.2: The operations of the example trace.

There are three complete operations: $\mathcal{O}_1 = (\text{push}_{p_2}(t_2), \text{push_resp}_{p_2})$, $\mathcal{O}_2 = (\text{pop}_{p_3}, \text{pop_resp}_{p_3}(t_2))$ and $\mathcal{O}_3 = (\text{pop}_{p_3}, \text{pop_resp}_{p_3}(t_1))$. Also, there is one incomplete operation: $\text{push}_{p_1}(t_1)$. The response of \mathcal{O}_3 returns the value t_1 (it has ‘seen’ the effect of $\text{push}_{p_1}(t_1)$), so we should extend μ to include a response for $\text{push}_{p_1}(t_1)$. So define

$$\mu' = \langle \text{push}_{p_1}(t_1), \text{push}_{p_2}(t_2), \text{pop}_{p_3}, \text{push_resp}_{p_2}, \\ \text{pop_resp}_{p_3}(t_2), \text{pop}_{p_3}, \text{pop_resp}_{p_3}(t_1), \text{push_resp}_{p_1} \rangle$$

and let $\mathcal{O}_4 = (\text{push}_{p_1}(t_1), \text{push_resp})$.

Every operation of μ' is complete so that $\text{complete}(\mu') = \mu'$; all we need to do now is construct the order $<$ to satisfy clause (1) of Definition 3.11. Since the response of \mathcal{O}_2 returns the value t_2 we need to place \mathcal{O}_1 's linearization point before that of \mathcal{O}_2 (because t_2 needs to be in the stack for \mathcal{O}_2 to be able to return the value). Also, \mathcal{O}_3 returns t_1 , the value pushed by the incomplete invocation $\text{push}_{p_1}(t_1)$, so we should choose a linearization point for \mathcal{O}_4 before that of \mathcal{O}_3 . Figure 3.3 illustrates one possibility for a set linearization points consistent with these constraints.

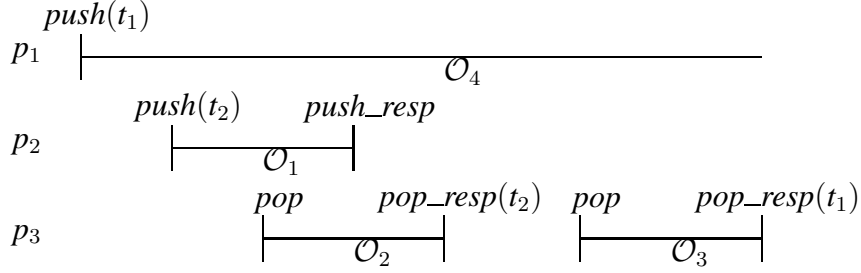


Figure 3.3: The operations of the example trace.

This set of linearization points induces the following order on the operations of μ' :

$$\mathcal{O}_1 < \mathcal{O}_2 < \mathcal{O}_4 < \mathcal{O}_3$$

This order induces the following trace of the *Stack* datatype:

$$\langle \text{push}(t_2), \text{push_resp}, \text{pop}, \text{pop_resp}(t_2), \text{push}(t_1), \text{push_resp}, \text{pop}, \text{pop_resp}(t_1) \rangle$$

This is the trace of the following execution of *Stack*:

$$\begin{aligned} &\langle \langle \rangle, \text{push}(t_2), \text{push_resp}, \langle t_2 \rangle, \text{pop}, \text{pop_resp}(t_2), \\ &\quad \langle \rangle, \text{push}(t_1), \text{push_resp}, \langle t_1 \rangle, \\ &\quad \text{pop}, \text{pop_resp}(t_1), \langle \rangle \rangle \end{aligned}$$

There is another possible choice for the linearization point chosen for the incomplete push: we could have stipulated that it occurred before the linearization point of \mathcal{O}_1 and still obtained a valid linearization. There is often a substantial degree of freedom in linearizing traces.

Note that choosing linearization points between the invocations and responses of each operation guaranteed that the resulting order contained $<_{\mu}$ (so we satisfied clause (2) of Definition 3.11).

Every trace of an atomic automaton is linearizable:

Definition 3.12 (Linearizability of an automaton) *An automaton A is linearizable with respect to a datatype \mathcal{D} if every trace of A is linearizable with respect to \mathcal{D} .*

So, an atomic automaton for a datatype has the correct interface; has well-formed traces; and is linearizable with respect to the datatype. We need a tractable proof method to show that a given automaton is atomic. This method is developed in the following sections. It consists of two steps: construct an automaton which is known to be atomic for the given datatype; then show that the traces of some other given automaton are included in the set of traces of the constructed automaton. The second step uses *simulation relations* which provide a way of reasoning about trace-inclusion based only on local properties of individual transitions. Simulation relations are described in Section 3.7. The first step uses a ‘canonical automaton’ construction described in the next section.

3.5 Canonical Automata

This section describes a method which takes a datatype \mathcal{D} and constructs an I/O automaton that is atomic for \mathcal{D} . This automaton is called a *canonical automaton* for \mathcal{D} and has the property that it can generate *all* the linearizable traces of \mathcal{D} , which means that the traces of *any* automaton that is atomic for \mathcal{D} will be contained in the traces of the canonical automaton. This property tells us that using the canonical automaton will not restrict the kinds of implementations that we can verify.

The canonical automaton for \mathcal{D} is fairly straightforward. Its global state is just some element from the value set of \mathcal{D} . It has actions for all of \mathcal{D} ’s invocations and responses, and internal actions that atomically update the current value according to each invocation of \mathcal{D} . The state also contains a program counter variable for each process, recording which processes are active and the invocation that each process is trying to respond to.

The canonical automaton for the datatype $\mathcal{D} = (V, v_0, I, R, f)$, constructed for access by processes from the set $PROC$, has the following signature:

$$Input(C) = \{inv_p \mid inv \in I, p \in PROC\}$$

$$Output(C) = \{resp_p \mid resp \in R, p \in PROC\}$$

$$Internal(C) = \{do_inv_p \mid inv \in I, p \in PROC\}$$

The labels do_inv_p must all be distinct from each other and distinct from everything in $Input(C) \cup Output(C)$.

$$states(C) = V \times \prod_{p \in PROC} COUNTER$$

where $COUNTER = \{idle\} \cup I \cup R$.

The automaton's state is accessed with the variables val and pc_p for each $p \in PROC$ where $s.val = \pi_1(s)$ and $s.pc_p = \pi_p(\pi_2(s))$ for all $s \in states(C)$. The variable pc_p is used to record whether process p is executing an operation and if so, which operation that is. A process with an *idle* program counter is not currently executing any operation. Note that the *COUNTER* set contains entire invocations and responses, including arguments (if any).

The start state of C is the state where the val variable of the automaton is the initial value of \mathcal{D} and each process is idle:

$$start(C) = \{s \in states(C) \mid s.val = v_0 \wedge \forall p \in PROC \bullet s.pc_p = idle\}$$

The transition relation of the canonical automaton is very simple. In essence, each idle process may execute an invocation of an operation; some time later it executes the operation atomically on the value in the automaton's state; some time later it returns the response associated with that operation. The transition relation is as follows:

$$\begin{array}{ll} inv_p : & do_inv_p : \\ \mathbf{pre} \quad pc_p = idle & \mathbf{pre} \quad pc_p = inv \\ \mathbf{eff} \quad pc_p := inv & \mathbf{eff} \quad (val, pc_p) := f(val, inv) \\ \\ resp_p : & \\ \mathbf{pre} \quad pc_p = resp & \\ \mathbf{eff} \quad pc_p := idle & \end{array}$$

Note that here inv_p , do_inv_p and $resp_p$ all represent *families* of actions: for example, inv_p represents all the $|PROC| \times |I|$ process labelled invocations.

The construction of canonical automata presented here differs from the construction presented in [28, Section 13.2] in certain respects. The use of *process*-indexed invocations and responses differs from the indexing used in [28]. Lynch uses indices on invocations and responses, but there the interpretation is that the indices represent *ports*. This difference is partly attributable to the fact that [28] is concerned with distributed systems, where as here the concern is multi-processor systems. However, there is a more substantive difference between the two constructions of canonical automata, related to the relaxation here of the *input-enabled* condition on I/O automata. The input-enabled condition insists that every input action is always enabled: that is, in every state of the automaton, every input action labels some transition whose precondition is met. This means that I/O automata can *receive* every input from the external environment at all times and allows the definition of a composition operator over automata which has several desirable properties. However, we are not concerned with composition of automata, only the implementation of datatypes, so the canonical automata defined here are *not* input enabled: invocations are enabled when the invoking process is *idle*. This approach provides an easy way to guarantee *well-formedness* of the traces of canonical automata (see Lemma 3.1).

3.6 Properties of Canonical Automata

This section presents results concerning atomicity of canonical automata, and explores theoretically their usefulness in the verification of implementations of datatypes. The results in this section are fairly straightforward and all have analogues in [28]. Their importance lies in showing soundness and the breadth of applicability of the proof method being developed.

For the rest of this section, fix a datatype $\mathcal{D} = (V, v_0, I, R, f)$. Let C be the canonical automaton for \mathcal{D} , constructed as in the previous section.

3.6.1 Atomicity of the Canonical Automaton

Recall that an atomic automaton has three properties: it has the correct external signature; its traces are well formed; and its traces are linearizable. Note that the external signature of C is exactly that required by atomic automata: the *Input* and *Output* actions are process-indexed versions of the invocations and responses of \mathcal{D} .

Our canonical automaton produces only well-formed traces by virtue of the preconditions on each transition:

Lemma 3.1 (Well-formedness of canonical automaton's traces.) *All the traces of C are well-formed.*

Consider some trace μ of C and process p :

- For each $s \in \text{start}(C)$, $s.pc_p = \text{idle}$. So the first p -indexed action in μ must be an invocation, since the precondition of every other p -indexed action requires that $s.pc_p \neq \text{idle}$.
- Assume there is an occurrence of a p -indexed invocation in μ . Each state s appearing in the execution which produced μ after this occurrence will have $s.pc_p \neq \text{idle}$ until an occurrence of a p -indexed response. Hence if there is an action following the p -indexed invocation in μ it must be a response.
- A similar consideration shows that any action following a response in $\mu \upharpoonright p$ is an invocation.

Now all we need to see that C is atomic is that it is linearizable (cf. [28], Theorem 13.3):

Lemma 3.2 (Linearizability of canonical automaton) *C is linearizable.*

[28] presents a proof of this theorem for a slightly different canonical automaton, but the proof carries directly to the automata discussed here. The basic motivation is that an order for the operations in any execution can be constructed according to the order of *do* actions in that execution. Since the transitions labelled

by *do* actions are just applications of the update function of the datatype being implemented to a member of that datatype, this order induces a valid sequential execution.

3.6.2 Completeness of the Canonical Automaton

The construction of the canonical automaton represents an important step in showing how to prove that some automaton is atomic for some datatype. Consider an automaton A such that A has the correct signature to be atomic for \mathcal{D} (and so has the same signature as \mathcal{D} 's canonical automaton). If we know that $A \leq_T C$ (that is, every trace of A is also a trace of C) then we know that A is atomic for \mathcal{D} : if $\text{traces}(A) \subseteq \text{traces}(C)$ and all members of $\text{traces}(C)$ are well-formed and linearizable then all members of $\text{traces}(A)$ are well-formed and linearizable.

This observation gives us a proof strategy for showing that a given automaton implements a particular datatype. How best to show trace inclusion from one automaton to another? A popular and effective approach is to use *simulation relations*, which are discussed in the following section. There is, however, another important question. Can we guarantee that C has *every* well-formed, linearizable trace? This is a very desirable property to have: if we have some automaton A meant to implement \mathcal{D} we know that if we are unable to show $A \leq_T C$, then either we are not clever enough or our implementation contains a bug. We do not have to find some other way to specify well-formed and linearizable traces. The following theorem formalises a sense in which the canonical automaton is *complete* (cf. [28], Theorem 13.5):

Theorem 3.1 (Completeness of the canonical automaton) *Let A be an I/O automaton which is atomic for \mathcal{D} . Then $A \leq_T C$.*

Again, [28] provides a proof. The idea is to observe that an execution of C can be constructed by associating *do* actions with linearization points. Briefly, for every trace of an automaton implementing \mathcal{D} there is a total order over the operations of that trace, witnessing its linearizability. An execution of C can be constructed,

containing invocation and response actions in the order given by the trace, with internal *do* actions in the order given by the linearization points.

3.7 Simulation Relations and Trace Inclusion

This section defines several notions of *simulation relation*. The existence of a simulation relation between two automata guarantees trace inclusion between the automata. There are several different kinds of simulation relation, differing from one another in their range of applicability and complexity. [29] provides a good survey of the classes of simulation relations available.

This section describes an important class of simulation relations: *forward simulations*. We present three definitions of forward simulation to allow a clear development of the ideas involved and to help make the point that the notion of forward simulation admits modification for specific purposes.

For the remainder of this section, let A and B be I/O automata.

3.7.1 General Forward Simulation and Representation

The following definition is adapted from [36].

Definition 3.13 (Forward Simulation, first version) *A forward simulation R from A to B is a relation over $states(A)$ and $states(B)$ satisfying:*

1. *For all $s_A \in start(A)$, there is some $s_B \in start(B)$ such that $R(s_A, s_B)$.*
2. *For all $s_A, s'_A \in states(A)$, if $s_A \xrightarrow{a} s'_A$ and $a \in external(A)$, then for all s_B such that $R(s_A, s_B)$, there is some $s'_B \in states(B)$ such that $R(s'_A, s'_B)$ and $s_B \xrightarrow{a} s'_B$.*
3. *For all $s_A, s'_A \in states(A)$, if $s_A \xrightarrow{a} s'_A$ and $a \notin external(A)$, then for all s_B such that $R(s_A, s_B)$, there is some $s'_B \in states(B)$ and execution fragment β such that $R(s'_A, s'_B)$, $s_B \xrightarrow{\beta} s'_B$ and β contains no external actions. Note that β may be the empty execution fragment.*

The automaton A in the above definition is ‘more concrete’ and the automaton B is ‘more abstract’: because of this, when discussing a particular simulation relation, we call A the *concrete automaton* and B the *abstract automaton*.

The basic idea behind forward simulation is that the simulation relation specifies the sense in which the states of the concrete automaton *represent* states of the abstract automaton: for $s_A \in \text{states}(A)$ and simulation relation R , $R[s_A]$ is the set of abstract states which s_A represents. From this perspective, the three conditions on R can be re-stated as follows:

1. Each $s_A \in \text{start}(A)$ represents some abstract start state.
2. If s_A represents s_B , and $s_A \xrightarrow{a} s'_A$ with a external, then s'_A represents some state of B which can be reached from s_B by a transition labelled with a .
3. If s_A represents s_B , and $s_A \xrightarrow{a} s'_A$ with a internal, then s'_A represents some state of B which can be reached from s_B without taking external transitions.

Put together, the three conditions allow us to construct for any execution of A , an execution of B with the same trace. We do this with an induction along the length of executions of A with the hypothesis that each state along executions of A represents a state of B ; furthermore, this abstract state can be reached by an abstract execution fragment with the same external actions in the same order (ie. the same *trace*) as that which led to the representing concrete state. (1) above gives us the base case and as the length of executions increases, the hypothesis is preserved by applying one of (2) or (3). These observations are the basis of the proof of the following soundness property:

Theorem 3.2 (Forward simulation implies trace inclusion) *If R is a forward simulation from A to B , in the sense of Definition 3.13, then $A \leq_T B$.*

See [27] for a proof.

So, a forward simulation between two automata guarantees trace inclusion from one to the other.

3.7.2 Reachability and Forward Simulations

Improvements can be made to the definition of forward simulation as presented so far to facilitate a verification attempt. Critically, we must incorporate the concept of *reachability*. Reachability can be defined inductively as follows:

Definition 3.14 (Reachable states) *For any automaton A , the set of reachable states, $reach(A)$ is the least set satisfying:*

- *For all $s \in states(A)$, if $s \in start(A)$ then $s \in reach(A)$.*
- *For all $s' \in states(A)$, if there exists some state $s \in reach(A)$ and action a such that $s \xrightarrow{a}_A s'$ then $s' \in reach(A)$.*

The advantage of incorporating reachability into our definition is that our proofs can use *invariants*. An invariant of an automaton A is a predicate over states of A that is satisfied by all states in $reach(A)$. Constructing proofs using invariants is a standard way of verifying properties of algorithms [28] and the verification attempt described in Chapter 5 makes some use of invariants. A predicate can be shown to be invariant using a proof rule based on an induction over the structure of the set $reach$ of an automaton. For any automaton A and predicate I over $states(A)$, I is an invariant of A if

- For all $s \in states(A)$, $s \in start(A)$ implies $I(s)$.
- For all $s, s' \in reach(A)$, $a \in acts(A)$, $s \xrightarrow{a}_A s'$ and $I(s)$ implies $I(s')$.

Incorporating reachability into forward simulation is straightforward. We only need to make claims about states of the concrete automaton which are reachable. The following definition is adapted from [29] and [36]:

Definition 3.15 (Forward Simulation, second version) *A forward simulation R from A to B is a relation over $states(A)$ and $states(B)$ satisfying:*

1. *For all $s_A \in start(A)$, there is some $s_B \in start(B)$ such that $R(s_A, s_B)$.*

2. For all $s_A, s'_A \in \text{reach}(A)$, if $s_A \xrightarrow{a} s'_A$ and $a \in \text{external}(A)$, then for all $s_B \in \text{reach}(B)$ such that $R(s_A, s_B)$, there is some $s'_B \in \text{states}(B)$ such that $R(s'_A, s'_B)$ and $s_B \xrightarrow{a} s'_B$.
3. For all $s_A, s'_A \in \text{reach}(A)$, if $s_A \xrightarrow{a} s'_A$ and $a \notin \text{external}(A)$, then for all $s_B \in \text{reach}(B)$ such that $R(s_A, s_B)$, there is some $s'_B \in \text{states}(B)$ and execution fragment β such that $R(s'_A, s'_B)$, $s_B \xRightarrow{\beta} s'_B$ and β contains no external actions.

This notion of forward simulation has the same soundness property as the first:

Theorem 3.3 *If R is a forward simulation from A to B , in the sense of Definition 3.15, then $A \leq_T B$.*

Every state along an execution is reachable, so we only need to consider these states as we construct an execution of B with the same trace as a given execution of A .

3.7.3 Simple Forward Simulation

The final development provides a definition of forward simulation that is simpler to use but has a smaller range of applicability. The clauses labelled (3) in Definitions 3.13 and 3.15 allow us to associate with an internal transition of the concrete automaton an entire execution fragment of the abstract automaton not containing external transitions. However, for the verification of the Snark algorithm, we do not need this much flexibility. Moreover, expressing such a property in a formal logic (such as that used in PVS) would introduce unnecessary complexity. For these reasons we will replace condition (3) with a simpler condition: if the concrete automaton takes an internal transition, then the abstract automaton must take one internal transition or take no transition. The modified definition of forward simulation is essentially that of [29] and runs as follows:

Definition 3.16 (Forward Simulation, third version) *A forward simulation R from A to B is a relation over $\text{states}(A)$ and $\text{states}(B)$ satisfying:*

1. For all $s_A \in \text{start}(A)$, there is some $s_B \in \text{start}(B)$ such that $R(s_A, s_B)$.
2. For all $s_A, s'_A \in \text{reach}(A)$, if $s_A \xrightarrow{a} s'_A$ and $a \in \text{external}(A)$, then for all reachable $s_B \in \text{reach}(B)$ such that $R(s_A, s_B)$, there is some $s'_B \in \text{states}(B)$ such that $R(s'_A, s'_B)$ and $s_B \xrightarrow{a} s'_B$.
3. For all $s_A, s'_A \in \text{reach}(A)$, if $s_A \xrightarrow{a} s'_A$ and $a \notin \text{external}(A)$, then for all $s_B \in \text{reach}(B)$ such that $R(s_A, s_B)$, either
 - (a) there is some $s'_B \in \text{states}(B)$ and action $b \notin \text{external}(B)$, such that $s_B \xrightarrow{b} s'_B$ and $R(s'_A, s'_B)$, or
 - (b) $R(s'_A, s_B)$

The third version of forward simulation shares the same soundness property as the first two. This is clear because item (3) in Definition 3.15 is implied by Definition 3.16. The relevant theorem is stated for completeness:

Theorem 3.4 *If R is a forward simulation from A to B , in the sense of 3.16, then $A \leq_T B$.*

The notion of forward simulation given in Definition 3.16 is the one used in the verification attempt described in Chapter 5. However, forward simulations are not complete for trace inclusion. As will be discussed in Chapter 8, one of the fixes for the Snark algorithm, presented in Chapter 7, provides an example where we believe that the implementation is correct but there is no forward simulation between the implementation and the canonical automaton.

Chapter 4

Correctness and Representation for the Snark Algorithm

Chapter 3 described the underlying specification and modelling strategy used in this thesis, based on I/O automata. This chapter describes the I/O automata used in the attempted verification of Snark. Section 4.1 defines the deque datatype and the canonical automaton for deques; it also defines a slight modification of this canonical automaton which exploits the symmetry in the Snark algorithm.

Section 4.2 presents the automaton which is used to model the Snark algorithm. This automaton makes heavy use of a structure which represents a dynamic heap, described in Section 4.2.2. A notation for describing the states of this heap model is defined in Section 4.2.3.

4.1 Correctness Requirements for the Snark Algorithm

We need to do two things to specify the behaviour of a concurrent implementation of a deque: define the deque data-type and construct the canonical automaton.

4.1.1 The Deque Data-type

Section 2.2.1 presented an informal definition of the deque datatype: a formal definition is presented here based on the notion of *sequential datatypes* presented in Chapter 3.

The set of values of the deque datatype is represented as a triple made up of a function from \mathbb{Z} into a set T that represents the type of the elements of the deque, and two integers *bot* and *top* which represent the lower and upper bounds of the deque. This is in contrast to the situation with the *Stack* datatype, which used a sequence. However, in the PVS language, sequences are straightforwardly represented using a pair made up of a function from \mathbb{N} into some other type and an upper bound¹: in our case it is advantageous to use \mathbb{Z} as the domain of the function because operations on the left and the right can then be symmetric.

Let t_0 be some element of T . The deque datatype is $Deque = (V, v_0, I, R, f)$ where:

- $V = \{v \in (\mathbb{Z} \rightarrow T) \times \mathbb{Z} \times \mathbb{Z} \mid \pi_2(v) < \pi_3(v)\}$. Given $v \in V$, let $v.seq = \pi_1(v)$, $v.bot = \pi_2(v)$ and $v.top = \pi_3(v)$. The integer *bot* is the greatest unused position less than *top* (the bottom of the deque); *top* is the least unused index greater than *bot* (the top of the deque).²
- $v_0 = (emp, 0, 1)$, where for all $i \in \mathbb{Z}$, $emp(i) = t_0$.
- The invocations are

$$I = \{push_left(t) \mid t \in T\} \cup \{push_right(t) \mid t \in T\} \cup \{pop_left, pop_right\}$$

The responses are

$$R = \{push_resp, resp_empty\} \cup \{pop_resp(t) \mid t \in T\}$$

¹PVS does support the ability to express algebraic specifications, which could be used to represent sequences. However, this would complicate the statement of the simulation relation given in Chapter 5.

²*bot* and *top* are *unused* indexes to prevent *top* needing to be less than *bot* in an empty deque.

Similarly to the stack datatype described in Section 3.3, $push_left(t)$ and $push_right(t)$ represent parameterised invocations of the push operations; $pop_resp(t)$ represents a response to a pop invocation with return value t ; $push_resp$ signals the termination of a push operation; $resp_empty$ indicates that a pop operation found the deque empty.

- A push on the left should cause the bottom of the sequence to be filled with the pushed value, and bot to be decremented. A pop on the left should behave symmetrically. That is bot should be incremented, and the value to its right returned. Right-side operations should be symmetric with their left-side counterparts. So define the update function f as follows:

$$f(v, push_left(t)) = ((v.seq[bot := t], v.bot - 1, v.top), push_resp)$$

$$f(v, push_right(t)) = ((v.seq[top := t], v.bot, v.top + 1), push_resp)$$

$$f(v, pop_left) = \begin{cases} (v, resp_empty) & \text{if } v.bot = v.top - 1 \\ ((v.seq, v.bot + 1, v.top), \\ pop_resp(v.seq(v.bot + 1))) & \text{otherwise} \end{cases}$$

$$f(v, pop_right) = \begin{cases} (v, resp_empty) & \text{if } v.bot = v.top - 1 \\ ((v.seq, v.bot, v.top - 1), \\ pop_resp(v.seq(v.top - 1))) & \text{otherwise} \end{cases}$$

4.1.2 The Canonical Automaton for the Deque

Following the discussion in Section 3.5, the canonical automaton for the deque can be constructed mechanically. Given a set of processes $PROC$, a type T , and the constants V, I, R and f defined above, the canonical automaton C is defined as follows:

$$Input(C) = \bigcup_{t \in T, p \in PROC} \{push_left_p(t), push_right_p(t), pop_left_p, pop_right_p\}$$

$$Output(C) = \bigcup_{t \in T, p \in PROC} \{push_resp_p, pop_resp(t)_p, resp_empty_p\}$$

$$Internal(C) = \bigcup_{t \in T, p \in PROC} \{do_push_left_p(t), \\ do_push_right_p(t), do_pop_left_p, do_pop_right_p\}$$

The states of C are

$$states(C) = V \times \prod_{p \in PROC} COUNTER$$

where $COUNTER = \{idle\} \cup I \cup R$.

