On the Use of Model Checking for the Bounded and Unbounded Verification of Nonblocking Concurrent Data Structures

by

David Friggens

A thesis
submitted to Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Logic and Computation.

Victoria University of Wellington
2013

Abstract

Concurrent data structure algorithms have traditionally been designed using locks to regulate the behaviour of interacting threads, thus restricting access to parts of the shared memory to only one thread at a time. Since locks can lead to issues of performance and scalability, there has been interest in designing so-called nonblocking algorithms that do not use locks. However, designing and reasoning about concurrent systems is difficult, and is even more so for nonblocking systems, as evidenced by the number of incorrect algorithms in the literature.

This thesis explores how the technique of model checking can aid the testing and verification of nonblocking data structure algorithms. Model checking is an automated verification method for *finite state* systems, and is able to produce counterexamples when verification fails. For verification, concurrent data structures are considered to be *infinite state systems*, as there is no bound on the number of interacting threads, the number of elements in the data structure, nor the number of possible distinct data values. Thus, in order to analyse concurrent data structures with model checking, we must either place finite bounds upon them, or employ an abstraction technique that will construct a finite system with the same properties.

First, we discuss how nonblocking data structures can be best represented for model checking, and how to specify the properties we are interested in verifying. These properties are the safety property linearisability, and the progress properties wait-freedom, lock-freedom and obstruction-freedom.

Second, we investigate using model checking for exhaustive testing, by verifying bounded (and hence finite state) instances of nonblocking data structures, parameterised by the number of threads, the number of distinct data values, and the size of storage memory (e.g. array length, or maximum number of linked list nodes). It is widely held, based on anecdotal evidence, that most bugs occur in small instances. We investigate the smallest bounds needed to falsify a number of incorrect algorithms, which supports this hypothesis. We also investigate verifying a number

of correct algorithms for a range of bounds. If an algorithm can be verified for bounds significantly higher than the minimum bounds needed for falsification, then we argue it provides a high degree of confidence in the general correctness of the algorithm. However, with the available hardware we were not able to verify any of the algorithms to high enough bounds to claim such confidence.

Third, we investigate using model checking to verify nonblocking data structures by employing the technique of canonical abstraction to construct finite state representations of the unbounded algorithms. Canonical abstraction represents abstract states as 3-valued logical structures, and allows the initial coarse abstraction to be refined as necessary by adding derived predicates. We introduce several novel derived predicates and show how these allow linearisability to be verified for linked list based nonblocking stack and queue algorithms. This is achieved within the standard canonical abstraction framework, in contrast to recent approaches that have added extra abstraction techniques on top to achieve the same goal.

The finite state systems we construct using canonical abstraction are still relatively large, being exponential in the number of distinct abstract thread objects. We present an alternative application of canonical abstraction, which more coarsely collapses all threads in a state to be represented by a single abstract thread object. In addition, we define further novel derived predicates, and show that these allow linearisability to be verified for the same stack and queue algorithms far more efficiently.

Acknowledgements

I owe many thanks to my principal supervisor, Lindsay Groves. He suggested this area of research, and has provided much guidance, advice and proof reading. My initial secondary supervisor, Ray Nickson, also provided much appreciated guidance and advice.

Thanks to my examiners — Mooly Sagiv, Steve Reeves and David Pearce — for their careful reading, kind words, and helpful suggestions.

Financially, I have been supported by a VUW Postgraduate Scholarship and a VUW PhD Submission Scholarship. A grant from Sun Microsystems provided some travel funds, notably to attend the 17th International School for Computer Science Researchers in Lipari, Italy.

This thesis was almost entitled "Model Checking for Godot", so a special thank you is due to Lindsay and all the VUW administrators who have had to employ much patience. I hope you agree it was worth the wait.

I am grateful for the hospitality and understanding of the Department of Computer Science and Library at the University of Waikato, including the use of the "symphony" computing cluster.

Thanks to my parents for everything. Especially for not asking any questions about my progress in the last two years — it was hard, but it helped.

Finally, I can never thank Olivia enough for her love, encouragement and support. Ultimately, this thesis only exists because of the sacrifices she has made. Our co-publications, produced during the writing of the thesis, are extremely well cited due to her input.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis we investigate ways in which certain types of concurrent data structures can be verified using the formal technique called model checking.

We are interested in data structures within the *shared memory* concurrency paradigm, where a collection of threads communicate by reading and writing to a shared pool of memory. For correctness, we desire the algorithms to satisfy the safety property linearisability [Herlihy and Wing, 1990], and say that a data structure is *linearisable* if (roughly speaking) any execution has an equivalent execution in a given sequential specification (see Section 2.2). We also desire the algorithms to satisfy one of a number of progress properties, collectively known as *nonblocking* properties, that specify how threads can affect each other's behaviour when there is contention for shared resources.

1.1 Nonblocking Concurrency

It is easy for concurrent threads to interfere with one another, causing *race conditions*. This can be avoided by synchronising threads using locks, which permit only one thread at a time to access some or all of the shared memory. However, locks have a number of issues that can affect their efficiency and scalability. Alternatively, it is possible to design so-called *nonblocking* algorithms that do not use locks, and that tend to scale much better [see, e.g. Greenwald and Cheriton, 1996; Michael and Scott, 1998]. However, as may be expected, they are also harder to design and reason about; many papers have been published containing algorithms with bugs

[e.g. Stone, 1990, 1992; Massalin and Pu, 1991; Valois, 1994, 1995; Detlefs et al., 2000; Shann et al., 2000; Tsigas and Zhang, 2001], some with pseudo-mathematical 'proofs'.

A number of general methods have been proposed to construct nonblocking implementations from sequential [Alemany and Felten, 1992; Barnes, 1993; Herlihy, 1993; LaMarca, 1994] and lock-based [Prakash et al., 1991; Turek et al., 1992] ones but the performance of the resulting algorithms is very poor compared with the corresponding lock-based algorithms. Also, some promising efforts have been made towards refinement-based methods that produce correct implementations by construction [Abrial and Cansell, 2005; Groves and Colvin, 2006; Groves, 2008a,b; Dongol and Mooij, 2008; Dongol and Hayes, 2009].

A number of approaches have been employed for constructing deductive proofs of the correctness of nonblocking data structures using theorem provers [Doherty et al., 2004b; Gao and Hesselink, 2004; Gao et al., 2004, 2005; Colvin et al., 2005, 2006; Colvin and Groves, 2005, 2007; Derrick et al., 2007, 2008; Colvin and Dongol, 2007, 2009; Doherty and Moir, 2009; Doherty, 2010]. Verifications by theorem provers can, however, be difficult and time-consuming. Furthermore, if a proof attempt has stalled it is not always easy to tell whether it is due to a bug in the algorithm or simply inexperience or mistakes on the part of the user [Doherty, 2003].

Some work has also begun using automated decision procedures for separation logic [Reynold, 2002]. Preliminary results are promising [Vafeiadis, 2007, 2009], but the method is currently only applicable to a small range of algorithms.

1.2 Model Checking

Model checking is an automated formal verification technique that works by exploring the entire state space of a finite-state system $\mathcal{M}$, checking a specification property φ (traditionally formalised in a temporal logic). Thus, it is able to determine the satisfiability of the property in the system ($\mathcal{M} \models \varphi$), returning "yes" if it is true, and "no" otherwise, along with a counterexample — an execution trace leading from an initial state to an error state (for safety properties) or an infinite loop (for progress properties) where the property is false.

Sometimes a system under consideration cannot be directly examined by a model checker — either it has an infinite number of states, or the rep-

resentation of the finite state-space exceeds the finite memory resources available. Technological advances (if it is possible to wait for their arrival) and distributed model checking techniques [e.g. Grumberg, 2002b; Melatti et al., 2006] may somewhat alleviate the latter situation by increasing the memory resources available, but this is offset by the state explosion problem — a linear increase in the number of concurrent threads, or any other parameter, results in an exponential increase in the state space. In both situations it may be possible to construct a (smaller) finite-state abstract system that preserves the property of interest. The framework of abstract interpretation [Cousot and Cousot, 1977, 1979] enables an abstraction on states to be used to construct an abstract system that preserves a particular temporal logic.

The limitation of model checkers to finite state systems prevents them from being applied directly to general representations of concurrent data structures. These data structure algorithms typically have three unbounded parameters as:

- there is no limit on the number of threads present;
- there is no limit on the size of the data structure (length of the list, etc.); and
- data values may have an infinite (e.g. integer) type.

Thus they must be considered as infinite state systems.

1.3 Bounded Verification

One way of analysing concurrent data structures using model checking is to construct a finite model with bounds on the parameters. This approach has been used in an ad hoc way by a number of authors [e.g. Harris, 2001; Fraser, 2003; Burckhardt et al., 2006, 2007; Colvin et al., 2006; Lamport, 2006; Fraser and Harris, 2007]. However, whilst it is a good approach for finding bugs, if no bugs are found it does not necessarily give any indication of the correctness of the algorithm.

'Folk wisdom' holds that most bugs will appear in small instances of systems [see, e.g. Jackson and Damon, 1996; Marinov et al., 2003; Dolby et al., 2007; Jackson, 2012; Oetsch et al., 2012] This suggests that verifying bounded models up to a "reasonable" size would give some confidence

in the correctness of the unbounded model. Several questions arise at this point:

1. What size instances would need to be verified in order to give a reasonable level of confidence in an algorithm?
2. What size instances are needed to trigger known bugs in existing algorithms?
3. Is this approach equally applicable to both linearisability and the nonblocking properties?
4. Is this approach worth doing before, or instead of, a full verification (using a theorem prover, model checker with abstraction, etc.)?

These questions have not been comprehensively explored in the literature. In this thesis we aim to address all of the questions, by presenting a range of case studies — of both correct and incorrect algorithms — checking linearisability and nonblocking properties.

1.4 Unbounded Verification

A powerful approach for verifying large or infinite state systems with a model checker is to construct a finite state abstraction that *preserves* the property of interest, i.e. such that the property holds in the original if it holds in the abstraction. Some information in the states is made less precise, allowing each abstract state to represent (infinitely) many concrete states.

Since the verification of parametrised systems is undecidable in general [Apt and Kozen, 1986], it is inevitable that any abstraction techniques will be limited in applicability. Many such techniques have been proposed for parametrised systems (see Section 3.5), though most only consider one parameter (generally the number of threads) so are not applicable to concurrent data structures.

The more general technique of canonical abstraction [Sagiv et al., 2002] is able to be applied to concurrent data structures, as it puts a finite bound on the number of possible abstract states. Canonical abstraction represents abstract states as 3-valued logical structures over a fixed finite set of predicates. It is possible to trivially abstract any system to an abstract system consisting of a single state, which (perhaps spuriously) fails every

property. Thus, the principal problem is not whether an abstraction can be constructed, but whether we can construct an abstraction that is detailed enough to allow the specified property to be verified, but not so detailed that it is too large to be fully examined. This involves choosing the right balance of formulas to be represented by additional *instrumentation* (or *derived*) predicates, which are defined in terms of the other core predicates.

There are several questions that can be asked about canonical abstraction in this context:

1. Can canonical abstraction be used to verify linearisability for nonblocking data structures, with unbounded instances of all three parameters?

2. Can canonical abstraction be used to verify nonblocking properties for concurrent data structures, with unbounded instances of all three parameters?

3. If so, for either of the above, are the abstractions efficient (i.e. are they small enough to enable practical verification)? If not, is it possible to improve them?

In this thesis I will answer the first question in the affirmative, at least for linked list based stacks and queues. Independently, other researchers have used canonical abstraction to achieve the same result, i.e. verify linearisability for linked list based stacks and queues with unbounded lists and data values but bounded threads [Amit et al., 2007], and then for unbounded threads also [Berdine et al., 2008]. One of the principal distinctions is that these add additional layers on top of canonical abstraction to achieve the results, whilst my approach is achievable purely within canonical abstraction.

I will also address the third question, proposing a technique to more effectively abstract threads than has been done before in canonical abstraction.

The second question is left for future work.

1.5 Outline

The remainder of the thesis is divided into three parts. The first part contains background material. Chapters 2 and 3, covering nonblocking

concurrent data structures and model checking, respectively, are utilised throughout. Chapter 4, covering canonical abstraction, is utilised in Part III.

The second part investigates modelling and testing nonblocking data structures using model checking. Chapter 5 details how to

- represent nonblocking data structure algorithms,
- specify the safety property linearisability, and
- specify the nonblocking progress properties wait-, lock- and obstruction-freedom

for model checking. Chapter 6 details the results of bounded verification for a range of nonblocking data structures.

The third part investigates verifying nonblocking data structures using model checking. Chapter 7 details how linearisability of linked list based nonblocking data structures can be verified using canonical abstraction. Chapter 8 details a method for more efficiently using canonical abstraction by abstracting all of the threads together.

Finally, Chapter 9 is the conclusion, in which we directly address the questions raised in this chapter.

Part I

Background

Chapter 2

Nonblocking Data Structures

In this chapter we introduce nonblocking data structures as used throughout the thesis. Section 2.1 discusses the general representation of concurrent data structures used. Section 2.2 discusses the safety property linearisability, which relates a concurrent algorithm to a sequential specification. Section 2.3 discusses the use and problems of mutual exclusion to synchronise access to shared information. Section 2.4 introduces the progress properties wait-freedom, lock-freedom and obstruction-freedom (collectively "nonblocking properties"[1] that can apply to concurrent algorithms that do not employ mutual exclusion, whilst Section 2.5 discusses some atomic operations that can be used to achieve concurrent synchronisation with these properties. Finally, Section 2.6 describes two data structure algorithms that will be used as examples.

2.1 Concurrent Data Structures

In this thesis we consider concurrent data structure implementations, where a fixed finite collection of sequential threads concurrently perform *operations* (push, pop, enqueue, dequeue, etc.) on a data structure representation. This representation consists of a (possibly dynamic) collection of shared/global variables, which all threads can read and write to. Additionally, each thread has a collection of local variables that only it can read and write. The atomic operations that the threads perform consist

[1] Note that there are other uses of the terms "blocking" and "nonblocking" in the literature, such as for distinguishing whether transitions are defined as full or partial relations [see e.g. Smith and Derrick, 2005].