The automaton has state variables pc_p for each $p \in PROC$ and deq where for any state s , $s.deq = \pi_1(s)$ and $s.pc_p = \pi_p(\pi_2(s))$.

The start states of C are

$$start(C) = \{s \in states(C) \mid s.deq = v_0 \wedge \forall p \in PROC \bullet s.pc_p = idle\}$$

The transition relation, described in the notation of Section 3.2 is presented in Figure 4.1 on page 59.

4.1.3 A Symmetric Deque Automaton

The canonical automaton described so far would suffice for a verification of the Snark algorithm. However, we can reduce redundancy in the proof by exploiting the symmetry in the Snark algorithm. The following automaton, *DeqAut*, uses pc variables to record whether a process is active and if so, how far through its operation the process is, but uses a *side* variable for each process to record the side of the deque that each active process is operating on. This will halve the number of internal actions in the automaton. This is a very modest simplification for the abstract automaton but it will make our Snark automaton substantially simpler than it would otherwise have been.

Let $SIDE = \{Left, Right\}$. The signature of the modified automaton is as follows (note that the *Internal* actions are the only part of the signature that has

<i>push_left_p</i> (<i>t</i>) :	<i>push_right_p</i> (<i>t</i>) :
pre $pc_p = idle$	pre $pc_p = idle$
eff $pc_p := push_left(t)$	eff $pc_p := push_right(t)$
<i>pop_left_p</i> :	<i>pop_right_p</i> :
pre $pc_p = idle$	pre $pc_p = idle$
eff $pc_p := pop_left$	eff $pc_p := pop_right$
<i>do_push_left_p</i> (<i>t</i>) :	<i>do_push_right_p</i> (<i>t</i>) :
pre $pc_p = push_left(t)$	pre $pc_p = push_right(t)$
eff $(deq, pc_p) :=$ $f(deq, push_left(t))$	eff $(deq, pc_p) :=$ $f(deq, push_right(t))$
<i>do_pop_left_p</i> :	<i>do_pop_right_p</i> :
pre $pc_p = pop_left$	pre $pc_p = pop_right$
eff $(deq, pc_p) :=$ $f(deq, pop_left)$	eff $(deq, pc_p) :=$ $f(deq, pop_right)$
<i>push_resp_p</i> :	<i>pop_resp_p</i> (<i>t</i>) :
pre $pc_p = push_resp$	pre $pc_p = pop_resp(t)$
eff $pc_p := idle$	eff $pc_p := idle$
<i>resp_empty_p</i> :	
pre $pc_p = resp_empty$	
eff $pc_p := idle$	

Figure 4.1: Canonical Deque Automaton - unmodified.

changed):

$$Input(DequeAut) = \bigcup_{t \in T, p \in PROC} \{push_left_p(t), push_right_p(t), pop_left_p, pop_right_p\}$$

$$Output(DequeAut) = \bigcup_{t \in T, p \in PROC} \{push_resp_p, pop_resp_p(t), empty_p\}$$

$$Internal(DequeAut) = \bigcup_{t \in T, p \in PROC} \{do_push_p(t), do_pop_p\}$$

The Snark algorithm, as described in Section 2.2, allows a pushing process to return "full" indicating that it was unable to allocate a new node. This

suggests that *DeqAut* should have an external action $resp_full_p$: the question is, how should *DeqAut* be modified to specify when this response is allowed to occur? We could modify the deque datatype so that its update function became a *relation*, which could non-deterministically choose to return $resp_full$ in response to a push invocation. However, this would create a problem of underspecification: any automaton with the appropriate signature, whose traces were well-formed sequences of push and pop invocations and the responses $resp_full_p$ and $resp_empty_p$, would implement such a datatype. Another possibility is that *DeqAut* could non-deterministically decide to return $resp_full_p$ in any state where $pc_p = do_push(t)$ is true for some p and t . Apart from the fact that *DeqAut* would no longer be a canonical automaton for the deque datatype, this would create the same problem of underspecification as the first option: the traces of an automaton that never successfully pushes values would be included in the traces of the modified *DeqAut*.

The solution presented in this chapter is to avoid the issue: we use the signature defined above and define the Snark automaton so that it is always able to allocate new nodes and thus never has to return "full". In the Snark algorithm, a pushing process does not operate on any global data if it finds the heap full, so not modelling this behaviour in a verification attempt does not reduce the level of assurance that the verification would provide. The mechanism by which the Snark automaton guarantees that there is always room in the heap are described in Section 4.2.2.

The states of *DeqAut* are like those of *C* above, but with some extra local state for each process to record which side of the deque an operation should act on:

$$states(DeqAut) = V \times \left(\prod_{p \in PROC} COUNTER' \right) \times \left(\prod_{p \in PROC} SIDE \right)$$

where *COUNTER'* is like *COUNTER* but with the information about whether pushes or pops are occurring on the left or right removed. That is:

$$COUNTER' = \{idle\} \cup \{pop, push_resp\} \cup \bigcup_{t \in T} \{push(t), pop_resp(t)\}$$

The variables *deq* and pc_p represent the same elements of the state as for *C*. Note that *COUNTER'* still contains the set of responses for dequeues, *R*: this means

that the pc_p variables can be assigned values from the second component of applications of the update function f , while respecting the type of pc_p . We also need the variables $side_p$ for each $p \in PROC$, where $s.side_p = \pi_p(\pi_3(s))$ represents the side that each active process is operating on.

The start states of *DeqAut* are defined as for C :

$$start(DeqAut) = \{s \in states(C) \mid s.deq = v_0 \wedge \forall p \in PROC \bullet s.pc_p = idle\}$$

The transition relation for *DeqAut* is presented in Figure 4.2 on page 62. The relation makes use of the functions $push : V \times T \times SIDE \rightarrow V \times R$ and $pop : V \times SIDE \rightarrow V \times R$, defined for all $v \in V$, $t \in T$ and $s \in SIDE$ as follows:

$$push(v, t, s) = \begin{cases} f(v, push_left(t)) & \text{if } s = Left \\ f(v, push_right(t)) & \text{otherwise} \end{cases}$$

$$pop(v, s) = \begin{cases} f(v, pop_left) & \text{if } s = Left \\ f(v, pop_right) & \text{otherwise} \end{cases}$$

It should be clear that *DeqAut* has the same set of traces as C , the canonical automaton for deques. The transition relation of *DeqAut* is essentially a notational variation on the actions of C : rather than carrying the side of the operation on the transition label, the side is carried in the local state of the process.

4.2 The Snark Automaton

This section describes the I/O automaton *SnarkAut*, that represents the Snark algorithm. Section 4.2.1 defines the Snark automaton's signature; Section 4.2.2 defines the model of the heap used in the Snark automaton; Section 4.2.3 describes some simple notation for talking about heap states; Section 4.2.4 defines the states of the Snark automaton and its state variables; Section 4.2.5 describes the transition relation.

The Snark automaton described here uses *side* variables in a similar manner to those used in *DeqAut* (see Sections 4.2.4 and 4.2.5). These variables do not appear

<pre> push_left(t) : pre pc_p = idle eff pc_p := push(t); side_p := Left </pre>	<pre> push_right_p(t) : pre pc_p = idle eff pc_p := push(t); side_p := Right </pre>
<pre> pop_left_p : pre pc_p = idle eff pc_p := pop; side_p := Left </pre>	<pre> pop_right_p : pre pc_p = idle eff pc_p := pop; side_p := Right </pre>
<pre> do_push_p(t) : pre pc_p = push(t) eff (deq, pc_p) := push(deq, t, side_p) </pre>	<pre> do_pop_p : pre pc_p = pop eff (deq, pc_p) := pop(deq, side_p) </pre>
<pre> push_resp_p : pre pc_p = push_resp eff pc_p := idle </pre>	<pre> pop_resp_p(t) : pre pc_p = pop_resp(t) eff pc_p := idle </pre>
<pre> resp_empty_p : pre pc_p = resp_empty eff pc_p := idle </pre>	

Figure 4.2: Transitions for *DeqAut*

in the code of the Snark algorithm and would not appear in an automaton developed using a more direct translation from the code. However, as with *DeqAut*, the difference is essentially notational.

4.2.1 The Signature of the Snark Automaton

SnarkAut has the same external signature as *DeqAut*:

$$Input(SnarkAut) = \bigcup_{t \in T, p \in PROC} \{push_left_p(t), push_right_p(t), pop_left_p, pop_right_p\}$$

$$Output(SnarkAut) = \bigcup_{t \in T, p \in PROC} \{push_resp_p, pop_resp_p(t), resp_empty_p\}$$

Note that we are using the same set of processes as that used for *DeqAut*.

SnarkAut has a large number of internal actions:

$$\begin{aligned}
& \text{Internal}(\text{SnarkAut}) = \\
& \bigcup_{p \in \text{PROC}} \{ \text{pop_3}_p, \text{pop_4}_p, \text{pop_5_yes}_p, \text{pop_5_no}_p, \\
& \quad \text{pop_6_yes}_p, \text{pop_6_no}_p, \text{pop_8_yes}_p, \text{pop_8_no}_p, \\
& \quad \text{pop_9_yes}_p, \text{pop_9_no}_p, \text{pop_12}_p, \text{pop_13_yes}_p, \\
& \quad \text{pop_13_no}_p, \text{pop_14}_p, \text{pop_15}_p, \text{push_2}_p, \\
& \quad \text{push_4}_p, \text{push_5}_p, \text{push_7}_p, \text{push_8}_p \\
& \quad \text{push_9_yes}_p, \text{push_9_no}_p, \text{push_10}_p, \\
& \quad \text{push_11}_p, \text{push_12_yes}_p, \text{push_12_no}_p, \\
& \quad \text{push_15}_p, \text{push_16_yes}_p, \text{push_16_no}_p \}
\end{aligned}$$

Each p -indexed action is associated with a line of code: for example, a transition labelled pop_4_p represents p executing line 4 of one of the pop routines (the line numbers referred to are those presented in Figures 2.7, 2.8, 2.9 and 2.10). Note that certain lines of code are not matched with corresponding internal actions:

- No actions exist for lines 13 and 17 of the push routines. These `return` instructions are represented by external actions of the automaton.
- Line 6 of the push routines and line 2 of the pop routines (the heads of the `while` loops) are not directly represented. This is because the control-flow specified by the `while` loops will be represented directly in the Snark automaton's transitions. This will be described in more detail below.
- Line 3 of the push routines (where a process tests if the heap is full) has no associated action. This is because the heap of *SnarkAut* never becomes full.
- Line 16 of the pop routines (where a process sets the value field of a node to `null`) is not modelled. This line is 'optional' in the definition of the Snark algorithm (see Section 2.2).

The conditional instructions are each represented by two sets of labels: one for the successful case (for example pop_6_yes_p); one for the unsuccessful case (pop_6_no_p). The conditions themselves become part of the precondition of each action.

Note that the internal actions make no reference to whether a line of `popLeft` or `popRight` is being modelled. This is because *SnarkAut* records this information in *side* variables in the same way as *DeqAut* does.

4.2.2 Modelling the Heap

The goal of the heap model discussed here is to represent the dynamically allocatable memory on which Snark operates. To do this, we use some infinite set *POINTER* which represents the set of pointer values. The idea is that these pointers point to records. Each record possesses a number of fields, each containing some value, which may be a pointer or a value in some type *T*, the type that the deque contains. We use the set $FIELD = \{L, R, V, Z\}$, to represent these fields: the fields *V*, *L* and *R* correspond to the fields of the same name in the Snark algorithm; the field *Z* is used to obtain the effect of the *address-of* operator (written $\&$) and will be explained later in this section. *FIELD* should be explicitly partitioned into two sets, depending on whether each field accesses a pointer or a member of *T*: let $FIELD_T = \{V\}$ be the value field and $FIELD_P = \{L, R, Z\}$ be the pointer fields.

A *heap state* is represented by a triple from the product

$$\begin{aligned} HEAPTYPE = & (POINTER \times FIELD_T \rightarrow T) \times \\ & (POINTER \times FIELD_P \rightarrow POINTER) \\ & \times \mathbb{P}(POINTER) \end{aligned}$$

For any $h \in HEAPTYPE$, define the access names $eval_T$, $eval_P$ and $free$, described below:

- $h.eval_T = \pi_1(h) \in POINTER \times FIELD_T \rightarrow T$. An application of this function, $eval_T(pt, f)$, returns the value contained in the field *f* of the record pointed to by *pt*.
- $h.eval_P = \pi_2(h) \in POINTER \times FIELD_P \rightarrow POINTER$. An application of this function, $eval_P(pt, f)$, returns the pointer value contained in the field *f* of the record pointed to by *pt*.

- $h.free = \pi_3(h) \subseteq POINTER$. $free$ represents the set of unallocated pointers. As noted in Section 4.2, the heap should never run out of space, so we insist that $free$ (like $POINTER$) be infinite.

We are only interested in heap states which have infinite $free$ sets, so we define the set of all possible heap states to be:

$$HEAP = \{h \in HEAPTYPE \mid h.free \text{ is infinite}\}$$

We need to be able to represent four operations on the heap: read, write, DCAS and allocation using the `new` operator. The read operation is modelled using applications of the $eval$ functions. No attempt is made to model the recycling of memory by garbage collection: all the garbage collector does is make allocation successful more often, it does not affect the correctness of the program. Also, the address-of operation is not modelled directly: it is easy to represent this using an extra level of indirection. The Snark automaton will make use of two pointer constants $\&LeftHat : POINTER$ and $\&RightHat : POINTER$ (strictly, each of these is an identifier, rather the operator $\&$ applied to an identifier) which will represent the *locations* where the current values of `LeftHat` and `RightHat` are stored. The value of `LeftHat` is accessed with the expression $eval_P(h)(\&LeftHat, Z)$.

The write operation sets the value at a field of a given record (referenced by some pointer). This operation is represented by two $update$ functions: the first, $update_T : HEAP \times POINTER \times FIELD_T \times T \rightarrow HEAP$ writes a value from T into a field; the second, $update_P : HEAP \times POINTER \times FIELD_P \times POINTER \rightarrow HEAP$, writes a pointer value. The first argument to each function is the heap state before the write; the returned element of $HEAP$ represents the state after the write. For any $h \in HEAP, pt, pt' \in POINTER, f \in FIELD, t \in T$ these functions are defined thus:

$$update_T(h, pt, f, t) = (h.eval_T[(pt, f) := t], h.eval_P, h.free)$$

$$update_P(h, pt, f, pt') = (h.eval_T, h.eval_P[(pt, f) := pt'], h.free)$$

Recall from Section 2.2 that there is an optional line of code in the pop routines: the `V` field of a node popped from a deque is set to `null` if the deque contains reference values, to allow garbage collection to work more effectively. In our heap model, the deque would contain reference values just when T is instantiated with the set $POINTER$, in which case it would be meaningful to include this optional line: we do not do so in this verification.

The DCAS operation as used in the Snark algorithm takes two addresses and four values as arguments (see Figure 2.2). However, the DCAS defined here uses a pointer and a field to represent an address. Thus, DCAS has the following type:

$$\begin{aligned} DCAS : HEAP \times POINTER \times FIELD_p \times POINTER \times FIELD_p \\ \times POINTER \times POINTER \times POINTER \times POINTER \\ \rightarrow HEAP \times bool \end{aligned}$$

The second and third arguments, together with the fourth and fifth arguments, specify the locations to be operated on. The remaining arguments are the expected values and the new values to write to the locations if the DCAS is successful. Similarly to the *update* functions, the $HEAP$ argument and the $HEAP$ part of the return value represent the states of the heap before and after the DCAS. The *bool* member of the return value signals whether the DCAS was successful. DCAS is defined using the *update* functions:

$$DCAS(h, pt_1, f_1, pt_2, f_2, old_1, old_2, new_1, new_2) = \begin{cases} (update(update(h, pt_1, f_1, new_1), & \text{if } eval(h)(pt_1, f_1) = old_1 \\ & pt_2, f_2, new_2), true) & \wedge eval(h)(pt_2, f_2) = old_2 \\ (h, false) & \text{otherwise} \end{cases}$$

Note that the Snark algorithm only uses DCAS on pointer fields, so the $DCAS$ function defined above is sufficient for our purposes.

We also need to model a new operator. This operator is represented by a function $new : HEAP \rightarrow HEAP \times POINTER$. Its argument is the heap before the allocation. The first component of its return value is the heap after the allocation, which is like the first heap, but with a pointer removed from the *free* set. This

removed pointer is the second component of the returned value, so the pointer returned by *new* was free before the allocation. The *new* operator can be any function that fulfils the following condition:

$$\begin{aligned} \forall h \in \text{HEAP} \bullet \\ \text{free}(h) \neq \emptyset \Rightarrow \exists pt \in \text{free}(h) \bullet \text{new}(h) = \\ ((h.\text{eval}_T, h.\text{eval}_P, h.\text{free} \setminus pt), pt) \end{aligned}$$

Since *h.free* is never empty for any $h \in \text{HEAP}$, this condition guarantees that *new* will always return a pointer out of the *free* set.³

The Snark algorithm uses two global constants: `null` and `Dummy`. `null` is used for two purposes, neither of which is needed in this model of Snark: to signal that the heap is full; and to eliminate references to objects that have been removed from the deque. `Dummy` will be represented by the constant value $Dummy \in \text{POINTER}$. As mentioned above, we use the constants $\&LeftHat$ and $\&RightHat$ to represent `&LeftHat` and `&RightHat`. Because we want to update these locations separately, we stipulate that

$$\&LeftHat \neq \&RightHat$$

4.2.3 Notation for the Heap Model

This subsection introduces some notation for describing heap states that is closely related to the notation used in C-style pseudo-code. The goal is to replace expressions like $h.\text{eval}_P(h.\text{eval}_P(pt, f_1), f_2)$ with something like the more familiar $pt \rightarrow f_1 \rightarrow f_2$. The notation is governed by the following grammar:

$$\begin{aligned} \text{EXPR} &:= \text{VAL TAIL} \\ \text{TAIL} &:= \epsilon \mid \xrightarrow{h} f \text{ TAIL} \end{aligned}$$

Here, ϵ is the empty string; **VAL** is some pointer value (a the value of a variable of the Snark automaton in some state, or a constant like $\&LeftHat$ or *Dummy*); *h*

³Note that the set *HEAP* is closed under all the operations described above: in particular, if $h \in \text{HEAP}$ then $\pi_1(\text{new}(h)) \in \text{HEAP}$.

is a value from *HEAP*; and f is an element of *FIELD*. The interpretation of these expressions is straightforward:

- An expression of the form *VAL* evaluates to that pointer value.
- An expression of the form $\text{EXPR} \xrightarrow{h} \text{F TAIL}$ evaluates to $h.\text{eval}_T(pt, F)$ (if $f \in \text{FIELD}_T$) or $h.\text{eval}_P(pt, F)$ (if $f \in \text{FIELD}_P$), where pt is the evaluation of *EXPR*.

When the heap in question is obvious from the context, it will be omitted: in that case, the expressions $h.\text{eval}_P(pt, f)$, $pt \xrightarrow{h} f$ and $pt \rightarrow f$ would all be equivalent.

4.2.4 The States of the Snark Automaton

Now that the heap model has been described, we are in a position to define the states of the Snark automaton. Like *DeqAut*, *SnarkAut* has a *pc* and *side* variable associated with each process; it also has variables associated with each process, that correspond to the local variables used in the Snark algorithm described in Chapter 2. The global state of *SnarkAut* is just a variable representing the heap.

The states of *SnarkAut* are as follows:

$$\begin{aligned} \text{states}(\text{SnarkAut}) = & \text{HEAP} \times \\ & \left(\prod_{p \in \text{PROC}} \text{PCOUNT} \right) \times \left(\prod_{p \in \text{PROC}} \text{SIDE} \right) \times \\ & \left(\prod_{p \in \text{PROC}} T^2 \right) \times \left(\prod_{p \in \text{PROC}} \text{POINTER}^5 \right) \end{aligned}$$

SnarkAut has the following state variables: $h \in \text{HEAP}$ is the heap; $side_p \in \text{SIDE}$ and $pc_p \in \text{PCOUNT}$ for each $p \in \text{PROC}$ represent the side of the operation each active process is performing and the program counter of each process. *SnarkAut* also has several other variables associated with each process: these variables and their relationship with the Snark algorithm as presented in Chapter 2 are presented in Figure 4.3. As usual the p indexes range over *PROC*. The specific association

Variable name	p executing op. on left	p executing op. on right
$hat_p \in POINTER$	lh of both operations	rh of both ops
$otherHat_p \in POINTER$	rh of both operations	lh of both ops
$hatIn_p \in POINTER$	lhR of pop operations	rhL of pop ops
$hatOut_p \in POINTER$	lhL of push operations	rhR of push ops
$val_p \in VAL$	v of push operations	
$nd_p \in POINTER$	nd of push operations	
$result_p \in VAL$	result of pop operations	

Figure 4.3: Local variables of the Snark automaton

between variables and components of elements of $states(SnarkAut)$ is not important, since we never construct new states of $SnarkAut$ using tuple construction.

The set $PCOUNT$ is defined as follows:

$$\begin{aligned}
 PCOUNT = & \\
 & \{idle, push_resp, resp_empty, \\
 & pc_pop_3, pc_pop_4, pc_pop_5, pc_pop_6, pc_pop_8, \\
 & pc_pop_9, pc_pop_12, pc_pop_13, pc_pop_14, pc_pop_15, \\
 & pc_push_2, pc_push_3, pc_push_4, pc_push_5, pc_push_7, \\
 & pc_push_8, pc_push_9, pc_push_10, pc_push_11, pc_push_12, \\
 & pc_push_15, pc_push_16\} \cup \\
 & \bigcup_{t \in T} pop_resp(t)
 \end{aligned}$$

The program counter values $idle$, $push_resp$ and $pop_resp(t)$ perform the same function as in $DeqAut$: if $pc_p = idle$, p is not executing an operation; if $pc_p = push_resp$, p is about to return from a push operation; if $pc_p = pop_resp(t)$, p is about to return from a pop operation with the value t ; if $pc_p = resp_empty$, p is about to return from a pop operation, signalling empty.

There is a close relationship between the rest of the members of $PCOUNT$ and the internal actions: for example, an action of the form pop_n_p for some natural number n will have as part of its precondition that $pc_p = pc_pop_n$; an action of the form $push_n_yes_p$ will have as part of its precondition that $pc_p = pc_push_n$.

After initialisation of the Snark algorithm, both `LeftHat` and `RightHat` point to `Dummy` (as per Figure 2.6): the start states of *SnarkAut* are those modelling this fact, constrained so that every process is *idle*:

$$\begin{aligned} \text{start}(\text{SnarkAut}) = \{ & s \in \text{states}(\text{SnarkAut}) \mid \\ & \& \text{LeftHat} \xrightarrow{s.h} Z = \text{Dummy} \wedge \\ & \& \text{RightHat} \xrightarrow{s.h} Z = \text{Dummy} \wedge \\ & \text{Dummy} \notin s.\text{free} \wedge \\ & \forall p \in \text{PROC} \bullet s.pc_p = \text{idle} \} \end{aligned}$$

4.2.5 The Transition Relation of the Snark Automaton

The transition relation of *SnarkAut* is defined in Figure 4.4 on page 73 and Figure 4.5 on page 74. The approach is to translate each line of code to a transition (or more precisely, a *set* of transitions) of the automaton. This strategy allows us to carry the atomicity assumptions made in the Snark algorithm over to the automaton representing that algorithm.