Shared: c : integer $:= 0$

```
1:  operation INC()
2:      a := c
3:      b := a + 1
4:      c := b
5:  end operation
```

```
6:  operation DEC()
7:      a := c
8:      b := a − 1
9:      c := b
10: end operation
```

Figure 2.1: Basic counter implementation

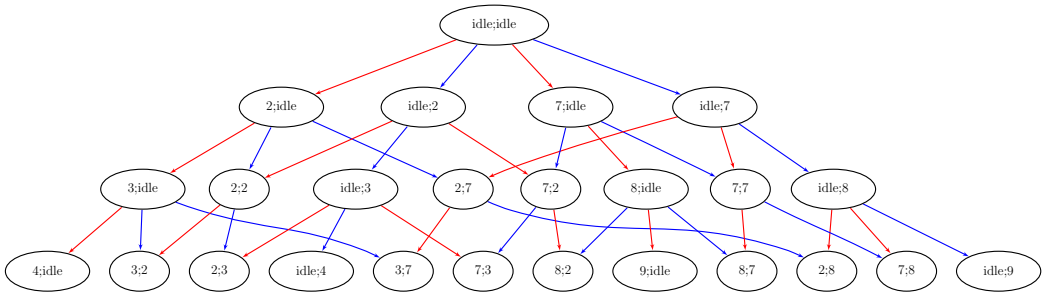


Figure 2.2: Initial part of the counter execution graph

of primitive read and write type actions (and some less primitive actions, discussed in Section 2.5). We use the *interleaving* model of concurrency, in which the concurrent system is constructed by a nondeterministic asynchronous composition of the threads. This means that every step of the system is a single step of a single thread, and at every state the next step could be performed by any of the active threads.

Example A basic counter algorithm is displayed in Figure 2.1. It has a single shared variable c (initially 0) that records the counter's value. There are two operations that increment and decrement the counter, and which utilise each thread's local variables a (to store the 'snapshot' copy of c) and b (to store the updated value).

Consider a system with two threads, each initially idle. The graph of possible executions begins (for three steps) as shown in Figure 2.2, with red indicating the first thread's steps, and blue indicating the second thread's steps.

It may seem at first glance that the implementation in Figure 2.1 is unnecessarily verbose — writing the increment for example as a single step

$$c := c + 1$$

is much more concise. However, this is generally not assumed to be an atomic step: it contains three atomic steps — a read of c, an update, and a write to c — each of which can be interleaved with the atomic steps of the other threads. ■

It is important when analysing concurrent algorithms that the atomic steps are clear so that potential executions are not missed. Some systems have atomic steps that can both read and write (see Section 2.5). Additionally, on some systems with relaxed memory models, reads and writes are not necessarily atomic [see, e.g. Adve and Gharachorloo, 1996]. In this thesis we assume that reads and writes are atomic steps.

2.2 Linearisability

The sequential approach to correctness is to consider the states of a data structure at the invocation and response of an operation to determine whether it has been applied 'correctly'. This is meaningless for concurrent data structures though, as multiple operations can be occurring at the same time. The intuitive notion of correctness that we adopt in this context is that an operation must be seen to take effect atomically at some point between its invocation and response by an outside observer with no knowledge of the internal state, and that the observed sequential behaviour is 'correct' with respect to a given sequential specification. This notion is captured by the *linearisability* property, which was introduced by Herlihy and Wing [1987, 1990].

A *history* of a concurrent system is a finite sequence of invocation and response events. We write invocations as $\text{inv}_p(\text{OP}, val^*)$ and responses as $\text{resp}_p(\text{OP}, val^*)$, where p is a process/thread name, OP is an operation, and val^* is a sequence of values that are arguments or results.[2] A response *matches* an invocation in a history if their process names agree. An invocation is *pending* if there is no subsequent matching response. For a history H, $complete(H)$ is the maximal subsequence of matching invocations and responses. A *process subhistory* H_p of a history H is the subsequence of all events for process p. Two histories H and H' are *equivalent* if for every p, $H_p = H'_p$.

[2]Herlihy and Wing also parametrise these with the name of a concurrent object (data structure). We will only discuss systems with one object, so leave this out for simplicity.

A history is *sequential* if the first event is an invocation, and (except for the last event) every invocation is immediately followed by a matching response and every response is immediately followed by an invocation. A history that is not sequential is *concurrent*. A set of sequential histories S is a *sequential specification* if it is prefix-closed, i.e. whenever $H \in S$, every prefix of H is also in S.

A history H induces an irreflexive partial order $<_H$ on operations:

$$\text{OP}_1 <_H \text{OP}_2 \quad \text{if resp}(\text{OP}_1) \text{ precedes inv}(\text{OP}_2) \text{ in } H.$$

For sequential histories, the order is total. A history H is *linearisable*, with respect to a sequential specification T, if it can be extended (by appending zero or more response events) to some history H' such that

- $complete(H')$ is equivalent to a history S in T, and
- $<_H \subseteq <_S$, i.e. the order of the non-concurrent operations in H is the same in S.

Example Figure 2.3 shows two possible executions of the counter algorithm in Figure 2.1. The execution on the left has the following trace of invocations and responses:

$$\text{inv}_1(\text{INC}), \text{inv}_2(\text{INC}), \text{resp}_1(\text{INC}, 1), \text{inv}_1(\text{DEC}), \text{resp}_1(\text{DEC}, 0), \text{resp}_2(\text{INC}, 1)$$

This can be rearranged to the following valid sequential trace, showing that the execution is linearisable:

$$\text{inv}_1(\text{INC}), \text{resp}_1(\text{INC}, 1), \text{inv}_1(\text{DEC}), \text{resp}_1(\text{DEC}, 0), \text{inv}_2(\text{INC}), \text{resp}_2(\text{INC}, 1)$$

The execution on the right has the following trace:

$$\text{inv}_1(\text{INC}), \text{inv}_2(\text{INC}), \text{resp}_2(\text{INC}, 1), \text{resp}_1(\text{INC}, 1)$$

There are two ways of sequentially rearranging this trace, both of the form

$$\text{inv}_i(\text{INC}), \text{resp}_i(\text{INC}, 1), \text{inv}_j(\text{INC}), \text{resp}_j(\text{INC}, 1)$$

neither of which is a valid counter trace. This means that the trace is not linearisable. Hence the algorithm is not a linearisable concurrent counter implementation, due to the possible interference between the threads (also known as a *race condition*). ■

Thread 1	Thread 2
$\text{inv}_1(\text{INC})$	
$a := 0$	
$b := 1$	
	$\text{inv}_2(\text{INC})$
	$a := 0$
	$b := 1$
$c := 1$	
$\text{resp}_1(\text{INC}, 1)$	
$\text{inv}_1(\text{DEC})$	
$a := 1$	
$b := 0$	
$c := 0$	
$\text{resp}_1(\text{DEC}, 0)$	
	$c := 1$
	$\text{resp}_2(\text{INC}, 1)$

Thread 1	Thread 2
$\text{inv}_1(\text{INC})$	
$a := 0$	
$b := 1$	
	$\text{inv}_2(\text{INC})$
	$a := 0$
	$b := 1$
	$c := 1$
	$\text{resp}_2(\text{INC}, 1)$
$c := 1$	
$\text{resp}_1(\text{INC}, 1)$	

Figure 2.3: Two counter executions

There are often many possible linearisations for an execution, but none is more 'correct' or 'desirable'. For some algorithms it may be possible to label a specific step in an operation as the *linearisation point*, where it is deemed to take effect. This is the approach that will be taken in this thesis (see Chapter 5), but it is not necessarily so simple — there is no requirement for an operation to be "linearised" at one of its own steps, nor for only one operation being "linearised" in a single step.

2.3 Mutual Exclusion

One of the greatest concerns with concurrent software is how to synchronize access to shared information. The algorithm in Figure 2.1 fails to be correct because two or more threads are able to read and write to the shared memory concurrently without regards to the changes made by the others.

The traditional method for dealing with this issue is to ensure *mutual exclusion* [Dijkstra, 1965] by the use of *locks* (or a similar mechanism such as *semaphores* or Java's `synchronised` keyword). Locks grant access to a particular area of memory to only one thread at a time — a thread must

```
1: operation INC( )
2:     acquire(lock)
3:     a := c
4:     b := a + 1
5:     c := b
6:     release(lock)
7: end operation
```

```
8: operation DEC( )
9:     acquire(lock)
10:    a := c
11:    b := a − 1
12:    c := b
13:    release(lock)
14: end operation
```

Figure 2.4: Counter algorithm with a lock

first acquire the lock before it enters the *critical section* that accesses the memory, and must release it afterwards.

Example Figure 2.4 shows a modified counter algorithm that uses a lock for synchronisation. In this small example, the lock enforces sequential behaviour — when one thread has acquired the lock, all competing threads must wait for it to be released — and avoids the race condition error. ■

Locks do solve the synchronization problem — each thread is guaranteed that no other will alter the shared data whilst it holds the lock — but locks can introduce additional problems [see, e.g., Herlihy and Shavit, 2008]. A *deadlock* can occur for a number of reasons, such as thread p having acquired $lock_1$ and now waiting to acquire $lock_2$, with thread q having acquired $lock_2$ and now waiting to acquire $lock_1$. Alternatively a thread could be killed before releasing a lock, thus blocking all other threads that require that lock from progressing.

Even if an algorithm is designed to avoid deadlocks, and no thread holding a lock fails,[3] mutual exclusion has some unavoidable performance issues. Because the lock owner blocks all waiting threads, it can lead to *priority inversion*, where a high priority thread is forced to wait for a low priority one, and *convoying*, where a number of fast threads are forced to closely follow a slower one through a series of locks — the overall performance being determined by the slowest thread. For these reasons mutual exclusion-based algorithms do not scale well — their efficiency decreases with the number of concurrent threads.

[3] Though in this model an aborted thread is indistinguishable from a very slow one.

2.4 Nonblocking Progress Properties

Producing non-blocking behaviour is not as simple as avoiding explicit calls to locks — there are many subtle ways of inducing *starvation*, where a thread is prevented from completing its operation. Instead, we define a hierarchy of progress properties that we might want an algorithm to satisfy. They are, in descending order: *wait-freedom*, *lock-freedom* and *obstruction-freedom*; an algorithm that satisfies any one of these is said to be *nonblocking*.[4]

2.4.1 Wait-freedom

An algorithm is wait-free if *every* thread is able to complete an operation within a finite number of its own steps. This property, due to Herlihy [1991], ensures that every thread will make progress, independent of the number and behaviour of other threads. Wait-freedom captures the ideal notion of nonblocking behaviour, but in practice wait-free algorithms are usually expensive to implement, and few algorithms have been proposed that are deemed to be of practical significance.

2.4.2 Lock-freedom

An algorithm is lock-free if *some* thread is able to complete an operation within a finite number of steps of the system. The first such algorithm was given by Lamport [1977] (for a single-writer / multi-reader shared variable) and the term "lock-free" was coined by Massalin and Pu [1991]. This property ensures the absence of deadlocks and livelocks, so the system will always make progress, independently of the number and behaviour of individual threads. In contrast to wait-freedom, it sacrifices individual guarantees of progress for a system guarantee of progress. This is less than ideal, but is often good enough in practice, as complete resource starvation is avoided — a thread is infinitely delayed only if other threads make progress infinitely often. All wait-free algorithms are also lock-free.

[4]The nomenclature used here conforms to the current general consensus in the literature. There can be some ambiguity, notably with "nonblocking" and "lock-free" being used interchangeably by some authors.

2.4.3 Obstruction-freedom

An algorithm is obstruction-free if every thread is able to complete within a finite number of steps when it is running "in isolation". This property, due to Herlihy et al. [2003], ensures the absence of deadlocks, so individual threads will be able to complete, regardless of the failure or delay of other threads. It does not rule out livelocks however, and allows conflicting threads to mutually prevent any of them from progressing. In practice, a contention manager is engaged for these situations to resolve the conflict and allow progress. The choice of contention manager employed is orthogonal to the algorithm implementation, and can even be changed on-the-fly.

Obstruction-freedom allows algorithms that are simpler to design and are more efficient in the (common) cases of low contention, compared to wait- and lock-freedom. Further, the separation of contention management from the algorithm allows different solutions to be explored without reverifying the algorithm. All lock-free algorithms are also obstruction-free.

2.5 Nonblocking Synchronization

In mutual exclusion synchronization, conflict is avoided by blocking access to resources. In nonblocking synchronization the key idea is to allow full access to resources and to detect conflict by atomically testing for changes to a value when it is updated. Primitives that can be used to implement this idea include *compare-and-swap* (CAS) and *load-linked/store-conditional* (LL/SC).

2.5.1 Compare-and-Swap

Compare-and-swap, as illustrated by the pseudocode in Figure 2.5, takes three values — an address, an old value and a new value — and returns a boolean result. If the value at *addr* is the same as *old* then it is atomically set to *new*, and true is returned; otherwise false is returned.

CAS is available on a number of modern processors, including x86, ia64 and SPARC.

Example Figure 2.6a shows the increment operation of a counter algorithm that uses CAS (the decrement is identical in structure). If there has

```
1: operation CAS(addr, old, new)
2:     atomic
3:         if addr = old then
4:             addr := new
5:             return true
6:         else
7:             return false
8:         end if
9:     end atomic
10: end operation
```

Figure 2.5: Pseudocode for Compare-and-Swap

```
1: operation INC( )
2:     repeat
3:         a := c
4:         b := a + 1
5:     until CAS(c, a, b)
6: end operation
```

(a) Using CAS

```
1: operation INC( )
2:     repeat
3:         a := LL(c)
4:         b := a + 1
5:     until SC(c, b)
6: end operation
```

(b) Using LL/SC

Figure 2.6: Nonblocking counter increment operations

been no interference from other threads then the CAS succeeds and the operation terminates. Otherwise, the CAS fails (returns false) and the operation retries.