The transition relation of *SnarkAut* makes use of several auxiliary functions. These functions are needed to allow the transition relation to exploit the symmetry in the Snark algorithm. Take for example, line 7 of the push routines: for `pushRight` it is `rh = RightHat`; for `pushLeft`, `lh = LeftHat`. In other words, the process loads the value of the hat corresponding to the side it is pushing on. To represent this in the automaton where the action labels suppress the side of the operation, we use the following function as a shorthand:

$$\text{onHat}(cc, p) = \begin{cases} \& \text{LeftHat} \xrightarrow{cc.h} Z & \text{if } cc.\text{side}_p = \text{Left} \\ \& \text{RightHat} \xrightarrow{cc.h} Z & \text{if } cc.\text{side}_p = \text{Right} \end{cases}$$

Similarly, to obtain the hat corresponding to the *opposite* side, we use:

$$\text{offHat}(cc, p) = \begin{cases} \& \text{RightHat} \xrightarrow{cc.h} Z & \text{if } cc.\text{side}_p = \text{Left} \\ \& \text{LeftHat} \xrightarrow{cc.h} Z & \text{if } cc.\text{side}_p = \text{Right} \end{cases}$$

We also need functions that obtain the fields L or R , depending on the side of the operation:

$$\text{onField}(cc, p) = \begin{cases} L & \text{if } cc.\text{side}_p = \text{Left} \\ R & \text{if } cc.\text{side}_p = \text{Right} \end{cases}$$

$$\text{offField}(cc, p) = \begin{cases} R & \text{if } cc.\text{side}_p = \text{Left} \\ L & \text{if } cc.\text{side}_p = \text{Right} \end{cases}$$

The transition relation also uses some functions that package the effect of each DCAS. These functions return the updated heap and a boolean indicating whether or not the DCAS was successful. They are as follows:

$$\begin{aligned} \text{push_12_dcas}(cc, p) = \\ & \text{DCAS}(cc.h, \text{onHat}(cc, p), Z, \text{offHat}(cc, p), Z, \\ & \quad cc.\text{hat}_p, cc.\text{otherHat}_p, cc.nd_p, cc.nd_p) \end{aligned}$$

$$\begin{aligned} \text{push_16_dcas}(cc, p) = \\ & \text{DCAS}(cc.h, \text{onHat}(cc, p), Z, cc.\text{hat}_p, \text{onField}(cc, p), \\ & \quad cc.\text{hat}_p, cc.\text{hatOut}_p, cc.nd_p, cc.nd_p) \end{aligned}$$

$$\begin{aligned} \text{pop_6_dcas}(cc, p) = \\ & \text{DCAS}(cc.h, \text{onHat}(cc, p), Z, cc.\text{hat}_p, \text{onField}(cc, p), \\ & \quad cc.\text{hat}_p, cc.\text{hat}_p, cc.\text{hat}_p, cc.\text{hat}_p) \end{aligned}$$

$$\begin{aligned} \text{pop_9_dcas}(cc, p) = \\ & \text{DCAS}(cc.h, \text{onHat}(cc, p), Z, \text{offHat}(cc, p), Z, \\ & \quad cc.\text{hat}_p, cc.\text{otherHat}_p, \text{Dummy}, \text{Dummy}) \end{aligned}$$

$$\begin{aligned} \text{pop_13_dcas}(cc, p) = \\ & \text{DCAS}(cc.h, \text{onHat}(cc, p), Z, cc.\text{hat}_p, \text{offField}(cc, p), \\ & \quad cc.\text{hat}_p, cc.\text{hatIn}_p, cc.\text{hatIn}_p, cc.\text{hat}_p) \end{aligned}$$

Note that all the functions described in this section take as an argument a state of *SnarkAut*. To invoke these functions from within the transition relation definition, we need to extend the transition relation notation presented in Section 3.2 by allowing it to directly mention the pre-state of the transition. To do this we use the symbol s .

Using the symmetric transition relation with the $side_p$ variables increases the complexity of stating the transition definition but halves the number of actions needed to define *SnarkAut*. This is very useful when verifying invariants or simulation relations. Recall that an invariant proof is made up of a series of sub-proofs, each one showing that the invariant property is preserved across the transitions labelled by each action. Halving the number of labels halves the number of sub-proofs. Simulation proofs have a closely related property, whereby the simulation relation is shown to be preserved across each action.

Given the heap model described in Section 4.2.2, the translation of the code of the Snark algorithm into the transitions of the Snark automaton is reasonably straightforward: loads from memory into local variables are translated into assignments to the process-indexed variables using applications of the *eval* functions (hidden behind the pointer-to-member notation); writes to memory are translated into applications of the *update* functions and assignments to the heap variable h . As mentioned in Section 4.2.1, the conditional statements are translated into pairs of (sets of) actions. For example, line 9 of the push routines (`if (lhl == lh) { . . . in pushLeft }`) is translated into $push_9_yes_p$ and $push_9_no_p$ for each p . A precondition of $push_9_yes_p$ is that $hatOut_p = hat_p$ and of $push_9_no_p$ that $hatOut_p \neq hat_p$. The effect of each of these transitions is to set the program counter of the process taking the transition to the program counter value corresponding to the appropriate branch of the conditional.

The `while` instructions are not directly translated. In the Snark algorithm, control loops back through the head of a while if one of the DCAS operations fail, so the effect of actions that model this failure is to set the program counter of the process taking the transition to the value corresponding to the top of the loop. For example, the effect of $push_16_no_p$ is to set pc_p to pc_push_7 .

<p><i>push_left_p</i>(<i>t</i>) :</p> <p>pre $pc_p = idle$</p> <p>eff $pc_p := pc_push_2;$ $side_p := Left;$ $val_p := t$</p>	<p><i>push_right_p</i>(<i>t</i>) :</p> <p>pre $pc_p = idle$</p> <p>eff $pc_p := pc_push_2;$ $side_p := Right;$ $val_p := t$</p>
<p><i>push_2_p</i> :</p> <p>pre $pc_p = pc_push_2$</p> <p>eff $(h, nd_p) := new(h);$ $pc_p := pc_push_4$</p>	<p><i>push_4_p</i> :</p> <p>pre $pc_p = pc_push_4$</p> <p>eff $h := update_p$ $(h, nd_p, onField(s, p), Dummy);$ $pc_p := pc_push_5$</p>
<p><i>push_5_p</i> :</p> <p>pre $pc_p = pc_push_5$</p> <p>eff $h :=$ $update_T(h, nd_p, V, val_p);$ $pc_p := pc_push_7$</p>	<p><i>push_7_p</i> :</p> <p>pre $pc_p = pc_push_7$</p> <p>eff $hat_p := onHat(s, p) \rightarrow Z;$ $pc_p := pc_push_8$</p>
<p><i>push_8_p</i> :</p> <p>pre $pc_p = pc_push_8$</p> <p>eff $hatOut_p :=$ $hat_p \rightarrow onField(s, p);$ $pc_p := pc_push_9$</p>	<p><i>push_9_yes_p</i> :</p> <p>pre $pc_p = pc_push_9 \wedge$ $hatOut_p = hat_p$</p> <p>eff $pc_p := pc_push_10$</p>
<p><i>push_9_no_p</i> :</p> <p>pre $pc_p = pc_push_9 \wedge$ $hatOut \neq hat_p$</p> <p>eff $pc_p := pc_push_15$</p>	<p><i>push_10_p</i> :</p> <p>pre $pc_p = pc_push_10$</p> <p>eff $h :=$ $update_p(h, nd_p, offField(s, p), Dummy);$ $pc_p := pc_push_11$</p>
<p><i>push_11_p</i> :</p> <p>pre $pc_p = pc_push_11$</p> <p>eff $otherHat_p := offHat(s, p);$ $pc_p := pc_push_12$</p>	<p><i>push_12_yes_p</i> :</p> <p>pre $pc_p = pc_push_12 \wedge$ $\pi_2(push_12_dcas(s, p))$</p> <p>eff $h := \pi_1(push_12_dcas(s, p));$ $pc_p := push_resp$</p>
<p><i>push_12_no_p</i> :</p> <p>pre $pc_p = pc_push_12 \wedge$ $\neg \pi_2(push_12_dcas(s, p))$</p> <p>eff $pc_p := pc_push_7$</p>	<p><i>push_15_p</i> :</p> <p>pre $pc_p = pc_push_15$</p> <p>eff $h := update_p(h, nd_p, offField(s, p), hat_p);$ $pc_p := pc_push_16$</p>
<p><i>push_16_yes_p</i> :</p> <p>pre $pc_p = pc_push_16 \wedge$ $\pi_2(push_16_dcas(s, p))$</p> <p>eff $h := \pi_1(push_16_dcas(s, p));$ $pc_p := push_resp$</p>	<p><i>push_16_no_p</i> :</p> <p>pre $pc_p = pc_push_16 \wedge$ $\neg \pi_2(push_16_dcas(s, p))$</p> <p>eff $pc_p := pc_push_7$</p>
<p><i>push_resp_p</i> :</p> <p>pre $pc_p = push_resp$</p> <p>eff $pc_p := idle$</p>	

Figure 4.4: Push transitions of the Snark automaton.

<p><i>pop_left_p</i> :</p> <p>pre $pc_p = idle$</p> <p>eff $pc_p := pc_pop_3;$ $side_p := Left$</p>	<p><i>pop_right_p</i> :</p> <p>pre $pc_p = idle$</p> <p>eff $pc_p := pc_pop_3;$ $side_p := Right$</p>
<p><i>pop_3_p</i> :</p> <p>pre $pc_p = pc_pop_3$</p> <p>eff $hat_p := onHat(s, p);$ $pc_p := pc_pop_4$</p>	<p><i>pop_4_p</i> :</p> <p>pre $pc_p = pc_pop_4$</p> <p>eff $otherHat_p := offHat(s, p);$ $pc_p := pc_pop_5$</p>
<p><i>pop_5_yes_p</i> :</p> <p>pre $pc_p = pc_pop_5 \wedge$ $hat_p \rightarrow onField(s, p) = hat_p$</p> <p>eff $pc_p := pc_pop_6$</p>	<p><i>pop_5_no_p</i> :</p> <p>pre $pc_p = pc_pop_5 \wedge$ $hat_p \rightarrow onField(s, p) \neq hat_p$</p> <p>eff $pc_p := pc_pop_8$</p>
<p><i>pop_6_yes_p</i> :</p> <p>pre $pc_p = pc_pop_6 \wedge$ $\pi_2(pop_6_dcas(s, p))$</p> <p>eff $pc_p := resp_empty$</p>	<p><i>pop_6_no_p</i> :</p> <p>pre $pc_p = pc_pop_6 \wedge$ $\neg \pi_2(pop_6_dcas(s, p))$</p> <p>eff $pc_p := pc_pop_3$</p>
<p><i>pop_8_yes_p</i> :</p> <p>pre $pc_p = pc_pop_8 \wedge$ $hat_p = otherHat_p$</p> <p>eff $pc_p := pc_pop_9$</p>	<p><i>pop_8_no_p</i> :</p> <p>pre $pc_p = pc_pop_8 \wedge$ $hat_p \neq otherHat_p$</p> <p>eff $pc_p := pc_pop_12$</p>
<p><i>pop_9_yes_p</i> :</p> <p>pre $pc_p = pc_pop_9 \wedge$ $\pi_2(pop_9_dcas(s, p))$</p> <p>eff $h := \pi_1(pop_9_dcas(s, p));$ $pc_p := pop_resp(hat_p \rightarrow V)$</p>	<p><i>pop_9_no_p</i> :</p> <p>pre $pc_p = pc_pop_9 \wedge$ $\neg \pi_2(pop_9_dcas(s, p))$</p> <p>eff $pc_p := pc_pop_3$</p>
<p><i>pop_12_p</i> :</p> <p>pre $pc_p = pc_pop_12$</p> <p>eff $hatIn_p :=$ $hat_p \rightarrow offField(s, p);$ $pc_p := pc_pop_13$</p>	<p><i>pop_13_yes_p</i> :</p> <p>pre $pc_p = pc_pop_13 \wedge$ $\pi_2(pop_13_dcas(s, p))$</p> <p>eff $h := \pi_1(pop_13_dcas(s, p));$ $pc_p := pc_pop_14$</p>
<p><i>pop_13_no_p</i> :</p> <p>pre $pc_p = pc_pop_13 \wedge$ $\neg \pi_2(pop_13_dcas(s, p))$</p> <p>eff $pc_p := pc_pop_3$</p>	<p><i>pop_14_p</i> :</p> <p>pre $pc_p = pc_pop_14$</p> <p>eff $result_p := hat_p \rightarrow V;$ $pc_p := pc_pop_15$</p>
<p><i>pop_15_p</i> :</p> <p>pre $pc_p = pc_pop_15$</p> <p>eff $h := update_p$ $(h, hat_p, onField(s, p), Dummy);$ $pc_p := pop_resp(result_p)$</p>	<p><i>pop_resp_p(t)</i> :</p> <p>pre $pc_p = pop_resp(t)$</p> <p>eff $pc_p := idle$</p>
<p><i>resp_empty_p</i> :</p> <p>pre $pc_p = resp_empty$</p> <p>eff $pc_p := idle$</p>	

Figure 4.5: Pop transitions of the Snark automaton.

Chapter 5

The Attempted Verification

This chapter presents a relation over $states(DeqAut)$ and $states(SnarkAut)$, which was designed to be a simulation relation in the sense of Definition 3.16 and used in the attempted verification of the Snark algorithm. The proposed relation is *not* a simulation between the concrete and abstract automata. In attempting to prove that this relation was a simulation, a bug was discovered in the Snark algorithm. The discussion of the proposed simulation in this chapter serves several purposes: it completes the account of the verification methodology used in this thesis; the issues faced in the development of this relation should be applicable to the development of simulation relations for the verification of other non-blocking algorithms using dynamic memory; and an analysis of the failure of the relation as a simulation lead directly to the discovery of a bug in an algorithm that was otherwise believed to be correct. The bug in the Snark algorithm and its relationship with the proposed simulation relation are described in Chapter 6.

Some invariants of $SnarkAut$ were stated and proved during the attempted verification: these invariants are presented and discussed in Section 5.6.

Obviously, this chapter cannot provide a *proof* that the relation is a simulation (it is not). However, two important lemmas about the relation are stated and detailed arguments provided in Section 5.7. These lemmas show that part of the simulation relation (asserting that the global state of the Snark automaton represents a deque data-structure) is preserved across critical transitions (those which

represent DCASs applied to this global data-structure). Providing these arguments serves two purposes: working through the proofs will help develop a good intuition about why the relation was constructed as it was; and the possibility of proving these lemmas provides evidence that the approach taken to building the relation was a reasonable one.

The reader will observe that the arguments for the preservation of even one part of the simulation relation are very detailed. This level of detail and the possibility of human error that it entails motivate the use of mechanical proof assistance. As described in the introduction, the attempted verification of the Snark algorithm was undertaken using the mechanical theorem prover PVS. The notion of simulation relation defined in Chapter 3, the automata *SnarkAut* and *DeqAut* described in Chapter 4 and the specific relation described here have all been encoded in the PVS language. Moreover, machine-checkable proofs for most of the proof obligations generated by the proposed simulation relation have been constructed using the PVS prover. This provides further evidence that the relation described in this chapter would have been a simulation between *SnarkAut* and *DeqAut* had the Snark algorithm been correct. The use of PVS is briefly described in Section 5.8.

To avoid confusion by over-use of the term *deque*, for the remainder of this chapter the deque data-structure of the Snark algorithm (that part of the heap accessible through `LeftHat` or `RightHat`) will be referred to as the ‘concrete data-structure’ or ‘the data-structure’ of a state of *SnarkAut*; the abstract variable *deq* (a member of the deque datatype) will be referred to as the ‘abstract data-structure’.

For the remainder of the chapter, occurrences of the following variables should be assumed to range over the stated sets: *cc* ranges over $states(SnarkAut)$; *ab* over $states(DeqAut)$; *p* over *PROC*; *f* over functions from the set $\mathbb{Z} \rightarrow POINTER$; and *a, a'* over $acts(SnarkAut) \cup acts(DeqAut)$. In definitions of predicates and functions, it is assumed that these variables are universally quantified.

The expression *concrete state* describes states of *SnarkAut*; similarly, *abstract state* describes states of *DeqAut*. When we refer to *related states*, we mean any

concrete state and abstract state that are related by the proposed simulation relation.

5.1 Step Correspondence

Before describing the simulation relation in detail, it is useful to describe how a proof that a relation is a simulation is structured. This structure follows directly from the definition of simulation relation given in Definition 3.16. This definition is presented here, instantiated with the names of the relevant automata for convenience.

A forward simulation R from $SnarkAut$ to $DeqAut$ is a relation over $states(SnarkAut)$ and $states(DeqAut)$ satisfying:

1. For all $cc \in start(SnarkAut)$, there is some $ab \in start(DeqAut)$ such that $R(cc, ab)$.
2. For all $cc, cc' \in reach(SnarkAut)$, if $cc \xrightarrow{a} cc'$ and $a \in external(SnarkAut)$, then for all $ab \in reach(DeqAut)$ such that $R(cc, ab)$, there is some $ab' \in states(DeqAut)$ such that $R(cc', ab')$ and $ab \xrightarrow{a} ab'$.
3. For all $cc, cc' \in reach(SnarkAut)$, if $cc \xrightarrow{a} cc'$ and $a \notin external(SnarkAut)$, then for all $ab \in reach(DeqAut)$ such that $R(cc, ab)$, either
 - (a) there is some $ab' \in states(DeqAut)$ and action $b \in internal(DeqAut)$, such that $ab \xrightarrow{b} ab'$ and $R(cc', ab')$, or
 - (b) $R(cc', ab)$.

Given a relation between state of $SnarkAut$ and $DeqAut$, if (2) or (3) is true for some concrete action a and all the pre- and post- states of transitions labelled by a , we say that the relation is *preserved across a* .

For each $ab \in reach(DeqAut)$ and $a' \in acts(DeqAut)$ there is at most one ab' such that $ab \xrightarrow{a'} ab'$. This removes any freedom we have in choosing an abstract

post-state for clauses (2) or (3a): once we have the abstract action (whether it is external and the choice was forced on us, or internal and we had some freedom) we have to show that the unique abstract post-state is related to the concrete post-state. Because of this, we tend to focus on *matching* concrete actions with abstract actions without mentioning that we are also matching concrete states with abstract states.

Collectively, the decisions for all the concrete internal actions about whether (3a) or (3b) is true and the matchings used to witness (3a) are called the *step correspondence* [34].

The choice of a step correspondence can be used to help develop (and motivate) a simulation relation. The step correspondence used in the present verification attempt is as follows:

- For transitions of *SnarkAut* that model a successful DCAS operation of a push procedure (ie., those labelled by $push_{12_yes_p}$ or $push_{16_yes_p}$ for some p) we choose option (3a) above: these transitions are matched with the transition of *DeqAut* labelled by $do_push_p(cc.val_p)$. The successful DCAS transitions represent a globally visible update that changes the sequence which the concrete data-structure represents. This change in representation can be reflected in the abstract automaton by making it take the appropriate do_push operation.
- Transitions that model a successful DCAS operation of the `pop` procedure (ie., those labelled by $pop_{6_yes_p}$, $pop_{9_yes_p}$ or $pop_{13_yes_p}$ for some p) are matched with the transition of *DeqAut* labelled by do_pop_p . Apart from those labelled by $pop_{6_yes_p}$, these transitions represent a globally visible update in the same way as successful push-DCAS transitions. Transitions labelled by $pop_{6_yes_p}$ represent p determining that the concrete data-structure is empty: this test is done directly in the abstract transition labelled by do_pop_p .
- For the transitions labelled by other internal actions we choose option (3b) above: these actions do not change the abstract state which is represented

by the concrete automaton.

Note that all transitions labelled by some $a \in \text{external}(\text{SnarkAut})$ are associated with the abstract transition labelled by a : this association is required by clause (2) of the definition of simulation relation.

5.2 Overview of the Simulation Relation

The proposed simulation relation presented here is essentially a conjunction of three major predicates. Each of these predicates is designed to achieve a particular high-level goal, and is itself a conjunction. The predicates are listed and briefly described below:¹

- *correspondence_ok?*: This predicate guarantees that, for related states of *SnarkAut* and *DeqAut*, each process is attempting to ‘do the same thing’ in both concrete and abstract states. For example, it guarantees that if a process is attempting to push a certain value onto the right of the deque in the concrete automaton, then in any related abstract state, that process is attempting to push the same value onto the same side of the abstract deque. It also guarantees that each process will be at analogous stages of the computation in each state. For example, if a process has just successfully completed a DCAS as part of the push sequence in the concrete state, then in any related abstract state, that process has just completed the *do_push* transition. The *correspondence_ok?* predicate is critical in enforcing the step correspondence described in Section 5.1.
- *obj_ok?* (a contraction of “object ok”): This predicate ensures that the concrete data-structure represents the same sequence of values as the abstract data-structure: for instance, it asserts that, for related states, the concrete data-structure is empty exactly when the abstract data-structure is. It also

¹The predicate names in this chapter follow the convention of many users of PVS: predicate names always end in ‘?’. In PVS this helps to distinguish predicates (functions into *bool*) from other types of functions.

contains information about the internal structure of the concrete deque: for example, it asserts that if the data-structure is not empty, then the node to the left of the node pointed to by $\&LeftHat$ is right-dead.

- $rest_ok?$: As its name suggests, the $rest_ok?$ predicate makes a variety of assertions, but they are all bound together by two features. Firstly, $rest_ok?$ is in a sense auxiliary to $obj_ok?$ and $correspondence_ok?$ in that these predicates force the simulation relation to have certain other properties: these properties are provided by $rest_ok?$. Secondly, the assertions contained in $rest_ok?$ are not ‘about’ the relationship between two automata, they are about the internal state of the concrete automaton.

The top-level predicate in the proposed simulation relation is $SR?$ (“Simulation Relation”) and is defined as follows:

$$SR?(ab, cc) \stackrel{def}{=} \exists f : \mathbb{Z} \rightarrow \text{POINTER} \bullet rel?(ab, cc, f)$$

The existentially quantified function f , called the *representation function* in this thesis, is used in the $obj_ok?$ predicate to relate the abstract data-structure to the concrete data-structure.

The predicate $rel?$ is as follows:

$$rel?(ab, cc, f) \stackrel{def}{=} injective_in_range?(f, ab.deq.bot, ab.deq.top) \wedge \\ correspondence_ok?(ab, cc) \wedge obj_ok?(ab, cc, f) \wedge rest_ok?(ab, cc, f)$$

$rel?$ is a notational convenience, allowing us to assert lemmas which make claims about the function f : to do this, we need to universally quantify over functions of that type, but the existential quantification in $SR?$ hides the function.

Because $SR?$ is the existential quantification of a function, we have to decide what function we will choose to witness the truth of $SR?$, after each transition of $SnarkAut$, given that a function existed fulfilling the requirements of $SR?$, prior to the transition. For transitions labelled by every concrete action except those with labels of the form $do_push_p(t)$, the function witnessing $SR?$ in the concrete and

abstract post-states will be the *same* as that witnessing $SR?$ for the pre-states. The reason for this is described in Section 5.4.

The predicate $injective_in_range?$ asserts that f is a 1-1 correspondence from integers between $ab.deq.bot$ and $ab.deq.top$ (not inclusive) to a subset of $POINTER$. This constraint on the representation function is discussed in Section 5.4.

5.3 The $correspondence_ok?$ Predicate

Figure 5.1 defines the $correspondence_ok?$ predicate; Figure 5.2 defines certain predicates used in the definition of $correspondence_ok?$ and elsewhere. Note that the first two entries in this figure, $pc_pop_i_j?$ and $pc_push_i_j?$, define *families* of predicates. Figure 5.2 and Figure 5.4, presented on page 87, define basic predicates over processes and pointers that are used throughout the proposed simulation relation.

The $correspondence_ok?$ predicate has two overlapping goals. The first is to ensure that each process is attempting the same operation in related states of $SnarkAut$ and $DeqAut$. The conjunct (9), for example, helps fulfil the first of these goals: it guarantees that, for related states, each process is operating on the same side. One aspect of what the other conjuncts do is to ensure that if a process is about to execute a transition modelling, say, part of a push operation, then that process is also about to execute part of the abstract push operation.

The second goal of $correspondence_ok?$ is to ensure that whenever $SnarkAut$ takes a transition, $DeqAut$ is able to take the appropriate transition, if it exists. For each external action, we need to know that, for related concrete and abstract states, whenever the concrete precondition of the action holds, so does the abstract precondition. Moreover, for each concrete internal action, we need to know that whenever the concrete precondition of that action holds, so does the abstract precondition of the matching action (if it exists, as determined by the step correspondence).

The conjuncts marked (1), (3), (5), and (8) guarantee collectively that whenever the concrete precondition of each external action is met, the abstract precon-

$$\begin{aligned}
\text{correspondence_ok?}(ab, cc) &\stackrel{\text{def}}{=} \\
&\forall p \in \text{PROC}, t \in T \bullet \\
&\quad (cc.pc_p = \text{idle} \Rightarrow ab.pc_p = \text{idle}) \quad (1) \\
&\quad \wedge \\
&\quad (in_do_push?(cc, p) \Rightarrow ab.pc_p = do_push(cc.val_p)) \quad (2) \\
&\quad \wedge \\
&\quad (cc.pc_p = \text{push_resp} \Rightarrow ab.pc_p = \text{push_resp}) \quad (3) \\
&\quad \wedge \\
&\quad (in_do_pop?(cc, p) \Rightarrow ab.pc_p = do_pop) \quad (4) \\
&\quad \wedge \\
&\quad (cc.pc_p = \text{resp_empty} \Rightarrow ab.pc_p = \text{resp_empty}) \quad (5) \\
&\quad \wedge \\
&\quad (cc.pc_p = pc_pop_14 \Rightarrow \\
&\quad \quad ab.pc_p = pop_resp(cc.hat_p \xrightarrow{cc.h} V)) \quad (6) \\
&\quad \wedge \\
&\quad (cc.pc_p = pc_pop_15 \Rightarrow \\
&\quad \quad ab.pc_p = pop_resp(cc.result_p)) \quad (7) \\
&\quad \wedge \\
&\quad (cc.pc_p = pop_resp(t) \Rightarrow ab.pc_p = pop_resp(t)) \quad (8) \\
&\quad \wedge \\
&\quad cc.side_p = ab.side_p \quad (9)
\end{aligned}$$

Figure 5.1: The *correspondence_ok?* predicate.

$$pc_pop_i_j?(cc, p) \stackrel{def}{=} \\ cc.pc_p = pc_pop_n \\ \text{where } i \leq n \leq j$$

p is about to execute a transition modelling a line of code of the `pop` routines, numbered between i and j .

$$pc_push_i_j?(cc, p) \stackrel{def}{=} \\ cc.pc_p = pc_push_n \\ \text{where } i \leq n \leq j$$

p is about to execute a transition modelling a line of code of the `push` routines, numbered between i and j .

$$in_do_pop?(cc, p) \stackrel{def}{=} \\ pc_pop_3_6?(cc, p) \vee \\ pc_pop_8_9?(cc, p) \vee \\ pc_pop_12_13?(cc, p)$$

p is executing the transitions which allow it to decide which DCAS to execute, or about to execute a DCAS.

$$in_return_val?(cc, p) \stackrel{def}{=} \\ pc_pop_14_15?(cc, p)$$

p is doing the cleanup prior to returning a value.

$$in_do_push?(cc, p) \stackrel{def}{=} \\ pc_push_2_16?(cc, p)$$

p is about to execute a transition modelling a line of the `push` routines which is not a return instruction (ie. a line up to and including a DCAS).