Note that the algorithm is lock-free, but not wait-free, as a slow thread could keep retrying indefinitely whilst a faster thread repeatedly performs an update in between its read and CAS steps. ■

2.5.2 Load-Linked, Store-Conditional

Load-linked (LL) and store-conditional (SC) are a pair of operations that perform a similar function to CAS. An LL operation reads the value at an address *addr*. A subsequent SC operation atomically replaces the value at *addr* with a new one iff no SC operations have been performed on *addr* (by any thread) since the previous LL (by the current thread), returning true if successful and false otherwise.

However, it is often implemented in such a way that an SC can fail spuriously when the location has not been modified, for example if another value has been modified on the same cache line. This is known as weak LL/SC and is less than ideal. A number of authors have investigated implementing strong LL/SC operations using weak LL/SC or CAS [Moir, 1997; Anderson and Moir, 1999; Jayanti and Petrovic, 2003; Doherty et al., 2004c; Jayanti and Petrovic, 2005].

Versions of LL/SC are also available on a number of modern processors, including MIPS, Alpha, ARM and PowerPC.

Example Figure 2.6b shows the increment operation of a counter algorithm that uses LL/SC. The structure is almost identical to the CAS-based version. ■

2.5.3 ABA Problem

An SC operation is always able to determine when a value has changed, but this is not always the case for CAS operations. This allows CAS-based algorithms to suffer from the so-called "ABA Problem". Suppose a process p performs read(x) and receives the value A. Subsequently another process q performs CAS$(x, \mathrm{A}, \mathrm{B})$ and later CAS$(x, \mathrm{B}, \mathrm{A})$. Now if p performs CAS$(x, \mathrm{A}, \mathrm{Z})$ it will succeed, despite x having been modified since it was last read. In some cases, such as the example counter algorithm, this does not have any adverse effects. In other cases, particularly with memory reuse, it can be a source of errors.

One approach to avoiding the ABA problem is to attach a version number to the values that are updated with CAS operations. The version number is incremented on each modification so, for example, p reads the value $\langle \mathrm{A}, 0 \rangle$, then q performs CAS$(x, \langle \mathrm{A}, 0 \rangle, \langle \mathrm{B}, 1 \rangle)$ and CAS$(x, \langle \mathrm{B}, 1 \rangle, \langle \mathrm{A}, 2 \rangle)$. Now p's CAS$(x, \langle \mathrm{A}, 0 \rangle, \langle \mathrm{Z}, 1 \rangle)$ operation will fail. Obviously, as computers use finite number representations that wrap around at the extremities, it is still possible to encounter an ABA error. However, if the size of the data type is large enough we can be confident that it is safe enough in practice [Moir, 1997].

2.5.4 Memory Management

The allocation and deallocation of memory is a key issue in designing linked-list based nonblocking data structures. The simplest approach is to

assume the presence of a garbage collector and never explicitly free memory — memory is only freed by the system when it is no longer referenced, and newly allocated memory is always "fresh", avoiding the ABA problem for list node pointers.[5] If there is no garbage collector then there is an effective memory leak — the algorithm uses as much memory as the number of nodes added during its lifetime.

When a garbage collector cannot be utilised, it is not possible for a thread to free a node immediately after removing it — another thread may still have a pointer to it and may attempt to access and modify it after the memory has been reallocated to a different program! One solution to this problem, introduced by Michael and Scott [1996], is to put removed nodes in a "free list" or "memory pool", never freeing them to the system. When a "new" node is required, one is taken from the free list — new memory is only allocated when the free list is empty. Thus the algorithm's memory use never decreases — the amount of memory used at any particular point in time is the maximum needed up until that point. Note that the reuse of nodes may allow the ABA problem to occur for node pointers, so it may be necessary to add version numbers, as discussed above.

Alternatively, more sophisticated techniques due to Michael [2002, 2004] and Herlihy et al. [2002a,b, 2005] can be used, which allow memory to be freed to the system.

In this thesis, we will only consider linked-list based algorithms that assume the presence of a garbage collector. The implementation of a nonblocking garbage collector is an important but independent field of research [see e.g. Herlihy and Moss, 1991, 1992; Gidenstam et al., 2005, 2009].

2.6 Algorithms

Below we describe three nonblocking data structure implementations that will be used in later chapters as examples. Section 2.6.1 describes a lock-free stack algorithm, and Section 2.6.2 describes two similar lock-free queue algorithms.

Type: Node = {*val* : T; *next* : Node}
Shared: *Head* : Node := null

```
1:  operation PUSH(lv:T)
2:      n := new(Node)
3:      n.val := lv
4:      repeat
5:          ss := Head
6:          n.next := ss
7:      until CAS(Head, ss, n)
8:  end operation
```

```
9:  operation POP()
10:     repeat
11:         ss := Head
12:         if ss = null then
13:             return empty
14:         end if
15:         ssnext := ss.next
16:         lv := ss.val
17:     until CAS(Head, ss, ssnext)
18:     ss.next := null
19:     ss.val := null
20:     return lv
21: end operation
```

Figure 2.7: A lock-free stack algorithm

2.6.1 Lock-free Stack

Figure 2.7 gives the pseudocode for a linked-list based stack algorithm. Each node of the list contains a value in the *val* field and a *next* field pointing to another node. A shared *Head* variable points to the first element when the stack is non-empty, and is *null* when the stack is empty. The algorithm assumes a garbage collector is present — popped nodes are not explicitly freed. For this reason, the ABA problem cannot occur, so version numbers are not necessary.

A push operation obtains a new node n and sets its value. It then takes a "snapshot" of *Head* and points n's *next* field at the snapshot. A CAS operation is used to ensure that *Head* is updated to point to n only if it has not been modified. If *Head* has been modified then there has been a conflict with another (successful) operation so the loop is restarted.

A pop operation first takes a snapshot of *Head* and tests to see if the snapshot is *null*; if so it returns "empty". Otherwise it takes a snapshot of this node's *next* field and records the value in the *val* field. As for push, a CAS is used to detect a conflict with another successful operation —

[5]Except for null pointers — see the discussion of the queue example in Section 2.6.2 on page 23.

if *Head* has been modified it retries, otherwise it uses the snapshots to advance *Head* along the list. The fields of the popped node can be reset (lines 18–19) to increase the efficiency of garbage collection (and model checking analyses, see Section 5.1) though this is not necessary.

This algorithm was first introduced by Treiber [1986] in IBM System/370 assembly, using version numbers. Pseudocode versions are given by Michael and Scott [1998] with version numbers, and by Colvin et al. [2005] without. Versions of the algorithm have been formally verified by several authors [e.g. Colvin et al., 2005].

We can see that the algorithm is linearisable by determining the linearisation points of the operations:

- A push operation takes effect at line 7, when the CAS is successful.
- A non-empty pop operation takes effect at line 17, when the CAS is successful.
- An empty pop operation "takes effect" at line 11 when it reads a null *Head* value. The linearisation point is not at line 12 when the snapshot is tested, because *Head* may have been changed by other threads, so the stack cannot be guaranteed to be empty at that point in time.

For the first two, the successful CAS step is where the change of an added or removed node becomes observable to the other threads, and it is the trigger for leaving the loop, so cannot repeat. For the third, a *null* snapshot causes the thread to execute lines 12–13 and exit the loop (and operation), so the linearisation point cannot be repeated.

Each of these linearisation points occur exactly once for each operation. In any history of the algorithm, the invocation and response of each operation can be rearranged to the point where the operation's linearisation point occurred, resulting in a valid sequential stack history.

Regarding progress properties, both operations repeat the loop if they detect conflict with another thread. Therefore it is possible for one thread to loop indefinitely, detecting conflict each time, and never completing its operation; thus the algorithm is not wait-free. However, these conflicts occur when a thread executes a successful CAS step and leaves the loop, allowing it to complete the operation without interference. Thus the algorithm is lock-free, as it is always the case that *some* thread will complete its operation — a thread is only delayed by repeating its loop if another thread exits its loop.

Type: Node = {*val* : T; *next* : Node}
Shared: *Head* : Node := **new**(Node)
Shared: *Tail* : Node := *Head*

```
 1: operation ENQUEUE(lv:T)
 2:     n := new(Node)
 3:     n.val := lv
 4:     loop
 5:         sstail := Tail
 6:         ssnext := sstail.next
 7:         if sstail = Tail then
 8:             if ssnext = null then
 9:                 if CAS(sstail.next, ssnext, n) then
10:                     break
11:                 end if
12:             else
13:                 CAS(Tail, sstail, ssnext)
14:             end if
15:         end if
16:     end loop
17:     CAS(Tail, sstail, n)
18: end operation
```

(a) Initialisation and enqueue operation

Figure 2.8: A lock-free linked list based queue

2.6.2 Lock-free Unbounded Queue

Figure 2.8 gives the pseudocode for (two versions of) a linked list based queue algorithm, which is similar in form to the stack algorithm and also assumes the presence of a garbage collector. This algorithm has shared *Head* and *Tail* variables, and both are initialised to a new dummy node; a queue of n elements is represented with a list of $n + 1$ nodes. *Head* always points to the dummy node at the beginning of the list and its *next* field points to the node that holds the first element in the queue; thus the queue is empty iff $Head.next = \text{null}$. An enqueue operation contains two distinct atomic updates — "add a new node to the end of the list" and "make *Tail* point to the new node". Because of this, *Tail* does not necessarily point to the final node in the list — it may lag one node behind and point to the

penultimate node. To avoid a mid-enqueue thread blocking other threads from proceeding, the step of updating *Tail* may be performed by another thread if it detects that the queue is in such a state.

An enqueue operation (see Figure 2.8a) begins by obtaining a new node *n* and setting its value. It then begins a loop by reading a "snapshot" first of *Tail* and then of the (snapshot's) *next* field. It checks whether *Tail* has changed (if so then restarts the loop) and then tests whether the second snapshot is null. If *ssnext* is null then it is (or was) the end of the list, so it attempts to append *n* to the list with a CAS; if this fails then the loop is restarted, otherwise it leaves the loop and attempts to make *Tail* point to *n* with a CAS. Alternatively, if *ssnext* is not null then there is another enqueue operation that is in between the CAS updates and has not updated *Tail* yet; thus it attempts to update *Tail* itself with a CAS before restarting the loop. For both of the CAS steps that update *Tail* the success or failure of the CAS does not need to be checked — the CAS will only fail if another thread has already performed the update, and it only needs to be performed once.

Figures 2.8b and c show two different dequeue operations. The first one begins by reading "snapshots" of *Head* and *Tail* and the *Head* snapshot's *next* field. It checks whether *Head* has changed (if so restarts the loop) and then checks whether (the snapshots of) *Head* and *Tail* are the same. If *Head* and *Tail* are the same then either the queue is empty — if *ssnext* is null — or another thread is mid-way through enqueuing the first element in a singleton queue; the dequeue operation attempts to help by updating *Tail* with a CAS before restarting the loop. Alternatively, if *Head* and *Tail* are not the same then the queue is (or was) non-empty, so a dequeue can be attempted. It reads the value of the first non-dummy node (*ssnext*) and then attempts to increment *Head* with a CAS. If the CAS succeeds then it returns the value, otherwise it restarts the loop.

As with the stack algorithm it is possible, but not necessary, for garbage collection and model checking efficiency to reset the value of the new dummy/*Head* node (line 33). It is not possible however, to reset the *next* field of the dequeued node. Doing so allows an ABA error in an enqueuing thread — the updating CAS is only able to check that *Tail*'s *next* field remains unchanged (i.e. is null) and not whether *Tail* itself has changed. If *Tail* has changed and the original node has been dequeued, then setting its *next* field to null allows the enqueue operation to erroneously append *n* to a node outside the list.

The second dequeue operation (see Figure 2.8c) is similar, but does not

```
19: operation DEQUEUE()
20:     loop
21:         sshead := Head
22:         sstail := Tail
23:         ssnext := sshead.next
24:         if sshead = Head then
25:             if sshead = sstail then
26:                 if ssnext = null then
27:                     return empty
28:                 end if
29:                 CAS(Tail, sstail, ssnext)
30:             else
31:                 lv := ssnext.val
32:                 if CAS(Head, sshead, ssnext) then
33:                     ssnext.val := null
34:                     return lv
35:                 end if
36:             end if
37:         end if
38:     end loop
39: end operation
```

(b) Original dequeue operation

Figure 2.8: A lock-free linked list based queue

read a snapshot of *Tail* at the beginning. Detecting an empty queue and incrementing *Head* proceed otherwise the same, and it is only after the successful CAS to increment *Head* that *Tail* is read and potentially incremented after detecting an unfinished singleton enqueue.

The second dequeue operation appears to be more efficient, as *Tail* is not read unnecessarily. It is only read at the end of a non-empty dequeue — in contrast, the first dequeue operation reads it at least once every time the loop is executed. This difference in behaviour also leads to a difference in the structures the lists can take. Because the second dequeue operation increments *Head* before checking whether there is a singleton queue with *Tail* lagging behind, it allows *Head* and *Tail* to "cross over", i.e. $Tail.next = Head$. This behaviour is not possible with the first dequeue operation, as it always increments the lagging *Tail* before incrementing *Head*.

This algorithm, with the first dequeue operation, was first introduced

```
40: operation DEQUEUE()
41:     loop
42:         sshead := Head
43:         ssnext := sshead.next
44:         if sshead = Head then
45:             if ssnext = null then
46:                 return empty
47:             else
48:                 lv := ssnext.val
49:                 if CAS(Head, sshead, ssnext) then
50:                     ssnext.val := null
51:                     sstail := Tail
52:                     if sshead = sstail then
53:                         CAS(Tail, sstail, ssnext)
54:                     end if
55:                     break
56:                 end if
57:             end if
58:         end if
59:     end loop
60:     return lv
61: end operation
```

(c) Simplified dequeue operation

Figure 2.8: A lock-free linked list based queue

by Michael and Scott [1996, 1998] using version numbers and a "free list" of nodes that never returns memory to the system. The algorithm was first verified by Doherty et al. [2004b], who introduced the second dequeue operation and assumed the presence of a garbage collector as we do here.