Figure 5.2: Auxiliary predicates defining intervals of a process' execution.

dition of that action will be met.

Conjunct (2) illustrates an important point: the step correspondence often induces an association between a set of concrete program counter values (from the set $PCOUNT$) and a single abstract value (from $COUNTER'$), in cases where the concrete automaton takes several small steps to represent a single step of the abstract automaton. As an example, recall that the step correspondence matches concrete transitions labelled by actions of the form $push_left_p(t)$ with the abstract transition with the same label; transitions labelled by $push_16_yes_p$ are matched with the abstract transition labelled $do_push_p(val_p)$; and none of p 's intervening transitions are matched with a transition of the abstract automaton. Thus, in order to guarantee that the abstract precondition of $do_push_p(val_p)$ will be met when $cc.pc_p = pc_push_16$, $correspondence_ok?$ must record that whenever $cc.pc_p$ has a value indicating that $push_left_p(t)$ has been invoked but $push_16_yes_p$ has not yet been executed, then for related abstract automata, $ab.pc_p = do_push(cc.val_p)$ is true.

Conjunct (4) ensures that the abstract precondition of do_pop_p will be met whenever a transition with a label of the form $pop_13_yes_p$ is enabled. Collectively, conjuncts (6) to (8) do the same thing for concrete and abstract transitions labelled by external actions of the form pop_resp_p , but the situation is slightly more complex. Recall that we associate the successful DCAS transitions labelled by $pop_13_yes_p$ with do_pop_p transitions of $DeqAut$ and that $DeqAut$ sets $ab.pc_p$ to $pop_resp(t)$ for some t . However, after its DCAS transition, the process p has several steps to take before returning a value: in particular, p has not loaded $hat_p \rightarrow V$ into $result_p$. Our only choice is to construct $correspondence_ok?$ so that it records that once $cc.pc_p = pc_pop_14$, then in related states, the corresponding pop_resp action is already enabled; and so that it records that the result to be returned by that operation is stored in $hat_p \rightarrow V$. This requirement is met by conjunct (6). Conjunct (7) records that the value to be returned has now been loaded into $result_p$ which enables the primary requirement, that each process will 'return' the same value: that is, $cc.pc_p = pop_resp(t) \Rightarrow ab.pc_p = pop_resp(t)$.

5.4 The Representation Function and *obj_ok?*

The *obj_ok?* predicate is presented in Figure 5.3; the predicates *empty_dll_state?*, *right_dead?* and *left_dead?* are defined in Figure 5.4. Where *correspondence_ok?* makes claims about the *local* state of each process (its program counter and local variables) in each automaton, *obj_ok?* makes claims about the *global* state of related automata: for related *cc* and *ab*, the data-structure belonging to *cc* represents the abstract data-structure *ab.deq*. This representation requirement can be broken into two parts:

- For related *ab* and *cc*, the data-structure in *cc* must represent an empty sequence exactly if *ab.deq* is empty: this is the content of conjunct (1) of *obj_ok?*.
- When *ab.deq* is not empty, then the data-structure of any related *cc* must contain the same elements in the same order. The representation function which is the third argument to *obj_ok?* is critical in this regard, providing a way to say that the concrete data-structure contains the same elements in the same order as the abstract sequence. Conjuncts (2) and (3) achieve this goal.

Figure 5.5 illustrates the claims about related concrete and abstract states made by conjuncts (2) and (3) of *obj_ok?*. (2) says that *f* takes each index of the abstract sequence to a pointer whose field *V* contains the value stored at that index; moreover, the order of the indexes is mirrored by the values in the *L* and *R* fields of each record.

(3) says two things: the outermost used indexes of the abstract sequence are taken by *f* to the values stored in *&LeftHat* and *&RightHat*; and the sentinels beyond the hats have the appropriate self-pointers. (3a) guarantees that, for example, the value in the *V* field of *&LeftHat* $\rightarrow Z$ is the leftmost value in the abstract data-structure; (3c) and (3d) guarantee that the sentinels have the property used by processes to ascertain whether the deque is empty.

There are two important respects in which *correspondence_ok?* depends on *obj_ok?*. While conjunct (1) of *obj_ok?* can be motivated by the intuition that

$$\begin{aligned}
obj_ok?(ab, cc, f) &\stackrel{def}{=} \\
& (empty_dll_state?(cc) \Leftrightarrow \\
& \quad ab.deq.bot = ab.deq.top - 1) \tag{1} \\
& \wedge \\
& (\forall i \in \mathbb{Z} \bullet \\
& \quad ab.deq.bot < i \wedge i < ab.deq.top \Rightarrow \tag{2} \\
& \quad \quad f(i) \xrightarrow{cc.h} V = ab.deq.seq(i) \wedge \tag{2a} \\
& \quad \quad (i \neq ab.deq.bot + 1 \Rightarrow f(i) \xrightarrow{cc.h} L = f(i - 1)) \wedge \tag{2b} \\
& \quad \quad (i \neq ab.deq.top - 1 \Rightarrow f(i) \xrightarrow{cc.h} R = f(i + 1))) \tag{2c} \\
& \wedge \\
& (\neg empty_dll_state?(cc) \Rightarrow \tag{3} \\
& \quad f(ab.deq.bot + 1) = \&LeftHat \xrightarrow{cc.h} Z \wedge \tag{3a} \\
& \quad f(ab.deq.top - 1) = \&RightHat \xrightarrow{cc.h} Z \wedge \tag{3b} \\
& \quad right_dead?(cc.h, \&LeftHat \xrightarrow{cc.h} Z \xrightarrow{cc.h} L) \wedge \tag{3c} \\
& \quad left_dead?(cc.h, \&RightHat \xrightarrow{cc.h} Z \xrightarrow{cc.h} R)) \tag{3d}
\end{aligned}$$

Figure 5.3: The $obj_ok?$ predicate.

the concrete data-structure should represent the abstract data-structure, we also need it to ensure that the $correspondence_ok?$ predicate is preserved across certain transitions. Recall that the step correspondence matches concrete transitions having labels of the form $pop_6_yes_p$ (the transition where p determines that the concrete data-structure is empty) with the abstract transition labelled by do_pop_p . After the concrete transition, p is ready to respond that the deque is empty ($pc_p = resp_empty$); we need the same to be true of the abstract post-state of the transition labelled by do_pop_p , from pre-states related to the pre-state of the concrete transition. If the related abstract pre-state contained a non-empty deque, p would be ready to return a popped value after do_pop_p . The forward direction of conjunct (1) of $obj_ok?$ guarantees that this does not happen. In a similar way, the correspondence between $pop_13_yes_p$ (the successful DCAS transition for a deque with many elements) and do_pop_p requires that we know that if the concrete data-structure is *not* empty, then the abstract sequence is *not* empty. The backwards direction of (1) achieves this.

$right_dead?(h, pt) \stackrel{def}{=} pt \xrightarrow{h} R = pt$	True when the R field of the record pointed to by pt contains a self-pointer.
$left_dead?(h, pt) \stackrel{def}{=} pt \xrightarrow{h} L = pt$	True when the L field of the record pointed to by pt contains a self-pointer.
$empty_dll_state?(cc) \stackrel{def}{=} right_dead?(cc.h, \&LeftHat \xrightarrow{cc.h} Z) \wedge left_dead?(cc.h, \&RightHat \xrightarrow{cc.h} Z)$	True when the deque is empty.
$not_in_range?(f, pt, i_1, i_2) \stackrel{def}{=} \forall j \in \mathbb{Z} \bullet i_1 < j \wedge j < i_2 \Rightarrow f(j) \neq pt$	True when pt is not in the sub-range of f between $i_1, i_2 \in \mathbb{Z}$ not inclusive.
$injective_in_range?(f, i_1, i_2) \stackrel{def}{=} \forall j_1, j_2 \in \mathbb{Z} \bullet i_1 < j_1 \wedge j_1 < i_2 \wedge i_1 < j_2 \wedge j_2 < i_2 \wedge j_1 \neq j_2 \Rightarrow f(j_1) \neq f(j_2)$	True when f is a 1-1 mapping over the sub-range between $i_1, i_2 \in \mathbb{Z}$ not inclusive.

Figure 5.4: Auxiliary predicates defining properties of pointers in the simulation relation.

The second way in which $correspondence_ok?$ depends on $obj_ok?$ relates to the values which popping processes return. Recall that conjuncts (6) and (7) of $correspondence_ok?$ collectively ensure that each process will return the same value when they take pop_resp_p actions. In both the abstract and concrete automata, this return value is determined by the state of the automaton during an internal transition that occurs before the response transition in the execution of each popping process:

- In the transition relation of $DeqAut$, this return value is the value in the appropriate outermost used index of $deq.seq$ in the pre-state of the do_pop_p transition (for example, it is the value $deq.seq(deq.top - 1)$ if the process is popping from the right) leading up to the pop_resp_p action.

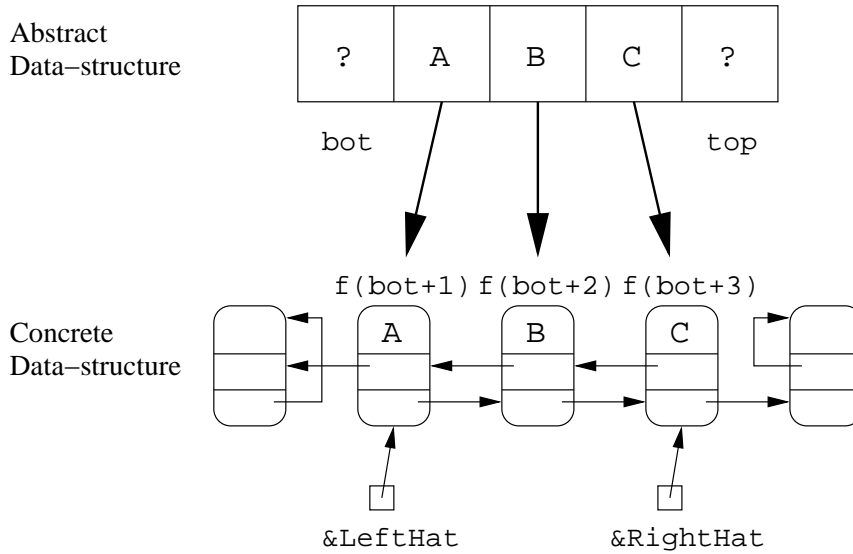


Figure 5.5: The representation function.

- In the transition relation of *SnarkAut*, it is the value in the V field of the node pointed to by the appropriate hat in the pre-state of the $pop_9_yes_p$ or $pop_13_yes_p$ transition (for example $\&RightHat \rightarrow Z$ if the process is popping from the right) leading up to the pop_resp_p action.

Since the step correspondence matches $pop_13_yes_p$ transitions with the transition labelled by do_pop_p , these two values must be equal in related states. Conjunctions (2a), (3a) and (3b) of $obj_ok?$ together imply the required equalities when the abstract and concrete data-structures are non-empty.

5.5 The $rest_ok?$ Predicate

The $rest_ok?$ predicate collects the auxiliary properties required of the simulation relation. Figure 5.6 defines the $rest_ok?$ predicate. The remainder of this section discusses each conjunct of $rest_ok?$.

$$\begin{aligned}
rest_ok?(ab, cc, f) &\stackrel{def}{=} \\
&dead_ok?(ab, cc, f) \wedge conditions_ok?(ab, cc, f) \wedge \\
&nds_ok?(ab, cc, f) \wedge distinctness_ok?(ab, cc, f) \wedge \\
&free_ok?(ab, cc, f)
\end{aligned}$$

Figure 5.6: The *rest_ok?* predicate.

$$\begin{aligned}
dead_ok?(ab, cc, f) &\stackrel{def}{=} \\
&(\forall i \in \mathbb{Z} \bullet \\
&\quad ab.deq.bot < i \wedge i < ab.deq.top \Rightarrow \\
&\quad \neg left_dead?(f(i)) \wedge \neg right_dead?(f(i))) \quad (1) \\
&\wedge \\
&left_dead?(Dummy) \wedge right_dead?(Dummy) \quad (2)
\end{aligned}$$

Figure 5.7: The *dead_ok?* predicate.

5.5.1 The *dead_ok?* Predicate

The *dead_ok?* predicate is presented in Figure 5.7. The *dead_ok?* predicate makes assertions about the concrete data-structure that preclude nodes within the deque containing a self-pointer. It also records the properties required of the *Dummy* node, that it be both left-dead and right-dead. These facts allow processes to correctly determine when a hat variable is pointing to a sentinel node.

Conjunct (1) is used several times in Section 5.7 to help show that the concrete data-structure is empty, during the transitions that rely on that property: those labelled by *push_12_yes_p* and *pop_6_yes_p*.

Conjunct (2) is used in conjunction with the *nds_ok?* (see Section 5.5.3) predicate to help show that after transitions modelling successful push DCASs (those labelled by *push_12_yes_p* and *push_16_yes_p*) the new sentinel nodes have the correct self-pointers.

$$\begin{aligned}
& \text{conditions_ok?}(ab, cc, f) \stackrel{\text{def}}{=} \\
& \forall p \in \text{PROC}\bullet \\
& \quad (\text{pc_push_10_12?}(cc, p) \Rightarrow \tag{1} \\
& \quad \quad \text{cc.hatOut}_p = \text{cc.hat}_p \xrightarrow{\text{cc.h}} \text{onField}(cc, p) \tag{1a} \\
& \quad \quad \vee \\
& \quad \quad \text{not_in_range?}(f, \text{cc.hat}_p, \text{ab.deq.bot}, \text{ab.deq.top})) \tag{1b} \\
& \quad \wedge \\
& \quad (\text{pc_pop_8_9?}(cc, p) \Rightarrow \tag{2} \\
& \quad \quad \text{cc.hat}_p \xrightarrow{\text{cc.h}} \text{onField}(cc, p) \neq \text{cc.hat}_p \tag{2a} \\
& \quad \quad \vee \\
& \quad \quad \text{not_in_range?}(f, \text{cc.hat}_p, \text{ab.deq.bot}, \text{ab.deq.top}) \wedge \\
& \quad \quad \neg\pi_2(\text{pop_9_dcas}(cc, p))) \tag{2b} \\
& \quad \wedge \\
& \quad (\text{pc_pop_12_13?}(cc, p) \Rightarrow \tag{3} \\
& \quad \quad \text{cc.hat}_p \xrightarrow{\text{cc.h}} \text{onField}(cc, p) \neq \text{cc.hat}_p \tag{3a} \\
& \quad \quad \vee \\
& \quad \quad \text{not_in_range?}(f, \text{cc.hat}_p, \text{ab.deq.bot}, \text{ab.deq.top}) \wedge \\
& \quad \quad \neg\pi_2(\text{pop_13_dcas}(cc, p))) \tag{3b}
\end{aligned}$$

Figure 5.8: The *conditions_ok?* Predicate.

5.5.2 The *conditions_ok?* Predicate

The *conditions_ok?* predicate is presented in Figure 5.8. It was in attempting to show that this predicate is preserved across transitions labelled by *pop_13_yes_p* and the matching abstract transition that the bug in the Snark algorithm was found. This issue is explained at the end of this subsection.

The *conditions_ok?* predicate has a certain form: it is a universal quantification over *PROC* whose scope is a series of conjuncts; each conjunct is an implication; the antecedent of each implication is a range of program counter values; the consequent is a predicate over local and global variables. Several of the remaining conjuncts of *rest_ok?* share this form, as do the invariants presented in Section 5.6. The importance of this form derives from features of the actions and the transition relation of *SnarkAut*: each action is labelled by a process; the precondition

of each action includes a predicate over the pc_p variable of the process indexing the action; and the post-state of the action often depends on the pre-state values of variables indexed by the process taking the action. Assertions with a form like that of *conditions_ok?* allow us to constrain the pre-state values of local variables when each transition is enabled, and so to constrain the post-state values of local and global variables.

The final program counter value in each antecedent range of program counter values enables the execution of one of the DCASs that update the concrete data-structure. The second arm of each disjunct ((1b), (2b) and (3b)) is an assertion about the relationship between the local variables of the process and the current global state that is powerful enough to entail that the DCAS will fail. The first arm is a condition on the relationship between the states of the process and the global state that we need to show that the concrete data-structure is empty or not, as required by the simulation relation, if the DCAS will be successful. These conditions are used in the arguments presented in Section 5.7.

Recall that the DCAS at line 12 of the push routines of the Snark algorithm is meant to succeed only when the deque is empty. Conjunct (1) of *conditions_ok?* allows us to show that in pre-states of transitions modelling successful executions of the line 12 DCAS, the concrete data-structure is empty. It does this by asserting that, for processes about to take this transition, either $hatOut_p$ is still the value contained in the field of hat_p pointing out of the concrete data-structure or else hat_p is no longer contained in the concrete data-structure. This, along with an invariant of *SnarkAut* (hat_hatOut , Figure 5.13 on page 99) is enough to show that the node pointed to by the corresponding global hat contains a self-pointer if the DCAS succeeds. As discussed in Section 5.7, This is sufficient to show that the concrete data-structure is empty.

The DCAS at line 16 of the push routines is meant to be executed on non-empty deques and we will need to show that the concrete data-structure is *not* empty in the pre-states of transitions modelling successful executions of this DCAS. However, *conditions_ok?* makes no assertions about processes that are about to take these transitions. This is because we can show that the concrete data-structure

is not empty if the node to which the DCAS is applied does not contain a self-pointer in the field pointing out from the concrete data-structure (for example, if a process is pushing onto the right, we need to know that $\&RightHat \rightarrow Z \rightarrow R \neq \&RightHat \rightarrow Z$). Together, the precondition of $push_16_yes_p$ and the invariant of $SnarkAut\ hat_not_hatOut$ are enough to guarantee this: for any process p , the DCAS will fail in state cc unless $hat_p = onHat(cc, p)$ and $hatOut_p = onHat(cc, p) \xrightarrow{cc, h} onField(cc, p)$ and $SnarkAut\ hat_not_hatOut$ asserts that if p can take the $push_16_yes_p$ then $hat_p \neq hatOut_p$.

Both the DCAS at line 9 and the DCAS at line 13 of the pop routines are meant to succeed only when the deque is not empty. (2a) and (3a) assert that the node pointed to by hat_p is *not* dead, and so the concrete data-structure will not be empty if the DCAS succeeds. (2b) and (3b) explicitly state that the DCASs fail; the assertions in (2b) and (3b), that hat_p not be in the range of the representation function, help to show that once (2b) or (3b) is true of a process, then (2a) or (3a) can never be true of that process again. The idea is that (2a) and (3a) are falsified when the hat_p node is popped from the concrete data-structure and once this happens no modifications to that structure make them true. Consider a process popping from the right, with $pc_p = pc_pop_13$ and $cc.hat_p \xrightarrow{cc, h} onField(cc, p) \neq cc.Hat_p$ (so that condition (3a) is satisfied). If hat_p is popped by some other process, then the conditions for p 's DCAS to succeed are falsified and hat_p is no longer in the specified range of the representation function. This *not_in_range?* condition tells us that further updates to the concrete data-structure will not re-enable p 's DCAS: if hat_p were still in the range of the representation function, it could be set by another pop to be the value of the $\&RightHat \xrightarrow{cc, h} Z$ and p 's DCAS could be enabled.

Unfortunately, it turns out that a process can end up in a situation where it is about to execute a transition modelling the successful execution of a DCAS at line 13 of a pop routine but both (3a) and (3b) are false. This can happen for a process p when the node pointed to by hat_p is the value of *both* hats, and hat_p is removed from the concrete data-structure by a process popping from the other side to p .

Consider a state cc satisfying the conjunction presented in Figure 5.9 for some processes p, q with $p \neq q$. The state cc is like that presented in Figure 6.5, on page

$$\begin{aligned}
cc.pc_p &= pc_pop_13 \wedge \\
cc.pc_q &= pc_pop_13 \wedge \\
cc.side_p &= Right \wedge cc.side_q = Left \wedge \\
cc.hat_p &= \&RightHat \xrightarrow{cc.h} Z \wedge \\
cc.hatIn_p &= \&RightHat \xrightarrow{cc.h} Z \xrightarrow{cc.h} L \wedge \quad \text{so that } p\text{'s DCAS can succeed.} \\
cc.hat_q &= \&LeftHat \xrightarrow{cc.h} Z \wedge \\
cc.hatIn_q &= \&LeftHat \xrightarrow{cc.h} Z \xrightarrow{cc.h} R \wedge \quad \text{so that } q\text{'s DCAS can succeed.} \\
cc.hat_p &\xrightarrow{cc.h} onField(cc, p) \neq cc.hat_p \wedge \quad \text{so that } p \text{ is in condition (3a).} \\
cc.hat_q &\xrightarrow{cc.h} onField(cc, q) \neq cc.hat_q \wedge \quad \text{so that } q \text{ is in condition (3a).}
\end{aligned}$$

Figure 5.9: A state of *SnarkAut* that leads to an incorrect execution.

123 of Chapter 6 with p set to p_2 and q to p_1 . It turns out that q can execute its DCAS (taking the transition labelled by $pop_13_yes_q$) and take *SnarkAut* to a state cc' where $\pi_2(pop_13_dcas(cc', p))$ is true but so is $hat_p \xrightarrow{cc.h} onField(cc', p) = hat_p$. So, in state cc' , neither condition (3a) nor (3b) is true for p . This means that *conditions_ok?* is not preserved across $pop_13_yes_p$ transitions and *SR?* is not a simulation relation.

The reader should consult Chapter 6 to understand the details of how cc is reachable and how it leads the Snark algorithm to malfunction. The point to note here is that the states for which *SR?* fails are those from which the Snark automaton (and the Snark algorithm which it represents) can produce incorrect executions. This provides good evidence that this part of the proposed simulation relation has been well-constructed.

5.5.3 The *nds_ok?* Predicate

The *nds_ok?* predicate is presented in Figure 5.10. Recall that during the push routines, each process sets fields of the its newly allocated node to the values required to make the push operation work correctly. The *nds_ok?* predicate records that these fields contain the correct values. Consider a pushing process p with $side_p = Right$: (1) says that $nd_p \rightarrow R = Dummy$, so that after the node is pushed, the right-hand sentinel will be left-dead; (2) says that the V field of nd_p contains

$$\begin{aligned}
nds_ok?(cc) &\stackrel{def}{=} \\
&\forall p \in PROC \bullet \\
&\quad (pc_push_5_16?(cc, p) \Rightarrow \\
&\quad\quad cc.nd_p \xrightarrow{cc.h} onField(cc, p) = Dummy) \quad (1) \\
&\quad \wedge \\
&\quad (pc_push_7_16?(cc, p) \Rightarrow cc.nd_p \xrightarrow{cc.h} V = cc.val_p) \quad (2) \\
&\quad \wedge \\
&\quad (pc_push_11_12?(cc, p) \Rightarrow \\
&\quad\quad cc.nd_p \xrightarrow{cc.h} offField(cc, p) = Dummy) \quad (3) \\
&\quad \wedge \\
&\quad (pc_push_16 \Rightarrow cc.nd_p \xrightarrow{cc.h} offField(cc, p) = cc.hat_p) \quad (4)
\end{aligned}$$

Figure 5.10: The $nds_ok?$ predicate.

the value being pushed; (3) says that if p is going to attempt to execute the DCAS modelled by $push_12_yes_p$ transitions (the DCAS that should be applied when the deque is empty), then $nd_p \rightarrow L = Dummy$ so that the left-hand sentinel will be right-dead; and (4) says that if p is going to attempt the DCAS modelled by $push_16_yes_p$ transitions (the DCAS that should be applied when the deque is not empty), then the L field of nd_p should point to hat_p , p 's view of the right-hat which a successful DCAS will confirm.