We can see that the algorithm is linearisable by determining the linearisation points of the operations. For the operations that make changes visible to other threads the linearisation points are relatively straightforward:

- For enqueue it is the CAS step at line 9 that adds the new node to the end of the list.
- For non-empty dequeues it is the CAS step at lines 32 and 49 that increment *Head* along the list.

For empty dequeues it is when the snapshot of *sshead.next* is read at lines 23 and 43 when entering the loop for the final time. It is not at lines 26 and 45 when *ssnext* is determined to be null — by this time other threads may have modified the queue and it may not be empty anymore. For this reason it is not possible to determine when the snapshot is taken whether it will be the linearisation point for an empty dequeue or not — even if the queue is empty, other threads may enqueue and dequeue nodes so that *Head* is changed and the operation restarts the loop at lines 24 and 44.

For the same reasons as for the stack algorithm, this queue algorithm is not wait-free but is lock-free. One thread can be infinitely delayed by detecting conflict with other threads and restarting the loop, but the conflicting behaviour is a thread executing a CAS step (one of the linearisation points) and exiting its loop, so it is always the case that some thread will complete its operation within a finite number of steps.

Chapter 3

Model Checking

Model checking is an automated formal verification technique commonly used for reactive systems [see e.g. Clarke et al., 1999; Bérard et al., 2001; Clarke and Schlingloff, 2001; Holzmann, 2004; Baier and Katoen, 2008; Grumberg and Veith, 2008]. Traditionally, *temporal logic model checking* — developed independently by Clarke and Emerson [1981] in the USA and Queille and Sifakis [1982] in France — is a method that investigates whether a structure $\mathcal{M}$, representing a system *is a model* of a temporal logic formula φ, representing a specification of the system, in other words that $\mathcal{M}$ satisfies φ, written $\mathcal{M} \models \varphi$. However, the term "model checking" has become broader to encompass other state-space exploration methods that "check" a "model" of a system for errors, not necessarily involving temporal logic. For example, the Spin model checker (see Section 3.6.1) provides options such as an assertion statement as a means of checking properties, as well as a temporal logic.

All model checkers work by exploring the entire state space of the system to verify the desired property — whether that be the satisfaction of a temporal logic formula, the affirmation of all embedded assertions, the absence of deadlock states, some other property, or a combination.

This chapter begins by introducing the formalisms used for systems (Section 3.1) and specifications (Section 3.2). Then we describe the three major paradigms: explicit state model checking (Section 3.3.1), BDD-based symbolic model checking (Section 3.3.2), and bounded model checking (Section 3.3.3).

In Section 3.4, we discuss methods commonly employed for reducing the size of the state space explored, to tackle the state explosion problem. In Section 3.5, we discuss the abstraction of systems for model checking

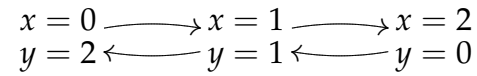


Figure 3.1: Example Kripke structure

in general terms. Finally, in Section 3.6, we discuss a number of model checking tools.

3.1 Systems

Systems are described abstractly by graph structures, with nodes representing states and arcs representing transitions. Plain directed graphs are not expressive enough, so they are extended with annotations providing more specific information. The most popular formalisms are *Kripke structures*, which annotate the nodes with atomic propositions, and *labelled transition systems*, which annotate the arcs with actions. The two formalisms are equivalent, with translations from each to the other [De Nicola and Vaandrager, 1990; Reniers and Willemse, 2011].

Kripke structures were introduced by Kripke [1959, 1963] as a model theory for modal logics, of which temporal logics are a subset [Goldblatt, 1992]. A Kripke structure is defined over a set AP of atomic propositions by a tuple $\langle S, Init, R, L \rangle$, where S is a set of *states*, $Init \subseteq S$ is a set of *initial states*, $R \subseteq S \times S$ is a *transition relation*, and $L : S \to 2^A$ is a *labelling function*, or *interpretation*, that assigns all the propositions that are true at each state. For model checking purposes, both S and AP are usually finite, and R is often required to be *total*, i.e. every state has at least one successor. Figure 3.1 [from Müller-Olm et al., 1999] shows an example Kripke structure where the propositions are of the form $var = num$ and represents two components, x and y, trading two resources back and forth.

In this chapter, we use Kripke structures to define the semantics of specification logics. In later chapters we specify systems using the modelling languages of the model checking tools used; these can be translated to Kripke structures.

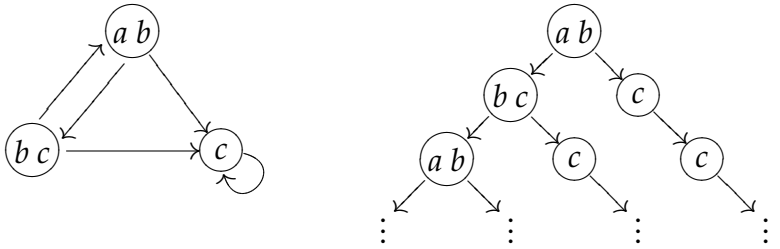


Figure 3.2: A Kripke structure and its computation tree

3.2 Specifications

Specification properties of the system are usually formalized in a temporal logic, expressing properties of states at different points in time (where time is relative to transitions). Temporal logics originate from the work of the philosopher Arthur Prior [1957, 1967], and their use as a specification language was first suggested by Amir Pnueli [1977, 1981]. They are a form of modal logic and have been extensively studied [Emerson, 1990; Goldblatt, 1992; Stirling, 1992; Gabbay et al., 1994]. Many logics have been used for model checking specifications, but the two most popular are propositional *linear temporal logic* ([P]LTL) and *computation tree logic* (CTL). The first is of the *linear-time* paradigm, where only linear executions of future states are considered, and the second is of the *branching-time* paradigm, where all possible future executions are considered at each state. First we define the logic CTL*, which extends both.[1]

3.2.1 CTL*

CTL* is a powerful logic for describing properties of *computation trees*, i.e. the tree of possible executions of a system. Figure 3.2 shows a simple Kripke structure and part of its matching computation tree. In addition to the standard propositional logic connectives, CTL* formulas contain *path quantifiers* and *temporal operators*. There are two path quantifiers, which refer to the paths that can possibly be taken from a state, **A** ("for all paths") and **E** ("for some path"). There are five temporal operators (three unary

[1] For an overview of the development of these three logics see the historical surveys of Goldblatt [2003, 2006, §7.3] and Vardi [2008a,b].

and two binary), which refer to the states in a path:

- **X** ("next") requires a property to hold in the second state of the path;
- **F** ("eventually" or "in the future") requires a property to hold in *some* state of the path;
- **G** ("always" or "globally") requires a property to hold in *every* state of the path;
- **U** ("until") is a binary operator that requires the first property to hold in every state until the second holds;
- **R** ("release") is the logical dual of **U** and requires the second property to hold in every state up to and including one where the first holds — i.e. when it is "released" by the first, which may never happen.