5.5.4 The $distinctness_ok?$ Predicate

The $distinctness_ok?$ predicate is presented in Figure 5.11: $distinctness_ok?$ tackles the issue of *pointer aliasing*. Two pointer expressions (either variables or dereferences of variables) alias each other when they both point to the same object (in our case, a node). When this shared node has a field updated, the properties of *both* expressions can change. For example, consider some process p that is executing a push operation on the right: if its nd_p variable aliases the pointer $\&RightHat \rightarrow Z$ then when p assigns its val_p variable into $nd_p \rightarrow V$, this will modify $\&RightHat \rightarrow Z$, potentially destroying a property required of it by the Snark algorithm. Conjunct (2) of $distinctness_ok?$ asserts that each nd_p references

a node distinct from each hat node, allowing us to prove that, for a concrete state $SR?$ -related to any abstract state, this pathological aliasing never happens.

It may seem that Conject (2) of *distinctness_ok?* is an invariant of *SnarkAut*, not something that depends on the existence of a related abstract state. It is true that (2) is an invariant: however, our simulation relation gives us a direct way to express and verify aliasing properties of *SnarkAut*. Consider what we would need to know about the states of *SnarkAut* in order to show that (2) is preserved across transitions labelled by $pop_13_yes_p$, the successful DCAS transitions that remove a node from the concrete data-structure, with $side_p = Right$. The new value for $\&RightHat \rightarrow Z$ comes from *within the concrete data-structure* so we need a way to specify that nd_p does not point to any node within the concrete data-structure. Our representation function allows us to do that easily: Conject (1) of *distinctness_ok?* achieves this directly, using the *not_in_range?* predicate, which in turn depends on the representation function f .

Conjuncts (3)-(7) of *distinctness_ok?* all assert inequalities between the variables nd_p and hat_p of each process and the nodes in the global state of *SnarkAut* (the values of global variables and the nodes which constitute the concrete data-structure). Each conjunct is an implication where the antecedent asserts the pc_p variable of each process is in range of values: this range must encompass the points at which a process will modify the node pointed to by each variable. This ensures that we can show that unintended side-effects of each update do not happen. The ranges of program-counter values must also encompass the point at which it is first true that the required pointer expressions are distinct. For example, we need to know that each hat_p is distinct from $\&RightHat \rightarrow Z$ when p takes the transition pop_15_p which modifies the node referenced by hat_p : but to know this we need to know that $hat_p \neq \&RightHat \rightarrow Z$ after p removed the node from the concrete data-structure. Therefore, the antecedent to (5) of *distinctness_ok?* is $pc_pop_14_15?(cc, p)$.

Note that the antecedents of Conject (3) and (7) assert that the concrete data-structure is not empty. These conjuncts protect the properties required of the sentinel nodes when the concrete data-structure is not empty.

$$\begin{aligned}
& \text{distinctness_ok?}(ab, cc, f) \stackrel{\text{def}}{=} \\
& (\forall p, q \in \text{PROC} \bullet \\
& \quad (\text{pc_push_3_16?}(cc, p) \Rightarrow \\
& \quad \quad \text{not_in_range?}(f, cc.nd_p, ab.deq.bot, ab.deq.top)) \tag{1} \\
& \quad \wedge \\
& \quad (\text{pc_push_3_16?}(cc, p) \Rightarrow \\
& \quad \quad cc.nd_p \neq \&\text{LeftHat} \xrightarrow{cc.h} Z \wedge cc.nd_p \neq \&\text{RightHat} \xrightarrow{cc.h} Z) \tag{2} \\
& \quad \wedge \\
& \quad (\text{pc_push_3_16?}(cc, p) \wedge \neg \text{empty_dll_state?}(cc) \Rightarrow \\
& \quad \quad cc.nd_p \neq \&\text{LeftHat} \xrightarrow{cc.h} Z \xrightarrow{cc.h} L \wedge \\
& \quad \quad cc.nd_p \neq \&\text{RightHat} \xrightarrow{cc.h} Z \xrightarrow{cc.h} R) \tag{3} \\
& \quad \wedge \\
& \quad (\text{pc_pop_14_15?}(cc, p) \Rightarrow \\
& \quad \quad \text{not_in_range?}(f, cc.hat_p, ab.deq.bot, ab.deq.top)) \tag{4} \\
& \quad \wedge \\
& \quad (\text{pc_pop_14_15?}(cc, p) \Rightarrow \\
& \quad \quad cc.hat_p \neq \&\text{LeftHat} \xrightarrow{cc.h} Z \wedge cc.hat_p \neq \&\text{RightHat} \xrightarrow{cc.h} Z) \tag{5} \\
& \quad \wedge \\
& \quad (\text{pc_push_8_16?}(cc, p) \vee \text{pc_pop_4_12?}(cc, p) \vee \\
& \quad \quad \text{pc_pop_14_15?}(cc, p) \Rightarrow cc.hat_p \neq \text{Dummy}) \tag{6} \\
& \quad \wedge \\
& \quad (\text{pc_pop_14_15?}(cc, p) \wedge \neg \text{empty_dll_state?}(cc) \Rightarrow \\
& \quad \quad cc.hat_p \neq \text{offHat}(cc, p) \xrightarrow{cc.h} \text{offField}(p)) \tag{7} \\
& \quad \wedge \\
& \quad (\text{pc_pop_14_15?}(cc, p) \wedge \text{pc_pop_14_15?}(cc, q) \wedge p \neq q \Rightarrow \\
& \quad \quad cc.hat_p \neq cc.hat_q) \tag{8} \\
& \quad \wedge \\
& \quad (\text{pc_pop_14_15?}(cc, p) \wedge \text{pc_push_3_16?}(cc, q) \Rightarrow \\
& \quad \quad cc.hat_p \neq cc.nd_q) \tag{9} \\
& \quad \wedge \\
& \quad (\text{pc_push_3_16?}(cc, p) \wedge \text{pc_push_8_16?}(cc, q) \Rightarrow \\
& \quad \quad cc.nd_p \neq cc.hat_q) \tag{10} \\
& \quad \wedge \\
& \quad (\text{pc_push_3_16?}(cc, p) \wedge \text{pc_pop_4_12?}(cc, q) \Rightarrow \\
& \quad \quad cc.nd_p \neq cc.hat_q) \tag{11}
\end{aligned}$$

Figure 5.11: The *distinctness_ok?* predicate.

$$\begin{aligned}
free_ok?(ab, cc, f) &\stackrel{def}{=} \\
&(\forall i \in \mathbb{Z} \bullet \\
&\quad ab.deq.bot < i \wedge i < ab.deq.top \Rightarrow f(i) \notin cc.h.free \quad (1) \\
&\wedge \\
&\forall p \in PROC \bullet \\
&\quad pc_pop_4_12? \vee pc_push_8_16? \Rightarrow hat_p \notin cc.h.free \quad (2)
\end{aligned}$$

Figure 5.12: The *free_ok?* predicate.

Conjuncts (8)-(11) of *distinctness_ok?* assert inequalities between local variables: these assertions ensure that the modification of a node locally accessed by one process does not have unintended consequences for the local variables belonging to that process or another process. The antecedent program-counter values perform the same function as for conjuncts (3)-(7).

5.5.5 The *free_ok?* Predicate

The *free_ok?* predicate is presented in Figure 5.12. This predicate records that all the nodes in the concrete data-structure and all the nodes pointed to by *hat_p* variables have been properly allocated.

Much of the *distinctness_ok?* predicate makes assertions about the *nd_p* variables and *free_ok?* is needed to make those assertions provable. Recall that the *nd_p* variable is allocated by *p* using the *new* function: the *new* function chooses an arbitrary pointer from the *free* set, removes it from that set, and returns the chosen pointer. If we want to show that each *nd_p* variable is distinct from each *hat_p* variable or node within the range of the representation function, then we need to be able to show that these variables and nodes are *not* in the *free* set: the *free_ok?* predicate allows us to show this.

As with the *distinctness_ok?* predicate, *free_ok?* needs to be embedded within the simulation relation: this allows us to express the claim that none of the nodes within the concrete data-structure are free.

5.6 Invariants of *SnarkAut*

Figure 5.13 presents the invariants of *SnarkAut* used in this verification. These invariants can be divided into three classes:

1. Distinctness properties simple enough to be proved without use of the simulation relation: *nds_distinct?* and *nds_not_Dummy?*.
2. Properties needed to show that the first class of properties are invariant: *nd_used?* and *dummy_used?*.
3. Control properties that only mention the local variables of processes and are true over ranges of program counter values by virtue of tests executed by the Snark automaton: *hat_hatOut?*, *hat_not_hatOut* and *hat_otherHat*.

The *nds_distinct?* invariant asserts that two different processes never allocate the same new node: this allows us to show that updates by one process to its *nd* variable do not interfere with the *nd* variables of other processes. The *nd_not_dummy?* invariant asserts that a process never allocates the *Dummy* node: this allows us to show that updates to *nd* variables do not affect *Dummy*.

The invariants of class (2) assert that *Dummy* and the *nd* variables are never free. Invariants of class (1) rely on invariants of class (2) in just the same way as the *distinctness_ok?* predicate relies on the *free_ok?* predicate. For example, in order to show that each *nd* variable references a node distinct from *Dummy* we need to know that *Dummy* is never *free*: the invariant *dummy_used?* satisfies this requirement. The invariant *nd_used?* satisfies the corresponding requirement for showing that the *nd* variables of distinct processes are distinct. These invariants are particularly easy to prove, since non-*free* nodes are never made *free*.

Class (3) invariants record the results of tests applied to the local variables of each process during its execution. The *hat_hatOut?* and *hat_not_hatOut?* record the result of the test at line 9 of the push routines. These invariants are used to help show that the concrete data-structure is empty or not empty if the appropriate DCAS can succeed. The *hat_otherHat?* invariant records that for processes which attempt to execute the DCAS at line 9 of the pop routines, the test at line 8 was

$$\begin{aligned}
nds_distinct?(cc) &\stackrel{def}{=} \\
&\forall p, q \in PROC \bullet \\
&\quad pc_push_3_16?(cc, p) \wedge pc_push_3_16?(cc, q) \Rightarrow \\
&\quad cc.nd_p \neq cc.nd_q
\end{aligned}$$

$$\begin{aligned}
nd_not_dummy?(cc) &\stackrel{def}{=} \\
&\forall p \in PROC \bullet \\
&\quad pc_push_3_16?(cc, p) \Rightarrow \\
&\quad cc.nd_p \neq Dummy
\end{aligned}$$

$$\begin{aligned}
nd_used?(cc) &\stackrel{def}{=} \\
&\forall p \in PROC \bullet \\
&\quad pc_push_3_16?(cc, p) \Rightarrow \\
&\quad cc.nd_p \notin cc.h.free
\end{aligned}$$

$$\begin{aligned}
dummy_used?(cc) &\stackrel{def}{=} \\
&Dummy \notin cc.h.free
\end{aligned}$$

$$\begin{aligned}
hat_otherHat?(cc) &\stackrel{def}{=} \\
&\forall p \in PROC \bullet \\
&\quad pc_p = pc_pop_9 \Rightarrow \\
&\quad cc.hat_p = cc.otherHat_p
\end{aligned}$$

$$\begin{aligned}
hat_not_hatOut?(cc) &\stackrel{def}{=} \\
&\forall p \in PROC \bullet \\
&\quad pc_push_15_16?(cc, p) \Rightarrow \\
&\quad cc.hat_p \neq cc.hatOut_p
\end{aligned}$$

$$\begin{aligned}
hat_hatOut?(cc) &\stackrel{def}{=} \\
&\forall p \in PROC \bullet \\
&\quad pc_push_10_12? \Rightarrow \\
&\quad cc.hat_p = cc.hatOut_p
\end{aligned}$$

Figure 5.13: Invariants of the Snark automaton.

successful. This invariant is used to show that if the DCAS can succeed then the concrete data-structure contains exactly one element.

5.7 Verifying the Simulation Relation

Recall that the step correspondence matches actions of the form $push_16_yes_p$ and $pop_13_yes_p$ (modelling the successful DCASs at lines 16 of the push routines and 13 of the pop routines, respectively) with the abstract actions $do_push_p(t)$ and do_pop_p . This section describes in detail how to prove that the $obj_ok?$ predicate is preserved across the actions $push_16_yes_p$ and $pop_13_yes_p$ under the assumption that $SR?$ is true in the pre-states of the associated concrete and abstract transitions. This helps to illustrate how the representation function is used in the proposed simulation relation and so helps to motivate the core part of that relation.

Certain other concrete actions are matched by the step correspondence with the abstract actions $do_push_p(t)$ and do_pop_p , namely $push_12_yes_p$, $pop_6_yes_p$ and $pop_9_yes_p$. Less detailed proofs that $obj_ok?$ is preserved across these actions under the same assumption. These proofs help to flesh out how the parts of $SR?$ which relate to showing emptiness conditions are used.

The proofs described in this section are the most difficult that were undertaken in the attempted verification of the Snark algorithm. The proof that $SR?$ is a simulation relation was separated out into a series of lemmas. Each lemma asserts that for one action, $SR?$ is preserved across that action. Some additional supporting lemmas were stated, asserting the preservation of particular conjuncts of $SR?$ or certain properties of states of *SnarkAut*. Proofs for all the lemmas asserting preservation for actions other than $poply13$ have been constructed in the PVS proof-checker, although some of the lemmas on which these proofs depend have been left unproven (the PVS theories and proofs are available from the author).

It should be noted that $SR?$ *cannot* be shown to be preserved across actions of the form $pop_13_yes_p$: the $conditions_ok?$ conjunct fails. The predicate $obj_ok?$ is shown to be preserved here under the assumption that the concrete and abstract

pre-states of the transitions are related by $SR?$.

5.7.1 What Happens During Push Operations?

Figure 5.14 illustrates the relationship between related concrete and abstract states immediately before some process p attempts to execute the transition labelled $push_16_yes_p$ with $side_p = Right$. Let cc and ab be related concrete and abstract states as represented by this diagram; the node marked nd stands for $cc.nd_p$; likewise, nodes marked hat and $hatOut$ denote $cc.hat_p$ and $cc.hatOut_p$. The figure represents concrete and abstract data-structures that contain one element, but that detail is not relevant to the rest of the discussion. It also indicates that $Dummy$ is distinct from both sentinels, which may not be true.

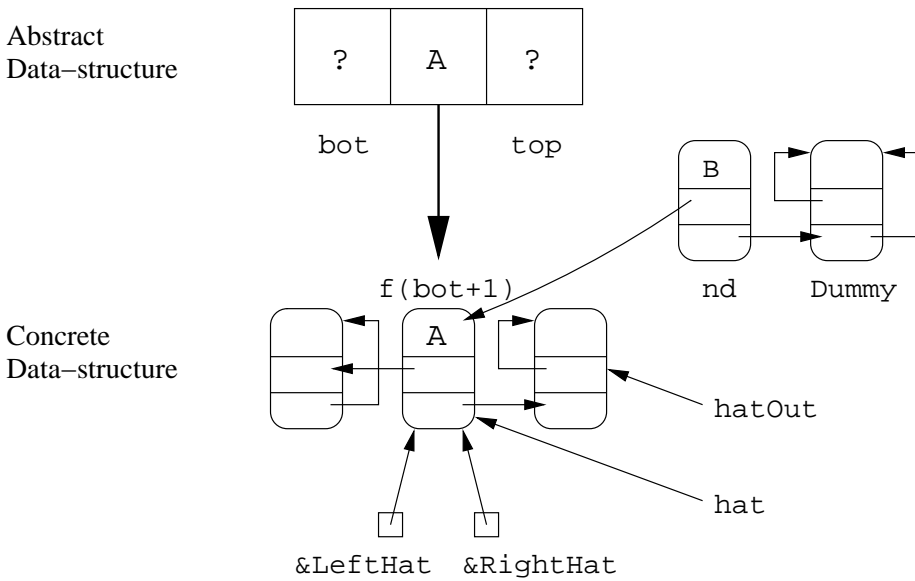


Figure 5.14: The representation function before a push DCAS on the right.

This section describes in detail how to show that $obj_ok?$ is preserved across actions of the form $push_16_yes_p$ when $side_p = Right$. The case when $side_p = Left$ is symmetric. Issues relating to the preservation of $SR?$ across the $push_12_yes_p$ actions are discussed briefly.

$$\begin{aligned}
cc.pc_p &= pc_push_16 && \text{(pre1)} \\
cc.hat_p &= \&RightHat_p \xrightarrow{cc.h} Z && \text{(pre2)} \\
cc.hatOut_p &= \&RightHat_p \xrightarrow{cc.h} Z \xrightarrow{cc.h} L && \text{(pre3)} \\
cc'.pc_p &:= push_resp && \text{(eff1)} \\
cc'.h &:= update_p(update_p(cc.h, \&RightHat, Z, cc.nd_p), && \\
&\quad \&RightHat \xrightarrow{cc.h} Z, R, cc.nd_p) && \text{(eff2)}
\end{aligned}$$

Figure 5.15: The $push_16_yes_p$ transition with $side_p = Right$.

$$\begin{aligned}
ab.pc_p &= do_push(cc.val_p) && \text{(pre1')} \\
ab'.deq.seq &:= ab.deq.seq[ab.deq.top := val_p] && \text{(eff1')} \\
ab'.deq.top &:= ab.deq.top + 1 && \text{(eff2')} \\
ab'.pc_p &:= push_resp && \text{(eff3')}
\end{aligned}$$

Figure 5.16: The $do_push_p(cc.val_p)$ transition with $side_p = Right$.

We need to marshal the premises that will be used in the argument that $obj_ok?$ is preserved across $push_16_yes_p$ actions. Figure 5.15 presents the preconditions and effects of the $push_16_yes_p$ action with pre-state cc , post-state cc' and $cc.side_p = Right$, unpackaged for convenience. Figure 5.16 presents the precondition and effect of the abstract transition labelled by $do_push_p(cc.val_p)$ with pre-state ab , post-state ab' and $ab.side_p = Right$. The assignments used in these two figures are meant to indicate that no other state variables change.

Figure 5.17 presents some assertions about processes and related concrete and abstract states and Lemma 5.1 collects certain inferences from these preconditions and assertions.

We need to be able to show that (pre1') is true, given that $SR?(cc, ab)$ so that $DeqAut$ is actually able to take the $do_push_p(cc.val_p)$ transition. This is a direct consequence of conjunct (2) of (pre1) and $correspondence_ok?$ (see Figure 5.1).

The $rest_ok?$ predicate and certain invariants of $SnarkAut$ guarantee that we

1. $cc.nd_p \rightarrow R = Dummy$ (*nds_ok?*, see Section 5.5.3)
2. $left_dead?(cc, Dummy) \wedge right_dead?(cc, Dummy)$
(*dead_ok?*, see Section 5.5.1)
3. $cc.nd_p \rightarrow V = cc.val_p$ (*nds_ok?*, see Section 5.5.3)
4. $cc.nd_p \rightarrow L = cc.hat_p$ (*nds_ok?*, see Section 5.5.3)
5. $cc.nd_p \neq \&RightHat \xrightarrow{cc.h} Z$ (*distinctness_ok?*, see Section 5.5.4)
6. $cc.nd_p \neq Dummy$ (the invariant *nds_not_dummy?*, see Section 5.6)
7. $cc.hat_p \neq cc.hatOut_p$ (the invariant *hat_not_hatOut?*, see Section 5.6)
8. $not_in_range?(f, ab.deq.bot, ab.deq.top, cc.nd_p)$
(*distinctness_ok?*, see Section 5.5.4)

Figure 5.17: Assertions needed to prove the preservation of *obj_ok?*.

are able to prove the assertions of Figure 5.17: the bracketed annotations indicate which conjuncts of *rest_ok?* or invariants assert each of these properties and where a discussion of these conjuncts can be found.

Lemma 5.1 *The following assertions are true for all concrete states $cc, cc' \in reach(SnarkAut)$ and abstract states $ab \in reach(DeqAut)$, and $p \in PROC$ such that $SR?(cc, ab)$, $cc \xrightarrow{push_16_yes_p} cc'$ and $cc.side_p = Right$:*

1. $ab.pc_p = do_push(cc.val_p)$
2. $cc.nd_p \xrightarrow{cc.h} L = \&RightHat \xrightarrow{cc.h} Z$
3. $cc'.nd_p \xrightarrow{cc'.h} R = Dummy$
4. $\neg right_dead?(cc, \&RightHat \xrightarrow{cc.h} Z)$
5. $left_dead?(cc'.h, Dummy)$

6. $ab.deq.bot < ab.deq.top - 1$

Proof: (2) can be seen by considering (pre2) and (4) of Figure 5.17.

(3) is a *preservation lemma*: it tells us that a property true of the pre-state is preserved by the transition. Observe that the only R field updated during the transition belongs to $\&RightHat \xrightarrow{cc.h} Z$, but by (5) of Figure 5.17 that node is distinct from nd_p , given that (1) says the property is true in the pre-state, the property is true in the post-state.

(7) of Figure 5.17 together with preconditions (pre1) and (pre2) tell us that $\&RightHat \xrightarrow{cc.h} Z$ is not right-dead and so (4) of Lemma 5.1 is true.

(5) can be seen by 2) of Figure 5.17 and by observing that no L fields are modified during the transition.

(6) is true by (7) of Figure 5.17 together with (pre1) and (pre2). $Q.E.D.$

We need to be able to construct a new representation function to witness $obj_ok?$ after the DCAS of $push_16_yes_p$ successfully completes. As mentioned in Section 5.1, transitions with labels of the form $do_push_p(t)$ are the only ones for which the representation function is updated. This new function f' is defined thus:

$$f' = \begin{cases} f[ab.deq.bot := cc.nd_p] & \text{if } cc.side_p = Left \\ f[ab.deq.top := cc.nd_p] & \text{if } cc.side_p = Right \end{cases}$$

Figure 5.18 illustrates the representation function, and the concrete and abstract data-structures after the push is complete: the dotted arrows indicate the pointer fields that have changed.

Given the assertions of Figure 5.17, Lemma 5.1 and the injectivity of f , we can argue that f' is injective and does witness $obj_ok?$ after the pair of transitions, $push_16_yes_p$ and $do_push_p(val_p)$.

Lemma 5.2 (Preservation of $obj_ok?$ across $push_16_yes_p$ transitions) *For any $p \in PROC$, $cc, cc' \in reach(SnarkAut)$, and $ab, ab' \in reach(DeqAut)$ such that $SR?(cc, ab)$, $cc \xrightarrow{push_16_yes_p} cc'$, $ab \xrightarrow{do_push_p(val_p)} ab'$ and $cc.side_p = Right$, letting*

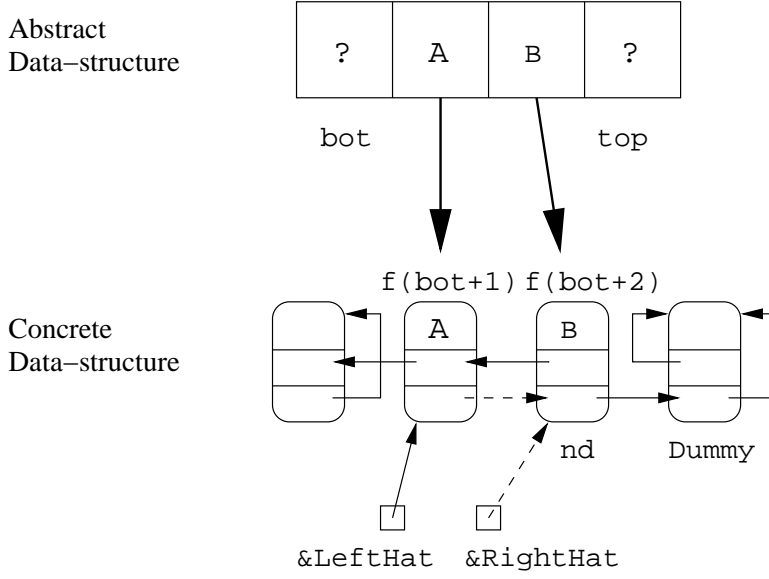


Figure 5.18: The representation function after a push DCAS.

f be the function witnessing $SR?(cc, ab)$ and f' be the function as defined above, $obj_ok?(cc', ab', f')$ and $injective_in_range?(f', ab'.deq.bot, ab'.deq.top)$.

Proof: Conjunct (1) of $obj_ok?$ is preserved. By (eff2') we know that $ab'.deq$ is not empty, so it is sufficient to know that $\&RightHat \xrightarrow{cc'.h} Z$ is not right-dead which will show that the concrete data-structure is not empty. The effect of the DCAS of $push_16_yes_p$ means that $\&RightHat \xrightarrow{cc'.h} Z = cc.nd_p$, and by (1) and (6) of Figure 5.17, we know that this new value for $\&RightHat \rightarrow Z$ was not right-dead in the pre-state of the transition. It is not right-dead in the post-state of the transition, since the only R field updated belongs to $\&RightHat \xrightarrow{cc.h} Z$, and by (5) of Figure (5.17), $nd_p \neq \&RightHat \xrightarrow{cc.h} Z$.

Conjunct (2) of $obj_ok?$ is preserved. The argument for this is based first on observing that for all $i \in \mathbb{Z}$, $ab'.deq.bot < i \wedge i < ab'.deq.top$ if and only if $ab.deq.bot < i \wedge i < ab.deq.top \vee i = ab.deq.top$ (by the transition relation of $DeqAut$). Since no V fields are updated, we know that for all i between $ab.deq.bot$ and $ab.deq.top$ not inclusive, (2a) is preserved. We also know that $f'(ab'.deq.top - 1) \xrightarrow{cc'.h} V = ab'.deq.seq(ab'.deq.top - 1)$ by the definition of f'

and (3) of Figure 5.17, $nd_p \rightarrow V = val_p$. Conject (2b) is preserved for all i where $ab.deq.bot < i < ab.deq.top$ because no L fields are updated during the transition and by (2) of Lemma 5.1. (2b) is preserved for $i = ab.deq.top$ by (2) of Figure 5.17 and (pre2). Conject (2c) is preserved for all i where $ab.deq.bot < i < ab.deq.top$ by the injectivity of f : that is, the only R field updated belongs to $\&RightHat \xrightarrow{cc.h} Z$, and this pointer value does not appear anywhere else in the deque. Also, $f'(ab'.deq.top - 1) = cc.nd_p = f(ab.deq.top - 1) \xrightarrow{cc'.h} R$ by the preconditions and effects of the DCAS and by (3b) of *obj_ok?*.

Conject (3) of *obj_ok?* is preserved. (3a) is preserved because, by (6) of Lemma 5.1 we know that $ab.deq.bot+1 \neq ab.deq.top$ so that $f'(ab'.deq.bot+1) = f(ab'.deq.bot + 1) = f(ab.deq.bot + 1) = \&LeftHat \xrightarrow{cc.h} Z = \&LeftHat \xrightarrow{cc'.h} Z$. (3b) is preserved because $f'(ab'.deq.top - 1) = cc.nd_p = \&RightHat \xrightarrow{cc'.h} Z$. (3c) is preserved: since no L fields are updated and the value of $\&LeftHat \xrightarrow{cc'.h} Z$ is not changed, we know that $\&LeftHat \xrightarrow{cc'.h} Z \xrightarrow{cc'.h} L = \&LeftHat \xrightarrow{cc.h} Z \xrightarrow{cc.h} L$; moreover, the node to the left of $\&LeftHat \xrightarrow{cc.h} Z$ is right-dead, so by (4) of Lemma 5.1 and the fact that the only R field modified during the transition was that belonging to the right-hat, we know that $\&LeftHat \xrightarrow{cc'.h} Z \xrightarrow{cc'.h} L$ is right-dead. (3d) is preserved by (3) and (5) of Lemma 5.1.

The required injectivity of f' can be seen by observing that $cc.nd_p$ was not in the range of f between the bottom and top of the pre-state data-structure (by 8 of Figure 5.17) and the injectivity of f given that f witnesses *SR?*. $Q.E.D.$

Note how the injectivity of f is used to show that properties required by parts of the concrete data-structure that were not directly updated during the transition are maintained.

The proof obligations and style of reasoning associated with actions of the form *push_I2_yes_p* (the DCAS attempted if a process believes the data-structure is currently empty) are similar. There is one issue which does not arise for *push_I6_yes_p* transitions: we need to know that the abstract data-structure is empty. This is because the *push_I2_yes_p* transitions update *both* hats, setting them to nd_p : if the deque contains any elements, those elements will be lost.

By (1) of the *obj_ok?* predicate, we know that $ab.deq$ is empty if we know

that the concrete data-structure is empty. If we know the following facts about the state cc of *SnarkAut* when a process p is about to take the transition labelled by $push_I2_yes_p$, then we can conclude that $empty_dll_state?(cc)$ is true:

1. $cc.hat_p = cc.hatOut_p$ (the invariant $hat_hatOut?$, see Section 5.6)
2. $cc.hatOut_p = cc.hat_p \xrightarrow{cc.h} R$ ($conditions_ok?$, see Section 5.5.2)
3. $\forall i \in \mathbb{Z} \bullet ab.deq.bot < i \wedge i < ab.deq.top \Rightarrow$
 $\neg right_dead?(cc, f(i))$ ($dead_ok?$, see Section 5.5.1)

Given that the precondition of $push_I2_yes_p$ states that $cc.hat_p = \&RightHat \xrightarrow{cc.h} Z$, (1) and (2) above together imply that $\&RightHat \xrightarrow{cc.h} Z$ is right-dead. Because conjunct (3b) of $obj_ok?$ asserts that $f(ab.deq.top - 1) = \&RightHat \xrightarrow{cc.h} Z$ when the concrete data-structure is not empty, we know that if the deque were not empty, there would be an index of the abstract deque taken to a right-dead element of the concrete data-structure. This contradicts (3) above, so both the concrete and abstract data-structures are empty.

5.7.2 What Happens During Pop Operations?

Recall that the step correspondence matches transitions labelled by $pop_6_yes_p$, $pop_9_yes_p$ and $pop_I3_yes_p$ with the abstract transition labelled by do_pop_p . This subsection describes how to show that $obj_ok?$ is preserved across concrete transitions labelled by $pop_I3_yes_p$ where $side_p = Right$, matched with the corresponding abstract transition. For the sake of brevity, the argument will be presented in less detail than that of Section 5.7.1.

Figure 5.19 illustrates the the relationship between some related concrete and abstract states immediately before some process p attempts to execute the transition labelled $pop_I3_yes_p$ with $side_p = Right$. As before, the exact number of elements in the concrete data-structure is not relevant, so long as there is more than one; also, the sentinel nodes may not be distinct.

Figure 5.20 presents the transition of the $pop_I3_yes_p$ action that we are about to discuss, where cc and cc' are the abstract pre- and post- states and ab and

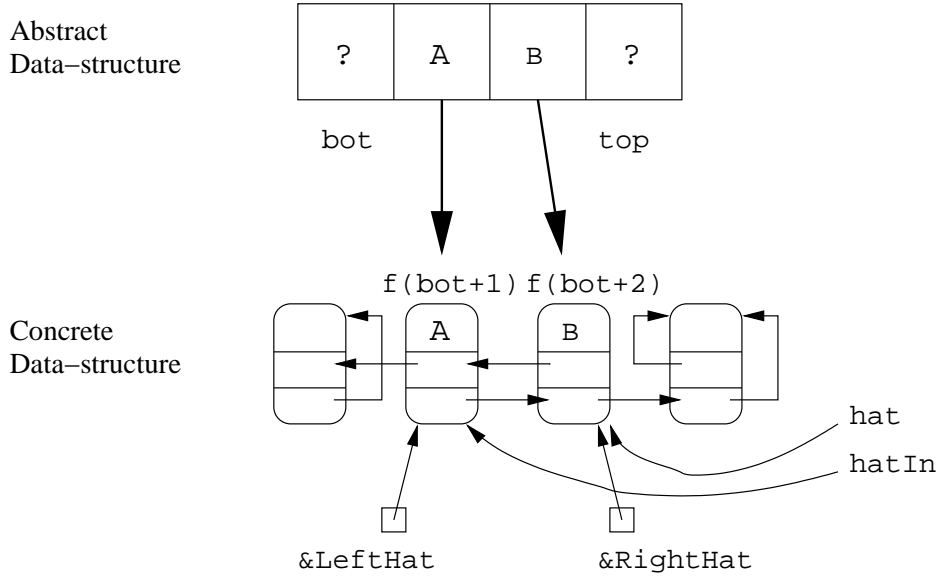


Figure 5.19: The representation function before a pop DCAS.

ab' are the concrete pre- and post-states. Lemma 5.3 collects certain assertions about related concrete and abstract states which are true when the preconditions of $pop_l3_yes_p$ are met, many of which are illustrated by Figure 5.19. This lemma is stated without proof, but some indication is given of the parts of the $rest_ok?$ predicate that would be used to construct a proof for this Lemma.

$$cc.pc_p = pc_pop_l3 \quad (\text{pre1})$$

$$cc.hat_p = \&RightHat_p \xrightarrow{cc.h} Z \quad (\text{pre2})$$

$$cc.hatIn_p = \&RightHat_p \xrightarrow{cc.h} Z \xrightarrow{cc.h} L \quad (\text{pre3})$$

$$cc'.h = update_p(update_p(\&RightHat, Z, cc.hatIn_p), \&RightHat \xrightarrow{cc.h} Z, R, hat_p) \quad (\text{eff1})$$

Figure 5.20: Preconditions of $pop_l3_yes_p$ with $side_p = Right$.

Again, we have to show that (pre1') is true given $SR?(cc, ab)$. This is a consequence of (pre1) and conjunct (4) of $correspondence_ok?$ (see Figure 5.1).

$$\begin{aligned}
ab.pc_p &= do_pop_p && (\text{pre1}') \\
ab'.deq.top &:= ab.deq.top - 1 && (\text{eff1}') \\
ab'.pc_p &:= pop_resp(ab.deq.seq(ab.deq.top - 1)) && (\text{eff2}')
\end{aligned}$$

Lemma 5.3 *The following assertions are true for all concrete states $cc, cc' \in \text{reach}(\text{SnarkAut})$ and abstract states $ab \in \text{reach}(\text{DeqAut})$, and $p \in \text{PROC}$ such that $SR?(cc, ab)$ and $cc \xrightarrow{\text{pop-13-yes}_p} cc'$:*

1. $\forall i \in \mathbb{Z} \bullet$
 $ab.deq.bot < i \wedge i < ab.deq.top \Rightarrow \neg \text{right_dead?}(cc, f(i))$, so that there are no right-dead nodes in the concrete data-structure.
2. $ab.deq.bot < ab.deq.top - 1$, so that the abstract deque is not empty.
3. $\text{left_dead?}(cc'.h, \&\text{RightHat} \xrightarrow{cc.h} Z)$, so that the pre-state value of the right-hat is left-dead in the post-state.
4. $\&\text{RightHat} \xrightarrow{cc'.h} Z = \&\text{RightHat} \xrightarrow{cc.h} Z \xrightarrow{cc.h} L$, so that the post-state right-hat is the node to the left of the pre-state right-hat.

Proof: Each assertion of Lemma 5.3 can be proven under the assumption that $SR?(cc, ab)$ holds. Here, we only discuss (2).

(2) holds: by conjunct (3) of conditions_ok? we know that hat_p is not right-dead so the concrete data-structure is not empty. So, by (1) of obj_ok? we know that the abstract data-structure is not empty. Note that we are not able to show that there is more than one element in either data-structure: recall that the line 13 DCAS can succeed if applied when the deque has only one element. Also, note that conjunct (3) of conditions_ok? is not preserved across the pop-13-yes_p action.

Figure 5.21 illustrates the situation after the DCAS has removed the rightmost node from the deque. Note that we do not need to modify the representation function: as described in Lemma 5.4, the representation function which witnessed $SR?$ in the pre-state can be used to represent $SR?$ in the post-state.

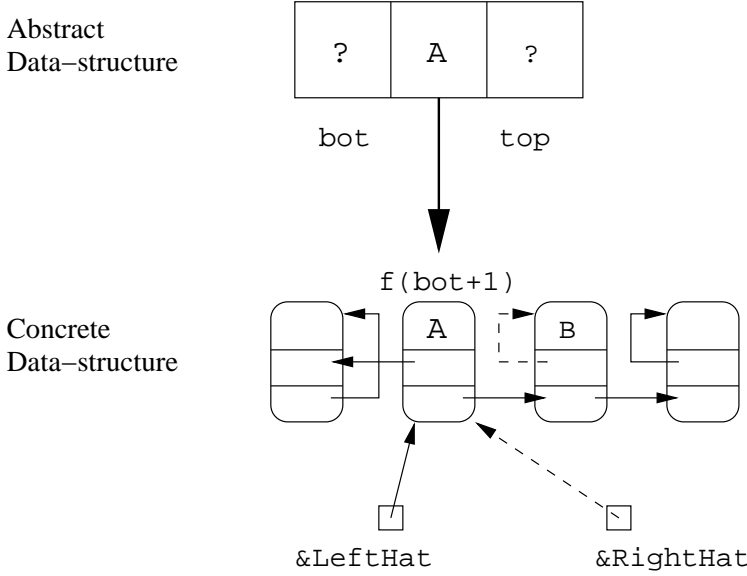


Figure 5.21: The representation function after a pop DCAS.

Lemma 5.4 (Preservation of $obj_ok?$ across $pop_13_yes_p$ transitions) For any $p \in PROC$, $cc, cc' \in reach(SnarkAut)$, and $ab, ab' \in reach(DeqAut)$ such that $SR?(cc, ab)$, $cc \xrightarrow{pop_13_yes_p} cc'$, $ab \xrightarrow{do_pop_p} ab'$ and $cc.side_p = Right$, letting f be the function witnessing $SR?(cc, ab)$, $obj_ok?(cc', ab', f')$ and $injective_in_range?(f', ab'.deq.bot, ab'.deq.top)$.

The preservation of each conjunct depends on a case analysis: by (2), either $ab.deq.bot = ab.deq.top - 2$ or $ab.deq.bot < ab.deq.top - 2$. The first case holds when the abstract data-structure has one element, and the second when the abstract data-structure contains more than one element. As noted below, showing the preservation of Conjuncts (2) and (3) of $obj_ok?$ in the first case is trivial.

Conjunct (1) of $obj_ok?$ is preserved. First, consider the case where there is more than one element in the abstract data-structure (as presented in Figure 5.19), so that $ab.deq.bot < ab.deq.top - 2$. In this case it suffices to show that $\&RightHat \xrightarrow{cc', h} Z$ is not right-dead since the abstract data-structure will not be empty after the abstract transition. By (1) of Lemma 5.3, together with conjunct (2b) of $obj_ok?$ and the assumption that $ab.deq.bot < ab.deq.top - 2$ we know that

$\&RightHat \xrightarrow{cc,h} Z \xrightarrow{cc,h} L$ is not right-dead. By the fact that no R fields are modified during the transition, we know that this node is not right-dead in the post-state either, but by (4) of Lemma 5.3 this node is the value of $\&RightHat \xrightarrow{cc',h} Z$.

When $ab.deq.bot = ab.deq.top - 2$, we need to show that $\&LeftHat \xrightarrow{cc',h} Z$ and $\&RightHat \xrightarrow{cc,h} Z$ are left- and right-dead respectively because after the abstract transition, the abstract data-structure will be empty. Both requirements depend on the fact that when the deque contains one element $\&LeftHat \xrightarrow{cc,h} Z = \&RightHat \xrightarrow{cc,h} Z$. This can be seen by considering Conjuncts (3a) and (3b) of $obj_ok?$. To see that $\&LeftHat \xrightarrow{cc',h} Z$ is left-dead observe that $\&LeftHat \xrightarrow{cc',h} Z = \&LeftHat \xrightarrow{cc,h} Z = \&RightHat \xrightarrow{cc,h} Z$, but by (3) of Lemma 5.3, this last node is left-dead in the post-state.

Now by (4) of Lemma 5.3, $\&RightHat \xrightarrow{cc',h} Z = \&RightHat \xrightarrow{cc,h} Z \xrightarrow{cc,h} L = \&LeftHat \xrightarrow{cc,h} Z \xrightarrow{cc,h} L$, but by (3c) of $obj_ok?$, this last node was right-dead in the pre-state. As usual, the fact that no R fields are modified during the transition implies that this node is right-dead in the post-state also.

Conjunct (2) of $obj_ok?$ is preserved. We need only consider the case where $ab.deq.bot < ab.deq.top - 2$, because if $ab.deq.bot = ab.deq.top - 2$ then $ab'.deq.bot = ab'.deq.top - 1$ and (2) will place no constraints on any pointers. Observe that, by the transition relation for $DequeAut$, $ab'.deq.bot + 1 < i < ab'.deq.top - 1$ if and only if $ab.deq.bot + 1 < i < ab.deq.top - 2$. Since no V fields are updated during the transition (2a) is preserved. The only L field updated belongs to $\&RightHat \xrightarrow{cc,h} Z$ and since f is injective over the required range, this update does not modify any of the L fields of any of the $f(i)$ for any $i < ab.deq.top - 2$, so (2b) is preserved. (2c) is preserved because no R fields are updated.

Conjunct (3) of $obj_ok?$ is preserved. Again, we only need to consider the case where $ab.deq.bot < ab.deq.top - 2$. (3a) is preserved because $ab'.deq.bot = ab.deq.bot$ and $\&LeftHat \xrightarrow{cc',h} Z = \&LeftHat \xrightarrow{cc,h} Z$. (3b) is preserved, since by (2b) and (3b) of $obj_ok?$ and the non-emptiness of the abstract data-structure, we know that $f(ab.deq.top - 2) = \&RightHat \xrightarrow{cc,h} Z \xrightarrow{cc,h} L$; and by (eff1'), $ab'.deq.top - 1 = ab.deq.top - 2$, so by (4) of Lemma 5.3 $f(ab'.deq.top - 1) =$

$\&RightHat \xrightarrow{cc,h} Z \xrightarrow{cc,h} L = \&RightHat \xrightarrow{cc',h} Z$. (3c) is preserved because no R fields are updated during the transition, and the node $\&LeftHat \xrightarrow{cc,h} Z \xrightarrow{cc,h} L$ is right-dead in the pre-state, so that $\&LeftHat \xrightarrow{cc,h} Z \xrightarrow{cc,h} L$ is right dead in the post state; moreover, $\&LeftHat \xrightarrow{cc',h} Z \xrightarrow{cc',h} L = \&LeftHat \xrightarrow{cc,h} Z \xrightarrow{cc,h} L$ because $\&LeftHat \xrightarrow{cc',h} Z = \&LeftHat \xrightarrow{cc,h} Z$ and the only L field modified during the transition belongs to $\&RightHat \xrightarrow{cc,h} Z$ which, by the injectivity of f over the appropriate range, together with (3b) of $obj_ok?$ and the fact that $ab.deq.bot + 1 \neq ab.deq.top - 1$ implies that the L field of $\&LeftHat \xrightarrow{cc,h} Z$ does not change during the transition. (3d) is preserved because (3) of Lemma 5.3 tells us that the old value of the right-hat is left-dead in the concrete post-state; we also know that $\&RightHat \xrightarrow{cc',h} Z = \&RightHat \xrightarrow{cc,h} Z \xrightarrow{cc,h} L$ by (4) of Lemma 5.3; this together with $\&RightHat \xrightarrow{cc,h} Z \xrightarrow{cc,h} L = f(ab.deq.top - 2)$ and $f(ab.deq.top - 2) \xrightarrow{cc,h} R = \&RightHat \xrightarrow{cc,h} Z$ by (2b), (2c) and (3b) of $obj_ok?$ and that no R fields are updated during the transition so that $f(ab.deq.top - 2) \xrightarrow{cc',h} R = f(ab.deq.top - 2) \xrightarrow{cc,h} R$ implies that $\&RightHat \xrightarrow{cc',h} Z$ is left-dead. $\mathcal{Q.E.D.}$

Transitions labelled by actions of the form $pop_9_yes_p$ can be shown to preserve $obj_ok?$ in much the same way as for actions of the form $pop_13_yes_p$. However, we need to know that the deque contains exactly one element. This can be shown using the following assertions about any state cc of $SnarkAut$ that has a related abstract state, when the precondition of $pop_9_yes_p$ with $side_p = Right$ is met:

1. $cc.hat_p \neq cc.hat_p \xrightarrow{cc,h} onField(cc, p)$ ($conditions_ok?$, see Section 5.5.2)
2. $cc.hat_p = otherHat_p$ (the invariant $hat_otherHat?$, see Section 5.6)

The precondition of $pop_9_yes_p$ implies that $\&RightHat \xrightarrow{cc,h} Z = cc.hat_p$. This fact, together with (1) above, means that $\&RightHat \xrightarrow{cc,h} Z$ is not right-dead in the pre-state. Once again, by (1) of $obj_ok?$ the abstract data-structure is not empty. But since the precondition of $pop_9_yes_p$ implies that $\&LeftHat \xrightarrow{cc,h} Z = cc.otherHat_p$ also, and by (2) above, we know that $\&LeftHat \xrightarrow{cc,h} Z = \&RightHat \xrightarrow{cc,h} Z$. Combining this with (3a) and (3b) of $obj_ok?$ implies that for any related abstract state ab , $ab.deq.bot = ab.deq.top - 2$.

For actions of the form $pop_b_yes_p$ (where p successfully applies a DCAS to test if the concrete data-structure is empty), we need to be able to show that the abstract data-structure is empty whenever these transitions can be taken. However, the precondition of $pop_b_yes_p$ transitions guarantees that the node $\&RightHat \xrightarrow{cc.h} Z$ is right-dead. So for any abstract state ab such that $ab.deq.bot < ab.deq.top - 1$ the predicate $dead_ok?(ab, cc, f)$ would be false.

5.8 Using PVS

The proposed simulation relation is large and complex and showing that the entire relation is preserved across all transitions of *SnarkAut* and their associated abstract transitions would require substantial effort. To reduce this effort, and provide a greater guarantee that the proofs are correct, we decided to construct proofs using the PVS theorem prover [7, 37]. PVS is a powerful theorem prover developed at SRI. Its specification language is a form of typed higher-order logic and is extremely expressive. PVS provides a good selection of proof strategies for discharging proof obligations automatically, as well as the capability to define new strategies for specialised purposes.

The PVS specification and proof files for this verification project are available from the author.

5.8.1 Describing I/O Automata

We defined PVS *theories* (the PVS term for modules defining a set of types, predicates and lemmas) to express the automata *SnarkAut* and *DeqAut*. To do this we used ideas developed by Devillers in [10] for translating I/O automata to PVS. We also expressed $SR?$ in the PVS language and defined a higher-order predicate expressing what it means for a relation to be a simulation. We then set about proving that $SR?$ is a simulation relation.

There is close association between work on I/O automata and theorem proving efforts using the Larch Prover and its associated specification language LSL [13,

5] (see also [26] for material on the Larch Prover). Using the Larch prover would have saved us some of the effort involved in translating our automata to PVS. However, Larch supports a first-order specification language and the higher-order specification language of PVS has an important property not provided by first-order languages: we did not need to construct first-order proof obligations for our simulation proof by hand. Not only could we directly express what it means for a relation to be a simulation in the higher-order logic, not possible in first-order languages, but we could directly express the existential quantification over a function, which is the critical part of $SR?$. Had we used a first-order prover, we would have needed to construct the proof obligations for each transition *by hand*: this would have been time consuming and error prone.

5.8.2 Using the PVS Prover

Some simple specialised strategies were developed to facilitate the verification attempt. For example, a strategy was developed to set up each proof obligation, determine the post-state associated with each label and name antecedent formulas so that they could be referenced directly; another was defined that could automatically verify simple invariants and some simple proof obligations related to $SR?$. However, most of the time, we relied on the inbuilt PVS strategies to provide automation.

This approach was reasonable successful: PVS was able to automatically discharge proof obligations that had been reduced, through human guidance, to properties provable by boolean and equational reasoning (and sometimes, simple quantifier instantiation). Reasoning of this kind can be the most tedious and error prone for humans, and we found the most useful aspect of PVS to be its provision of decision procedures for these sub-logics.

However, a significant amount of human input was normally required to make correct quantifier instantiations and to decide whether different parts of the simulation relation were relevant to a given proof obligation. This second kind of guidance was particularly frustrating: the time taken for PVS to discharge a provable sequent varied from hours down to a few seconds, depending on the number

of formulae in the sequent and the PVS strategies seemed to make no attempt to tell if a formula could be relevant to a proof goal.

However, it was while trying to complete a proof goal that was believed to be provable that the bug in the Snark algorithm was found (see Section 5.3), so the level of rigour required by the mechanical proof-checker did provide a positive consequence.

Chapter 6

The Bug in Snark

This chapter describes a bug in the Snark algorithm. This bug, described in 6.1, was discovered during the verification attempt presented in this thesis: as described in Section 5.5.2, certain conjuncts of the proposed simulation relation could not be shown to be preserved across transitions of *SnarkAut* and an analysis of the failed proof led to uncovering the bug. Section 6.2 attempts to characterise the class of behaviours of the Snark algorithm which cause it to malfunction and describes the properties of the Snark algorithm which allow these behaviours to occur. This analysis motivates the proposed corrections to the Snark algorithm which are presented in Chapter 7.

6.1 The Bug in the Snark Algorithm

This section describes a concrete scenario which takes a system running the Snark algorithm from a reachable state to an unsafe state. The final state of this scenario would allow the same node to be popped twice from the deque and its value to be returned twice. At the end of this section is a general description of the stages needed to reach this unsafe state, and an explanation of why the problem can occur.

Figure 6.0 represents a heap state where the deque contains two elements. In this and the following figures, the dotted box contains the set of nodes that are

part of the current representation of the deque, ie: the nodes that lie between the `LeftHat` and `RightHat`, and the left- and right-dead nodes that are just beyond the hats. This state is typical of a deque with two elements, and is clearly reachable.

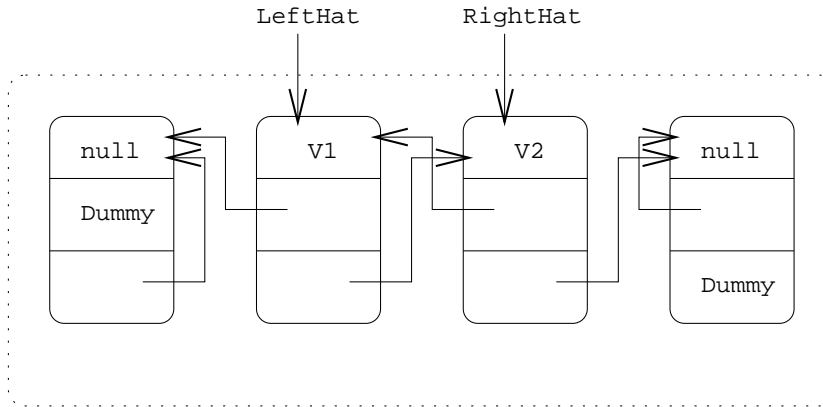


Figure 6.0: Initial state for bug sequence.

The following scenario involves three distinct processes `p1`, `p2`, and `p`. `p1` and `p2` finish in a kind of ‘race’ to pop the same node. Process `p` completes the operations required to ‘stymie’ `p1` and `p2`. In general, the role of `p` could be filled by several different processes, all distinct from `p1` and `p2`.

The stages of the concrete scenario are briefly described below; they will be described in detail later in the chapter.

1. `p1` executes the code for the `popLeft` routine until just before it can complete the DCAS at line 13. See Figure 6.1.
2. `p` executes all of the code for the `popRight` routine. See Figure 6.2
3. `p` executes all the code for the `pushLeft` routine. This temporarily falsifies the conditions necessary for `p1`’s DCAS to succeed. See Figure 6.3.
4. `p2` executes the code for the `popRight` routine until just before it can complete the DCAS at line 13. At this point, both `p1` and `p2` are attempting to pop the same node. See Figure 6.4.

5. p executes all of the code for the `popLeft` routine, popping the node that was pushed in step (2). This restores the conditions for the success of p 's DCAS. See Figure 6.5.

At the end of Step 5, both p_1 and p_2 are able to complete their pop operations *on the same node*.

Tables 6.1 and 6.2 contain the code executed by the three processes. For the sake of brevity, code that lies outside the `while` loop of the `pushLeft` procedure has been omitted (this code does not update global memory and is irrelevant to the scenario at hand). The code contained in each entry of the table is executed without interruption. Note that we have included the optional line 16 of the pop routines: because of the bug, this line can change the external behaviour of the Snark algorithm.

During Step 1, p_1 executes lines 2 to 12 of the `popLeft` routine. Note that the tests at lines 5 and 8 (`rh->R == rh` and `rh == lh`) both evaluate to false. No updates to the heap occur. Figure 6.1 represents the state of the system after this step. The labels `p1.lh` and `p1.lhR` indicate that these nodes are the values of `lh` and `lhR` in the context of p_1 . These are the values on which the DCAS at line 13 of the `popLeft` procedure depend.

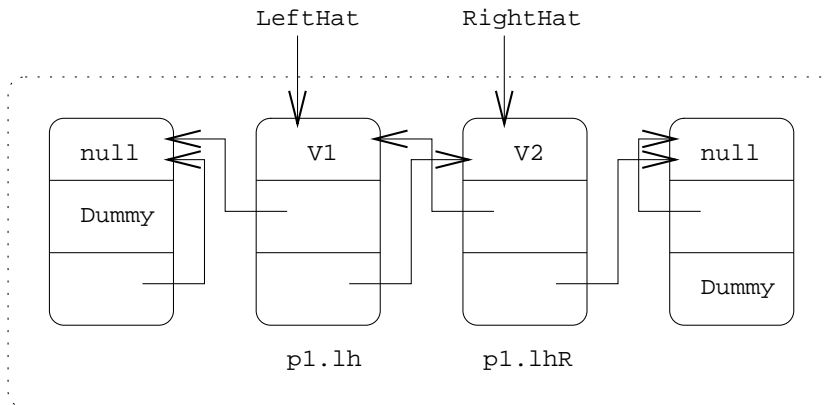


Figure 6.1: After setting up left pop.

	p1	p	p2
Step 1.	<pre>popLeft: 3. lh = LeftHat; 4. rh = RightHat; 5. if (lh->L == lh) [Test fails.] 8. else if (lh == rh) [Test fails.] 12. lhR = lh->R;</pre>	Idle	Idle
Step 2.	Idle	<pre>popRight: 3. rh = RightHat; 4. lh = LeftHat; 5. if (rh->R == rh) [Test fails.] 8. else if (rh == lh) [Test fails.] 12. rhL = rh->L; 13. if (DCAS(&RightHat, &rh->L, rh, rhL, rhL, rh)) [DCAS succeeds.] 14. result = rh->V; 15. rh->R = Dummy; 16. rh->V = null; 17. return result;</pre>	Idle
Step 3.	Waiting	<pre>pushLeft: 7. lh = LeftHat; 8. lhL = lh->L; 9. if (lhL == lh) [Test fails.] 15. nd->R = lh; 16. if (DCAS(&LeftHat, &lh->L, lh, lhL, nd, nd)) [DCAS succeeds.] 17. return ``ok``;</pre>	Idle

Table 6.1: First section of code executed.

	p1	p	p2
Step 4.	Waiting	Idle	popRight: 3. rh = RightHat; 4. lh = LeftHat; 5. if (rh->R == rh) [Test fails.] 8. else if (rh == lh) [Test fails.] 12. rhL = rh->L;
Step 5.	Waiting	popLeft: 3. lh = RightHat; 4. rh = LeftHat; 5. if (lh->L == lh) [Test fails.] 8. else if (lh == rh) [Test fails.] 12. lhR = lh->R; 13. if (DCAS(&LeftHat, &lh->R, lh, lhR, lhR, lh))[DCAS succeeds] 14. result = lh->V; 15. lh->L = Dummy; 16. lh->V = null; 17. return result;	Waiting

Table 6.2: Second section of code executed.

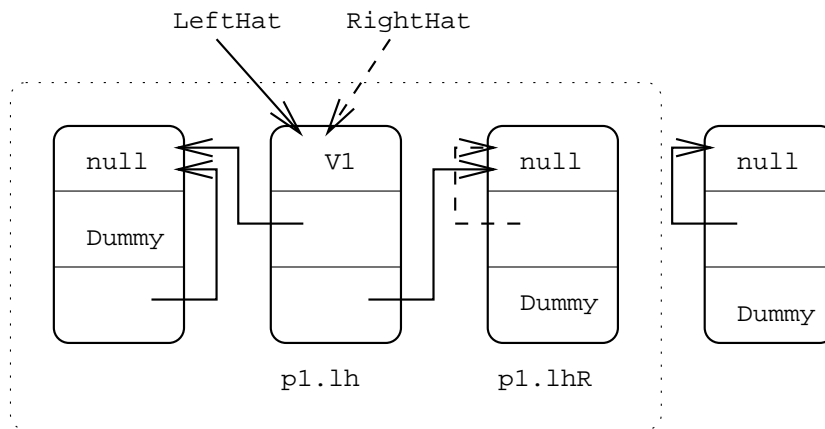


Figure 6.2: After right pop.

During Step 2, p completes a `popRight` operation. The deque now contains one element. Figure 6.2 shows the new state. The pointers that have been changed are shown with a dotted arrow. Note that the value `lhR` in $p1$'s context is still equal to `LeftHat->R`.

During Step 3, p completes a `pushLeft` operation. The deque now contains two elements. Although the value of `LeftHat` has changed, `lhR` is still equal to `lh->R` for the values in $p1$'s context. Figure 6.3 represents the new state.

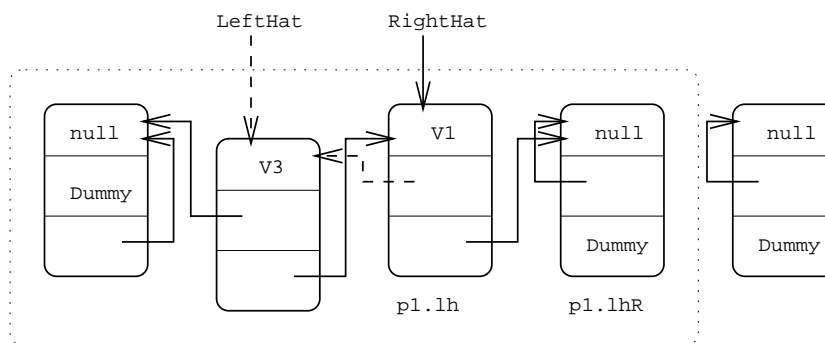


Figure 6.3: After left push.

During Step 4, p_2 executes lines 2 to 12 of `popRight`. Again, the tests at lines 5 and 8 both fail. p_2 is interrupted before it completes any updates to the

heap. Figure 6.4 indicates the values of rh and rhL in p_2 's context.

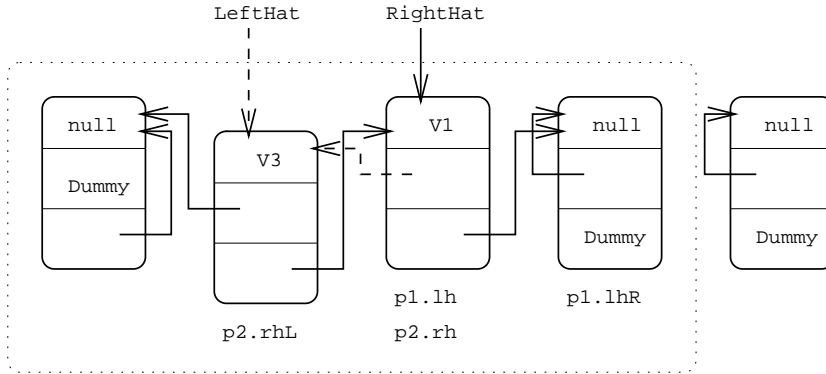


Figure 6.4: After setting up right pop.

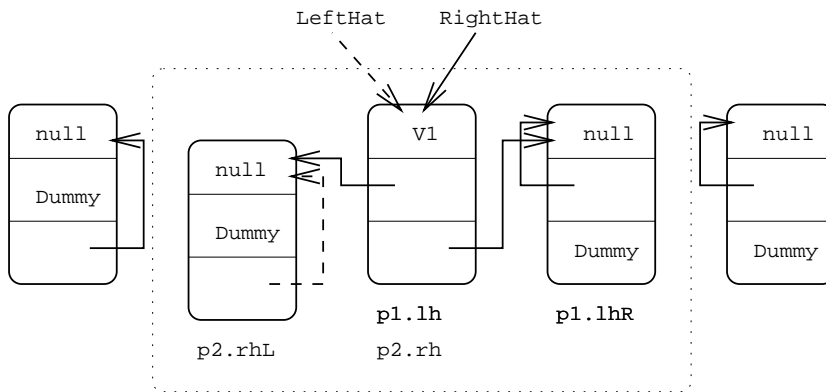


Figure 6.5: After left pop.

During Step 5, p completes a $popLeft$ operation. The state of the deque after this operation is illustrated in Figure 6.5. The deque now contains one element. The identity between $p1.lhR$ and $p1.lh \rightarrow R$ has been preserved *and* it is again the case that $LeftHat$ has the same value as $p1.lh$. So, if $p1$ were to attempt its DCAS at line 12, it would succeed. Also, if $p2$ were to attempt *its* DCAS it would succeed as well.

If Snark were to perform to specification, then since there is only one node in the deque, one of $p1$ or $p2$ should fail to return the value in that node. However,

both `p1` and `p2` are able to complete their DCASs *and neither DCAS will interfere with the other*. That is, the DCAS of `p1` will test and update the values of `LeftHat` and `LeftHat->R`, but `p2`'s DCAS will test and update the values at `RightHat` and `RightHat->L`. If both `p1` and `p2` were able to execute their respective DCASs without some other process making a change to the deque, then both would proceed through their cleanup sections and return. This would have one of two consequences: both processes could return the same value (`v1` in Figure 6.5) even though it only occurred once in the deque when it was popped; or one process could return the value and the other return `null` as if it were a real value (if, for example, `p1` executed the optional line 16, setting `p1.lh->v` to `null` before `p2` read that address).

6.2 Characterising the Bug

The scenario just described is one of many sequences of operations that can cause Snark to malfunction. The important steps are described below. At the beginning of such a sequence, the deque must contain more than one element.

1. One process sets up a pop on the deque, but is interrupted before it can execute its DCAS. For simplicity, assume that this process is popping from the *left*.
2. The deque is modified by other processes so that the node that was the `LeftHat` becomes the rightmost node.
3. The deque is further modified so that one or more nodes are pushed onto the left side of the deque.
4. Some process attempts a pop on the right. It is interrupted before it can complete its DCAS.
5. The nodes that were added to the left are removed, so that only one node remains.

There are both constraints and degrees of freedom on the order of these steps. The important thing is that both popping processes are committed to executing the DCAS at line 13 before being interrupted. This is achieved by ensuring that the tests at lines 5 and 8 fail for both processes. Hence, the constraints described in the following paragraph are needed to ensure two things: when each process executes these tests, the deque has *more than one element*; and both processes *attempt to pop the same node*. These constraints are described at the granularity of the steps enumerated above. They could be relaxed slightly, if we were to consider the interleaving of individual instructions executed by the popping processes with operations completed by the other processes. However, a very fine description of the possible sequences would be very complex without being more enlightening.

Step (1) must occur before all the other steps. Steps (2) and (3) can be interleaved. That is, the pops on the right that make the old `LeftHat` the rightmost node and the pushes on the left may occur in any order. It is important that step (2) is completed before step (4) begins. This ensures that the two conflicting processes are popping the same node. Also, step (4) must be completed after at least *one* of the nodes mentioned in step (3) has been pushed onto the left, and before step (5). This guarantees that when the process executing `popRight` makes the tests at lines 5 and 8, it will find that the deque has more than one element.

6.2.1 Approaches to Fixing the Bug

Chapter 7 presents two fixes for the Snark algorithm, each attacking the bug in the Snark algorithm from a different angle: the first allows the DCASs to detect all changes made to the deque that affect the side each process is operating on; the second increases the level of interaction between popping processes. This section shows how explanations for the bug can motivate these fixes.

The Snark algorithm has the property that substantial changes can be made to the deque that may not be detected by the DCAS instruction in the pop routines. In this respect, Snark is a victim of the *ABA problem* [31]. The ABA problem is a common phenomenon in the design of lock-free algorithms that use CAS or

DCAS. It is illustrated in the concrete scenario presented above. Immediately after step 1, p_1 's DCAS would succeed (the A condition). After step (3), p_1 's DCAS would no longer succeed (the B condition), but the change to the deque allows p_2 to set up a DCAS on the node that p_1 is about to pop. However, after step (5), p_1 's DCAS has been re-enabled (back to A), allowing two non-interfering DCASs to occur on the same node. The correction to the Snark algorithm presented in Section 7.2 attempts to solve this problem.

Another issue is the lack of interaction between processes popping from opposite ends of the deque. The execution described in Section 6.1 provides an example where two processes pop from opposite ends of the deque and do not interact *even though they are popping the same node*. This lack of interaction is generally beneficial in that it improves the parallelism of the Snark algorithm: when the deque contains many elements, two different processes may concurrently pop from opposite ends of the deque without interfering with each other. Section 7.3 attempts to solve the problem of lack of interaction without sacrificing this parallelism: roughly, after it has removed a node from the data-structure with its DCAS, a process signals to other processes that this node has been popped. This additional interaction prevents the value contained in a node being returned twice, but because the interaction occurs *after* the node has been removed from the data-structure, the parallelism of the Snark algorithm is not affected.

Chapter 7

Corrections for Snark

Having found and analysed the bug in Snark, we are now in a position to consider corrections for the algorithm. This chapter proposes two distinct fixes presented in Sections 7.2 and 7.3. The first uses a technique called *version numbering* which is often used to solve instances of the ABA problem [38, 32]. However, this technique requires that the DCAS operation be able to operate on two words that are wider than the size needed to represent a pointer. The second option uses a CAS to allow a popping process to ‘claim’ a node after it has been removed from the data-structure. As explained below, this has the disadvantage of requiring a relatively expensive CAS operation on every pop, although the kind of contention that the CAS is designed to solve would be very rare.

Both fixes incorporate two optimisations presented in Section 7.1. Both these optimisations could also have been applied to the Snark algorithm.

No formal evidence of the correctness of either fix is provided in this chapter. Enough description of the algorithms has been provided to make it intuitively plausible that they would not fall victim to the bug described in the previous chapter. Detailed arguments about the correctness of these fixes is beyond the scope of this thesis. However, issues relating to how a fully formal verification of each algorithm should proceed will be discussed in Section 7.4.

Both fixes use the same node structure as the original algorithm. This is re-

peated in Figure 7.1 for convenience.

7.1 Optimisations for the Corrections

Both fixes presented in this chapter incorporate two optimisations. The DCAS at line 6 of the pop routine (see Figure 2.8), which ensures that the deque is currently empty, can be replaced with a simpler test. The Snark algorithm (in all its forms) has the property that once the `L` or `R` field of a node is set to a self-pointer, that node will always have a self-pointer in that field. If a process popping from the left (for example) determines that `lh == lh->L` (ie: the test at line 5 succeeds), then this will be true of that node until the memory is recycled. Hence, if it is true that `lh == LeftHat` after this test, then the node pointed to by `LeftHat` is left-dead, and the deque is currently empty. So the expensive DCAS at line 6 can be replaced by a simple equality test.

The second optimisation reduces the code-length of the pop routine. The test at line 8, which attempts to tell if there is exactly one element in the deque is redundant. Firstly, the scenario described in Chapter 6 is a case where a process is ‘tricked’ into popping from the deque using the DCAS at line 13, rather than the one at line 9. Also, the authors of [8] present a case where the DCAS at line 13 is successfully executed when the concrete data-structure contains only one element, but where this operation does not disturb the invariants associated with the deque data-structure. So even in the original algorithm, as long as the deque is not empty, the line 13 DCAS always represents a valid popping operation on the deque. By virtue of these observations, lines 8-10 can be safely removed from

```

1. structure Node {
2.     Node *L;
3.     Node *R;
4.     val V;
5. }
```

Figure 7.1: Node data structure.

the pop routine. The reduction in code length would make a verification of the resulting algorithm much simpler, as it reduces the number of transitions over which the simulation relation must be shown to be preserved.

Note that both of these optimisations could have been applied to the original algorithm: neither optimisation was found until the Snark algorithm underwent the close analysis required for a fully formal verification.

7.2 Algorithm 1 - Version numbering

Recall from the previous chapter that the Snark bug is an instance of the ABA problem. The first fix uses version numbering to directly attack this problem by making it highly probable that, during a period of time sufficient for each process to complete its operation, updates to the global data-structure do not take that structure to a state that it has been in before. This is achieved by attaching a *version number* to each of the `LeftHat` and `RightHat` variables. Both the version number and the pointer are contiguous in memory and must fit into a word which is atomically updatable by a DCAS: this restriction is the source of the caveat that it is only ‘highly probable’ that the system will not return to a state which it was previously in after a given update and it is discussed in Section 7.2.1.

The structure `VersionedRef` presented in Figure 7.2 describes the pointer/version-number pair which is now the type of the hat variables. For clarity, the types of all the global and local variables are presented in Figure 7.3.

```
1. structure VersionedRef{
2.     Node *ptr;
3.     int ctr;
4. }
```

Figure 7.2: Atomically updatable reference structure.

Figures 7.4 and 7.5 contain code for the right-side operations of the version numbering solution (the left side operations are symmetric and have been omit-

```

/* Global Variables */
1. VersionedRef LeftHat, RightHat;

/* Local Variables*/
2. Node *nd, rhR, lhL;
3. VersionedRef rh, lh;

```

Figure 7.3: Variable types for version numbering fix.

ted). A dot syntax is used to represent access to one or other of the members of the `VersionedRef` structure. For example, the expression `rh.ptr` denotes the pointer value contained in `rh`. Also, angle brackets are used to construct instances of `VersionedRef`. So `<nd, rh.ctr+1>` is the `VersionedRef` value whose `ptr` is `nd` and `ctr` is `rh.ctr+1`.

Note that the DCAS on line 12 of the `pushRight` routine cannot be executed atomically. The values `<nd, rh.ctr+1>` and `<nd, lh.ctr+1>` must be constructed before the DCAS is executed. Similar comments can be made about the other DCAS operations in Figures 7.4 and 7.5. The presentation used here has been chosen for clarity: it does not affect the correctness of the algorithm because the values used to construct the arguments to the DCAS are private to each process and so will not be modified by interleaved executions of other processes.

Both the `push` and `pop` routines work in almost the same way as the original algorithm, except that they exploit the optimisations of Section 7.1. The only other differences involve managing the version numbers. When either of the hats is updated, its associated version number is simultaneously incremented. Suppose for a moment that the width of `ctr` in `VersionedRef` is unbounded so that it can represent any natural number. Recall that the bug in the Snark algorithm happens when a process popping from the right (for example) reads a value from `RightHat` and then the value of `RightHat` changes. Later the `RightHat` changes back to the value the popping process originally saw, and that process is able to complete the `pop`. In the version numbering algorithm, the `RightHat.ctr` would be greater when the popping process attempts its DCAS

```

1. rtype pushRight(val v) {
2.     nd = new Node();
3.     if (nd == null) return "full";
4.     nd->R = Dummy;
5.     nd->V = v;
6.     while(true) {
7.         rh = RightHat;
8.         rhR = rh.ptr->R;
9.         if (rhR == rh.ptr) {
10.            nd->L = Dummy;
11.            lh = LeftHat;
12.            if (DCAS(&RightHat, &LeftHat,
13.                    rh, lh, <nd,rh.ctr+1>, <nd,lh.ctr+1>))
14.                return "ok";
15.        } else {
16.            nd->L = rh.ptr;
17.            if (DCAS(&RightHat, &rh.ptr->R,
18.                    rh, rhR, <nd,rh.ctr+1>, nd))
19.                return "ok";
20.        }
21.    }

```

Figure 7.4: Snark fix, algorithm 1 - right push.

```

1.  val popRight() {
2.    while (true) {
3.      rh = RightHat;
4.      lh = LeftHat;
5.      if (rh.ptr->R == rh.ptr)
6.        if (rh.ptr == RightHat.ptr) return "empty"
7.      else {
8.        rhL = rh.ptr->L;
9.        if (DCAS(&RightHat, &rh.ptr->L,
10.              rh, rhL, <rhL.ptr, rh.ctr+1>, rh)) {
11.          result = rh.ptr->V;
12.          rh.ptr->R = Dummy;
13.          rh.ptr->V = null;
14.          return result;
15.        }
16.      }
17. }

```

Figure 7.5: Snark fix, algorithm 1 - right pop.

than when it saw the original value of `RightHat`. In this case the DCAS at line 9 of the `pop` routine would fail.

7.2.1 Limitations of Version Numbering

Unfortunately, it is clear that the `ctr` component of `VersionedRef` cannot in practice have unbounded width. Because of this, there is a possibility that the version number of one of the hats could wrap-around to zero, and then be incremented to the value that a waiting popping process saw when it first loaded the value of a hat. If this happened, it would be possible for the bug to occur. This possibility is reduced or amplified depending on the width of `ctr`. In a system that supports a 64-bit DCAS but only uses 32 bits for a pointer, 32 bits could be devoted to `ctr`. Then, for the bug to occur, the value of a hat would have to be changed *an exact multiple of 2^{32}* times between the moment when a popping process loaded the value of the hat and attempted its DCAS. Moreover, the other events which contribute to the bug would all have to occur at their appropriate times. This seems astronomically unlikely. Conversely, if the system offers a 32-bit DCAS and 32-bit pointers, it will only be possible to use a small number of bits for `ctr`.¹ In this case, the version numbering fix would be little safer than the original algorithm.

The fact that version numbers are bounded in width is a problem with all version numbering strategies [32] and the same count-width trade-off exists. In a sense this makes such algorithms unverifiable, because the problem that the version numbering was meant to solve *can always* still appear. However, the algorithm presented here would be verifiable under the assumption that the version number has unbounded width. Also, in some circumstances, the problem of bounded width version numbering can be completely overcome: Moir [32] discusses this issue and presents a tagging scheme where the ABA problem can be prevented from occurring using only bounded width tags.

¹For example, if the system insists on 16-bit word aligned addressing, the low-order bit will be 0 in every pointer. This bit could be used as a counter.

```

boolean CAS(val *addr,
            val old,
            val new) {
    atomically {
        if (*addr == old){
            *addr = new;
            return true;
        } else return false;
    }
}

```

Figure 7.6: Semantics of CAS.

7.3 Algorithm 2 - Using CAS

The second fix uses the synchronisation primitive CAS. As described in Chapter 2, CAS is just like DCAS except it operates on only one address. The semantics of CAS is presented in Figure 7.6 for convenience. Code for the `popRight` routine of this fix is presented in Figure 7.7. As before `popLeft` is symmetric with `popRight`. The push routines are exactly as in the original routine; `pushRight` is presented in Figure 7.8 for convenience.

This fix requires that there be some special value `block`, which may not be pushed into the deque: this value is used for communication between processes. In cases where `null` is never pushed into the deque, `null` may be used for `block`.

The approach of this solution is to ‘let the bug happen’. That is, the algorithm allows two different processes to pop the same node from the deque just as with the original. However, before returning the value in the popped node, each process attempts to “claim” the node: this is done in such a way that only *one* process can successfully claim the node. This is achieved by reading the value in the node; checking that it didn’t read `block` and then attempting to CAS `block` into the `v` field of the node. If this CAS succeeds, then any other process that pops the node will either see `block` when it reads the value field, or its CAS will fail when it attempts to “claim” the node. In either case, the process fails to claim the node and returns the empty value. Such a process is called a *failing* process.

```
1. val popRight()
2.   while (true) {
3.     lh = LeftHat;
4.     lhr = lh->R;
5.     if (lh->L == lh) {
6.       if (LeftHat == lh) return "empty";
7.     } else {
8.       if (DCAS (&LeftHat, &lh->R,
9.                lh, lhr, lhr, lh)) {
10.        result = lh->V;
11.        if (result != block) {
12.          if (CAS(&lh->V, result, block)) {
13.            lh->L = Dummy;
14.            return result;
15.          } else return "empty";
16.        } else return "empty";
17.      }
18.    }
```

Figure 7.7: Snark fix, Algorithm 2 - right pop.

```
1. rtype pushRight(val v) {
2.     nd = new Node();
3.     if (nd == null) return "full";
4.     nd->R = Dummy;
5.     nd->V = v;
6.     while (true) {
7.         rh = RightHat;
8.         rhR = rh->R;
9.         if (rhR == rh) {
10.            nd->L = Dummy;
11.            lh = LeftHat;
12.            if (DCAS(&RightHat, &LeftHat,
13.                    rh, lh, nd, nd))
14.                return "ok";
15.        } else {
16.            nd->L = rh;
17.            if (DCAS(&RightHat, &rh->R,
18.                    rh, rhR, nd, nd))
19.                return "ok";
20.        }
21.    }
```

Figure 7.8: Snark fix, Algorithm 2 - right push.

A formal argument that it is acceptable for failing processes to return "empty" would be based on the claim that the deque data-structure was empty at some point in time during the execution of the failing process. If this claim is true, that point in time would provide a *linearization point* at which the operation of the failing process can be thought of as taking effect (see Section 3.4). Recall the bug described in Chapter 6. It is clear that if some process observes that the `v` field of the node it just popped from the deque (ie., the node that was the target of a successful DCAS at line 8) contains `block`, either by the test at line 10, or failing the DCAS at line 11, then some *other* process has popped that node. In the bug described in Chapter 6 this can only happen if the first successful DCAS empties the deque. The process which executes the second DCAS must have started its pop operation *before* this first DCAS in order to load a pointer to this node as the hat. So, the deque is empty during the execution of *both* processes and whichever fails can return the empty value.

7.4 Modelling and Verifying the Corrections

As has already been mentioned, no formal verification of these fixes is provided in this thesis. However, it is possible to give a brief description of how these verifications could proceed. Algorithm 1 could be verified using a forward simulation, just as the attempted verification of the Snark algorithm. Algorithm 2 would be verified by showing trace inclusion between an implementation automaton and the canonical automaton for deques in the same way as discussed in Chapter 3. However, in order to show trace inclusion, we would need to use a more complex kind of simulation relation, known as *backward simulation*.

The I/O automata representing the modified algorithms would be very similar in structure to *SnarkAut* (see Section 4.2): they would possess transitions labelled by internal actions corresponding to each line of code and external actions corresponding to invocations and responses of the operations; they would possess program counter variables and local variables, indexed by processes, that would be used in the same way as in *SnarkAut*; and they would operate on a global heap

structure that would be very similar to that described in Chapter 4.

However, the automata would differ from *SnarkAut* in that they would need to have some extra capabilities:

- Algorithm 1 would be modelled by an automaton that could represent `VersionedRefs`, either by using variables from the product $\mathbb{Z} \times \text{POINTER}$, or by adding global and local variables that correspond to, for instance, `RightHat` and `rh.ctr`, that would be loaded or updated atomically at the appropriate times.
- The heap used in Algorithm 2 would need a CAS operation: this could be modelled straightforwardly in the same way as DCAS.

7.4.1 Verifying Algorithm 1

Any verification of Algorithm 1 would have to proceed under the assumption that the counters could represent any natural number. Given that assumption, Algorithm 1 could be verified using a simulation relation between an automaton representing the algorithm and *DeqAut* (see Section 4.1.2). This simulation relation would be based on a step-correspondence exactly analogous to that presented in Section 5.1: that is, transitions modelling successful pop DCASs would be matched with a *do_pop* abstract transition; transitions modelling successful push DCASs would be matched with *do_push* transitions; all other internal transitions would be matched with the empty execution fragment of *DeqAut*. This is because these DCASs still constitute a globally visible change to the sequence represented by the concrete data-structure.

The simulation relation used would be like the original: the *correspondence_ok?* predicate could be used with minor modifications to the values of the antecedent program counter variables; *obj_ok?* would need to be modified, but only to reflect the fact that the ends of the deque are accessed through `VersionedRefs`; most of *rest_ok?* could be used almost unchanged, but with one major difference.

Recall that the *conditions_ok?* predicate (see Figure 5.8) captured aspects of the executions of *SnarkAut* that were to be used to guarantee that if a DCAS would

be successful, then the concrete data-structure would be empty or nonempty as required. This predicate would be greatly simplified in a verification of Algorithm 1. Algorithm 1 has the property that as soon as the value in one of the hats is changed, every process that could successfully execute a DCAS operation with that hat as a target before the change cannot successfully execute the DCAS after the change and will never be able to successfully execute it again. This is the contribution of version numbering. We could replace conjunct (2) for instance, with the following simplification (expressed in English and pseudo-code for processes popping on the right, to avoid the need to fix mathematical notation):

If a process is about to execute the DCAS at line 9, then either

1. `rh.ptr->R != rh.ptr` and `rh.ctr == RightHat.ctr`,
- or*
2. `rh.ctr < RightHat.ctr`

Note that the strict ordering used in condition (2) precludes a successful DCAS if that condition holds. Once any update to a hat occurs, all processes that are in condition (1) will move to condition (2) and stay there. Moreover, it is trivial to show that, as the counter on `RightHat` increases, processes in condition (2) must stay there, without needing to refer to any other part of the state of the automaton.

7.4.2 Verifying Algorithm 2

Algorithm 2 has an unusual property: when a process successfully executes its DCAS at line 8 of the pop routines, it has not yet been determined which process will return the value contained in that node. However, the canonical automaton *DeqAut* decides which process will return a value as soon as that value is popped from *deq*. This means that the simple step correspondence between a successful popping DCAS and the abstract *do_pop* transition would no longer work.

There are reasons to believe we cannot find a forward simulation relation to verify Algorithm 2. Is there another option for the step correspondence of such a simulation relation? We could try matching transitions modelling a successful

CAS at line 11 of Algorithm 2 with *do_pop* transitions. However, if we did this, we would have to destroy the clear relationship between the global data-structures for related states: the concrete data-structure could contain fewer items than the abstract data-structure in related states.

The real problem is that, during a pop operation, Algorithm 2 determines the deque value represented by its globally accessible data-structure *before* it determines which processes will return values popped from that structure, whereas the canonical automaton for the deque datatype resolves both these issues simultaneously, during the *do_pop* transition.

There is another kind of simulation relation, which can be used to provide a proof technique for trace inclusion in cases like this: *backward simulations* [29, 36]. Backward simulations are similar in structure to forward simulations: the major difference is that where a forward simulation requires an abstract action and *post-state* for each concrete transition (see 3.16, clauses (2) and (3)), backwards simulations require an abstract *pre-state* and action for each concrete transition. The idea here is that we can choose an abstract state and transition, based on what we already know about a concrete post-state.

Backward simulation proofs are more complex than forward simulation proofs and splitting the effort into easier stages is a wise thing to do [36]. Recall that trace inclusion is a reflexive and transitive relation. Because of this, if we can find an *intermediate* automaton I , such that there is a backward simulation relation from I to $DeqAut$ and a forward simulation from an automaton representing Algorithm 2 to I , then we will be able to show that Algorithm 2 is correct. If I has a simple state space, then the problem of finding the backward simulation will be simplified, relative to finding a backward simulation from a complex concrete automaton to the canonical automaton.

This technique of finding an intermediate automaton that satisfies a backward simulation to the specification automaton is, in fact, complete for trace inclusion. That is, for automata A and B , if $A \leq_T B$, then there is some automaton I such there is a forward simulation from A to I and a backward simulation from I to B (see [29] for a proof).

A verification of Algorithm 2 is being undertaken as part of the work leading on from this thesis (see Section 8.4). It is less interesting to pursue a verification of Algorithm 1: as mentioned in Section 2.2, the Snark algorithm was designed to improve on a deque algorithm that required the DCAS to be able to operate on words containing a pointer and a bit. Algorithm 1 does worse than this, requiring the DCAS to operate on words wide enough to contain a pointer and a counter. In this respect, Algorithm 1 is a retrograde step.

The intermediate automaton has been designed to capture the ‘unusual’ behaviours of the Algorithm 2 automaton in an abstract way: a popping process is able to remove elements from a globally accessible sequence in the same way as *DeqAut* but when this happens, a *key* is associated with the value that is removed. A popping process is also allowed to *contend* on a key and later *claim* the key on which it was contending. After a process has claimed a key, it can return a value associated with that key by some prior popping event. This behaviour is analogous to the behaviour of Algorithm 2: processes contend on nodes by executing a DCAS on a node that has already been popped from the data-structure; processes claim the node by executing the CAS on it, thus obtaining the right to return the associated value.

The forward simulation relation from the concrete automaton to the intermediate automaton is built around the use of a representation function in the same way as the *SR?* relation of Chapter 5. It also contains conjuncts that express the analogy between behaviours of the intermediate and concrete automata indicated above.

Chapter 8

Conclusions

This chapter discusses the conclusions that can be drawn from the work presented in this thesis and some ways in which this work could be extended. Section 8.1 comments on the verification methodology presented in the thesis; Section 8.2 discusses ways in which the complexity of verifying lock-free algorithms in general, and the outcome of the attempted verification of the Snark algorithm in particular, bear on issues currently being explored in the non-blocking algorithms literature; Section 8.3 comments on the relevance of the Snark algorithm, now that it has been shown to be incorrect; and Section 8.4 discusses ongoing work in the verification of dynamic-memory non-blocking algorithms.

8.1 Evaluating the Verification Methodology

The work on which this thesis is based tackled the difficult task of verifying a concurrent algorithm that works without locks over dynamic memory. The fact that the verification attempt presented here led to the discovery that the algorithm is incorrect provides excellent evidence that research into formal verification methodologies for algorithms of this kind is worthwhile. This section discusses whether the particular verification methodologies used here are appropriate for the verification of non-blocking algorithms that work in dynamic memory.

8.1.1 I/O Automata and Simulation Relations

The technique of constructing canonical automata and showing trace inclusion using simulation relations provides a tractable proof method for showing that an algorithm correctly implements its specification. The canonical automaton captures all traces that are linearizable to executions of the implemented datatype; simulation relations reduce reasoning about global properties of automata (their executions) to reasoning about local properties (their transitions) [29].

The fact that simulation relations provide a global to local reduction is useful when dividing a proof up into manageable chunks: it is natural to show that each part of the simulation relation is preserved across the transitions labelled by each action, one action at a time. This was the approach taken in the attempted verification of the Snark algorithm and, given the size of the simulation relation, it was very useful.

8.1.2 The Representation Function

The proposed simulation relation presented in Chapter 5 is built around asserting the existence of a representation function between related states. This approach seems particularly apt for the verification of algorithms that work in dynamic memory: it provides a direct way to assert that the structure of links between nodes in the dynamic heap matches the structure of the abstract datatype, at the same time allowing us to distinguish nodes that participate in that heap structure from nodes that do not. These two properties provide solutions to two of the major difficulties in verifying dynamic-memory algorithms: dealing with structures in the heap of unbounded size; and dealing with the issue of pointer aliasing.

We could have constructed the simulation relation in a very different way, by using a *retrieve function*. What follows briefly describes how a retrieve function could have been used in the attempted verification of the Snark algorithm. This allows us to assess the merits of the representation function relative to another commonly used technique (see [39] for a discussion of retrieve functions in the context of data refinement).

A retrieve function takes as argument the concrete data-structure (in our case, the heap and the current values of the hats) and returns the abstract data-structure being represented (a member of the deque datatype). A simulation relation built on such a function would assert that, for related states, the abstract data-structure could be obtained by applying the retrieve function to the concrete data-structure. Retrieve functions offer the advantage that the required value for the abstract data-structure in the post-state of an abstract transition can be *calculated* from the value of the concrete data-structure in the post-state of the concrete transition. This property can greatly simplify verification.¹

To construct a retrieve function for a verification of the Snark algorithm, we would need to modify the deque datatype so that it used a *unique representation* for each sequence. This could be achieved by replacing the value set of the *Deque* datatype (whose elements were tuples made up of a function and two indexes, as defined in Section 4.1.1) with the set of sequences over the contained type. Such a retrieve function would recursively construct the represented sequence by traversing the concrete data-structure from one of the hats, appending the value found at each node to the sequence as it went, stopping when it reached a dead node.

We would need to construct a recursive predicate to collect all the pointers reachable by following links in the heap from one of the hats, stopping at a dead node: a pointer would satisfy this predicate just when it pointed to a node included in the concrete data-structure. Such a predicate would be used for two purposes. Firstly it would allow us to universally quantify over all the nodes in the concrete data-structure so that we could make claims about its internal structure. For example, we would need to do this to express the fact that the concrete data-structure is a doubly-linked list; in the simulation relation presented here, this was achieved by conjunct (2) of *obj_ok?* (see Figure 5.3 on page 86). Secondly, this predicate would allow us to state which nodes were not included in the list, in a manner

¹If we were using a retrieve function we could use a specialisation of forward simulation, called in [29] a *refinement*. Refinements offer the same advantages for states of automata as retrieve functions do for data-structures.

similar to the *not_in_range?* predicate (see Figure 5.4 on page 87).

The need for the recursive predicate over pointers is the principle disadvantage of the retrieve function approach. Note that, because it depends on links through the heap, the set of nodes specified by this predicate could change on every update to the heap. The representation function allows us to reason about inclusion in the concrete data-structure without mentioning reachability over the heap: all we need to mention is the set of indexes between the *top* and *bot* of the abstract data-structure. These indexes only change on particular transitions of the abstract automaton and they change in ways that are very easy to reason about.

However, as has been noted in Chapter 5, there is a drawback in the use of the representation function. Because so much of what is known about the heap is bound up in the simulation relation (in particular, the information we need to argue about aliasing and the maintenance of properties of nodes), the range of invariants that can be proved independently of that relation is very limited. This interferes with the *modularity* of the proof. It is easy to make mistakes about what needs to be known to show that certain properties are always true. If it is discovered during a proof that a proof obligation cannot be met because some other property has not been included in the simulation relation, that relation has to be *re-defined*. If this happens in a mechanical proof-checking context, great care must be taken to add conjuncts to the relation in such a way that the proofs of properties of the relation already constructed are not broken. If the simulation relation contained less information about the state of the concrete automaton, more properties of the concrete automaton could be stated and proved as invariants independently of the simulation relation.

There are probably several ways around this issue. The most obvious would be to define an intermediate automaton whose state is made up of the states of both the concrete and abstract automata *and* the representation function. Then the information contained in the representation function could be used to verify any required invariant.

8.1.3 Mechanical Proof-checking

Machine-checked proofs are the ultimate in fully rigorous demonstrations. Because of this, the construction of machine checkable proofs is almost guaranteed to lead to the discovery of bugs, if they exist (and if the user is sufficiently patient). Moreover, the automatic proof generation facilities and the strategy language of the PVS system are powerful enough to discharge simple proof obligations.

Unfortunately, there were difficulties associated with the use of the PVS system. As noted in Chapter 5, PVS needs substantial guidance to make correct decisions on the relevance of antecedent formulae and quantifier instantiation, if the proofs are to be completed in reasonable periods of time. Much more thought needs to be given to how the proof automation available can be better used.

8.2 Complexity of Verification

This section comments on issues relating to the complexity of verifying the Snark algorithm in particular and lock-free algorithms in general.

Recall that the DCAS operation is not widely implemented as a primitive instruction on multiprocessors and so the development of algorithms that use DCAS serves mainly to assess the utility of providing a primitive DCAS instruction. After presenting a long and difficult semi-formal proof of the correctness of the Snark algorithm, the authors of [8] state in their conclusions that “we are not sure that we can wholeheartedly recommend DCAS as the synchronisation primitive of choice for everyday concurrent applications programming.” They imply that the DCAS operation fails to “keep the necessary proofs of correctness as simple as possible.” If they are right to suggest that DCAS is partly to blame for the complexity of verifying the Snark algorithm, then the fact that the algorithm turned out to be incorrect, despite being the subject of a semi-formal verification, amplifies their point.

However, it should be noted that much of the complexity of lock-free algorithms comes from the need to guarantee a very strong progress property. In recent

months, a weaker non-blocking progress property has begun to receive research attention: *obstruction-freedom* [19, 20, 21]. Without going into details, obstruction freedom, like lock-freedom, precludes the use of locks in an implementation of a datatype, but allows the possibility that processes can become live-locked.

Preliminary results suggest that obstruction-free implementations are substantially simpler and easier to understand than their lock-free counterparts ([20, 21]) and that this simplicity will have implications for the effort needed to complete proofs of correctness. The design philosophy so far explored for obstruction-free algorithms focuses on their safety more than their progress, the idea being that progress properties can be provided by a *contention manager*. The contention manager is implemented separately from any data-structure and attempts to guarantee progress by preventing live-lock (as well as other performance problems caused by contention). The simplicity enabled by removing the responsibility for guaranteeing progress from the data-structure implementation constitutes a real advantage if, as this thesis suggests, the complexity of lock-free algorithms makes the required level of formality in their proofs extremely high.

8.3 The Snark Algorithm

We believe that the algorithms presented in Chapter 7 are correct implementations of concurrent dequeues. They are both closely based on the Snark algorithm and work in very similar ways. This suggests that the basic structure of the Snark algorithm, the use of the doubly-linked list and the left- or right-dead sentinels, is sound. In the case of Algorithm 2, most of the design goals of the Snark algorithm have been met, using the same basic structure. That is, the algorithm allows concurrent, non-interfering access to both ends of the deque when it contains several elements and no extra width is needed in the addresses operated on by the DCAS. So the Snark algorithm still represents a useful contribution to the development of lock-free algorithms (as well as a useful case study in their verification). The only wrinkle is that an extra CAS is required on every pop, subverting the goal that only one expensive operation would be needed per operation (the CAS operation,

like DCAS, is normally expensive).

8.4 Future Work

At the time of writing, a verification of Algorithm 2 from Chapter 7 is being undertaken. This verification uses a backward simulation, an intermediate automaton, and a forward simulation, in the manner outlined in Section 7.4.2. We believe that an intermediate automaton can be designed to abstractly model the behaviour of the concrete automaton and that this approach can greatly reduce the effort required to construct a relation between states of the automata and show that it is a simulation. The verification of Algorithm 2 provides an excellent opportunity to test this speculation.

As well as completing the verification of Algorithm 2, it would be beneficial to conduct verifications of non-blocking algorithms working in dynamic memory that implement other kinds of datatypes. This would allow us to test the hypothesis that the representation function approach used here is generalisable to other kinds of structures.

In the longer term, we need to develop techniques that make proofs of correctness more automatic and so less time consuming for the human. There are at least two complementary ways to attack this problem: one is to use a *series* of intermediate automata so that a complex simulation proof could be broken into a set of smaller proofs, each of which may be tractable for automated strategies; another is to work on developing generally useful automated proof strategies in the strategy language of PVS (or some other prover).

The first path would introduce significant amounts of human input: each intermediate automaton must be defined by a human, as must each simulation relation. Defining automata is often not time consuming, and each simulation relation could be expected to be much simpler than the one presented in this thesis. The goal would be to make up for the extra effort by avoiding the need to guide the theorem prover, as it constructs proofs. This human guidance of the proof checker was the most tedious and time consuming part of the attempted verification pre-

sented in this thesis.

We believe that the second path should be taken *after* several case studies have been completed. The goal of a particular strategy is to encapsulate a repeatedly useful piece of reasoning in executable form. We need to explore the kinds of reasoning required in the verification of dynamic-memory based concurrent algorithms *in general*, rather than building proof strategies based on limited verification experiences. However, until the goal of greater use of automation is met, we will not be fully exploiting the potential of mechanical theorem provers.

Bibliography

- [1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. Technical Report WRL-95-7, Compaq Western Research Laboratory, Pal Alto, California, September 1995.
- [2] Ole Agesen, David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Martin, Nir Shavit, and Guy L. Steele Jr. DCAS-based concurrent dequeues. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 137–146. ACM Press, 2000.
- [3] James H. Anderson and Mohamed G. Gouda. Atomic semantics of nonatomic programs. *Information Processing Letters*, 28:99–103, 1988.
- [4] N. S. Arora, B. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM Press, 1998.
- [5] Andrej Bogdanov. Formal verification of simulations between I/O automata. Master’s thesis, Massachusetts Institute of Technology, September 2001.
- [6] Manhoi Choy and Ambuj K Singh. Reasoning with non-atomic memories. Technical Report 1993-05, Department of Computer Science, University of California, Santa Barbara, August 1993.
- [7] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995.

- [8] David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Martin, Nir N. Shavit, and Guy L. Steele Jr. Even better DCAS-based concurrent dequeues. In *In Proceedings of the 14th International Conference on Distributed Computing*, pages 59–73. IEEE Computer Society Press, 2000.
- [9] David Detlefs, Paul Martin, Mark Moir, and Guy L. Steele, Jr. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, August 2001.
- [10] M. C. A. Devillers. Translating IOA automata to PVS. Preliminary Research Report CSI-R9903, February 1999.
- [11] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification*. EATCS monographs on theoretical computer science. Springer-Verlag, 1990.
- [12] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.
- [13] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. *IOA: A Language for Specifying, Programming and Validating Distributed Systems*. MIT Laboratory for Computer Science, October 2001.
- [14] Michael Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data-structures using DCAS. In *Proceedings of the twenty-first annual symposium on Principles of Distributed Computing*. ACM Press, 2002.
- [15] Michael Barry Greenwald. *Non-Blocking Synchronisation and System Design*. PhD thesis, Stanford University, August 1999.
- [16] T. Harris. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th Annual Symposium on Distributed Computing*, pages 300–314. IEEE Computer Society Press, October 2001.

- [17] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.
- [18] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Dynamic-sized lock-free data structures. In *ACM Symposium on Principles of Distributed Computing*, page 131. ACM Press, 2002.
- [19] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free software NCAS and transactional memory. Unpublished manuscript, Sun Microsystems Laboratories, Burlington, Massachusetts, 2002.
- [20] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free software transactional memory for supporting dynamic data structures. Unpublished manuscript, Sun Microsystems Laboratories, Burlington, Massachusetts, 2002.
- [21] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. To appear, International Symposium on Distributed Computing Systems, 2003.
- [22] Maurice P. Herlihy and Jeannette M. Wing. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 13–26. ACM Press, January 1987.
- [23] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, November 1990.
- [24] M.P. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structure. In *Proceedings of 16th International Symposium on Distributed Computing*, October 2002.
- [25] L Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans Computers*, 46:779–782, 1993.

- [26] LP, the Larch Prover, <http://nms.lcs.mit.edu/larch>.
- [27] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151. ACM Press, August 1987.
- [28] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.
- [29] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations – part I: untimed systems. Technical Report CS-R9313, Centrum voor Wiskunde en Informatica (CWI), 1993.
- [30] Paul A. Martin, Mark Moir, and Guy L. Steele, Jr. Better still DCAS-based concurrent dequeues. Technical Report TR-2002-111, Sun Microsystems Laboratories, 2002.
- [31] M. M. Michael and M. L. Scott. Simple, fast and practical nonblocking and blocking concurrent queue algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 267–275. ACM Press, 1996.
- [32] Mark Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing, Santa Barbara, CA.*, August 1997.
- [33] Mark Moir. Personal Communication, February 2002.
- [34] J. Antonio Ramírez-Robredo. Paired simulation of I/O automata. Master’s thesis, Massachusetts Institute of Technology, September 2000.
- [35] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3), September 1993.
- [36] Jørgen F. Søgaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anya Pogoyants. Computed-assisted simulation proofs. In

Costas Courcoubetis, editor, *Computer-Aided Verification, Fifth International Conference, CAV '93, Elounda, Greece, Lecture Notes in Computer Science 697*, pages 305–319. Springer-Verlag, 1993.

[37] The PVS Specification and Verification System, <http://pvs.csl.sri.com/>.

[38] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[39] Jim Woodcock and Jim Davies. *Using Z. Specification, Refinement and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.