There are two types of formulas in CTL* — *state formulas*, which are true or false for a specific state, and *path formulas*, which are true or false for a specific path. The syntax of the logic is given by the following rules:

- If $p \in \mathsf{AP}$, then p is a state formula.
- If φ and ψ are state formulas, then $\neg\ \varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$ and $\varphi \rightarrow \psi$ are state formulas.
- If φ is a path formula, then $\mathbf{A}\varphi$ and $\mathbf{E}\varphi$ are state formulas.
- If φ is a state formula, then φ is a path formula also.
- If φ and ψ are path formulas, then $\neg\ \varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$, $\mathbf{X}\varphi$, $\mathbf{F}\varphi$, $\mathbf{G}\varphi$, $\varphi\ \mathbf{U}\ \psi$ and $\varphi\ \mathbf{R}\ \psi$ are path formulas.

CTL* is the set of *state formulas* defined by these rules.

The semantics of CTL* are defined with respect to a Kripke structure $\langle S, Init, R, L\rangle$ where R is total. A *path* in $\mathcal{M}$ is a sequence of states $\pi = s_0, s_1, s_2, \ldots$ such that for every $i \geqslant 0$, $\langle s_i, s_{i+1}\rangle \in R$. This corresponds to an infinite branch in the computation tree of the Kripke structure. The i-th *suffix* of a path, π^i, is the path that starts from the i-th state of π. For a state formula φ, the notation $\mathcal{M}, s \models \varphi$ means that φ holds at state s in $\mathcal{M}$. For a path formula φ, the notation $\mathcal{M}, \pi \models \varphi$ means that φ holds along the

path π in $\mathcal{M}$. If $p \in \mathsf{AP}$, φ and φ' are state formulas, and ψ and ψ' are path formulas, then the relation $\models$ is defined inductively as follows:

$$
\begin{array}{lll}
\mathcal{M}, s \models p & \text{iff} & p \in L(s) \\
\mathcal{M}, s \models \neg\, \varphi & \text{iff} & \mathcal{M}, s \not\models \varphi \\
\mathcal{M}, s \models \varphi \wedge \varphi' & \text{iff} & \mathcal{M}, s \models \varphi \text{ and } \mathcal{M}, s \models \varphi' \\
\mathcal{M}, s \models \mathbf{E}\psi & \text{iff} & \exists\, \pi = s, \ldots \bullet \mathcal{M}, \pi \models \psi \\
\mathcal{M}, s \models \mathbf{A}\psi & \text{iff} & \forall\, \pi = s, \ldots \bullet \mathcal{M}, \pi \models \psi \\
\mathcal{M}, \pi \models \varphi & \text{iff} & \mathcal{M}, s \models \varphi \text{ where } \pi = s, \ldots \\
\mathcal{M}, \pi \models \neg\, \psi & \text{iff} & \mathcal{M}, \pi \not\models \psi \\
\mathcal{M}, \pi \models \psi \wedge \psi' & \text{iff} & \mathcal{M}, \pi \models \psi \text{ and } \mathcal{M}, \pi \models \psi' \\
\mathcal{M}, \pi \models \mathbf{X}\psi & \text{iff} & \mathcal{M}, \pi^1 \models \psi \\
\mathcal{M}, \pi \models \mathbf{F}\psi & \text{iff} & \exists\, i \geqslant 0 \bullet \mathcal{M}, \pi^i \models \psi \\
\mathcal{M}, \pi \models \mathbf{G}\psi & \text{iff} & \forall\, j \geqslant 0 \bullet \mathcal{M}, \pi^j \models \psi \\
\mathcal{M}, \pi \models \psi\, \mathbf{U}\, \psi' & \text{iff} & \exists\, k \geqslant 0 \bullet \mathcal{M}, \pi^k \models \psi' \text{ and } \forall\, 0 \leqslant j < k \bullet \mathcal{M}, \pi^j \models \psi \\
\mathcal{M}, \pi \models \psi\, \mathbf{R}\, \psi' & \text{iff} & \forall\, j \geqslant 0 \bullet \text{ if } \mathcal{M}, \pi^i \not\models \psi \text{ for every } i < j \\
 & & \qquad \text{then } \mathcal{M}, \pi^j \models \psi'
\end{array}
$$

The semantics of $\vee$ and $\rightarrow$ are given by their standard definitions:

$$
\begin{array}{rcl}
\varphi \vee \psi & \equiv & \neg\, (\neg\, \varphi \wedge \neg\, \psi) \\
\varphi \rightarrow \psi & \equiv & \neg\, \varphi \vee \psi
\end{array}
$$

Similarly, it would have been possible to define **R**, **F**, **G** and **A** in terms of $\neg$, **U** and **E**:

$$
\begin{array}{rcl}
\varphi\, \mathbf{R}\, \psi & \equiv & \neg\, (\neg\, \varphi\, \mathbf{U}\, \neg\, \psi) \\
\mathbf{F}\varphi & \equiv & \text{true}\, \mathbf{U}\, \varphi \\
\mathbf{G}\varphi & \equiv & \neg\, \mathbf{F}\, \neg\, \varphi \\
\mathbf{A}\varphi & \equiv & \neg\, \mathbf{E}\, \neg\, \varphi
\end{array}
$$

We say that $\mathcal{M}$ *is a model* of φ, $\mathcal{M} \models \varphi$, iff $\mathcal{M}, s_0 \models \varphi$ for every $s_0 \in \mathit{Init}$.

It is occasionally useful to refer only to universal or existential formulas. We define

$$
\begin{array}{l}
\text{ACTL*} = \{\varphi \mid \varphi \text{ does not contain } \mathbf{E} \text{ and is in NNF}\} \\
\text{ECTL*} = \{\varphi \mid \varphi \text{ does not contain } \mathbf{A} \text{ and is in NNF}\}
\end{array}
$$

where a formula is in negation normal form (NNF) iff all negation operators occur only before atomic propositions. NNF is required to ensure that there are no implicit path quantifiers of the wrong kind, as $\neg\, \mathbf{A}\phi = \mathbf{E}\, \neg\, \phi$ etc.

We are now able to define the logics LTL and CTL as subsets of CTL*.

3.2.2 LTL

Linear temporal logic (LTL) is a logic for expressing properties that hold for every execution; for this reason it is a useful logic for describing properties of a system's behaviour. Time is treated as if each state has a unique possible future. As such, there is only a single quantifier in each formula — a universal quantifier at the beginning; in practice it is usually not written explicitly.

The logic is obtained by restricting CTL* to formulas of the form $\mathbf{A}\varphi$, where φ is a path formula not containing any path quantifiers; thus it expresses a subset of the properties expressed by ACTL*. The syntax can be given in Backus normal form (BNF) as

$$\begin{array}{lll} \text{LTL} & ::= & \mathbf{A}\phi \\ \phi & ::= & \mathsf{AP} \mid \neg\,\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \\ & & \mathbf{X}\,\phi \mid \mathbf{F}\,\phi \mid \mathbf{G}\,\phi \mid \phi\,\mathbf{U}\,\phi \mid \phi\,\mathbf{R}\,\phi \end{array}$$

3.2.3 CTL

Computation tree logic (CTL) is a logic for expressing properties of a set of executions; for this reason it is a useful logic for describing properties of a system's structure. Time is treated such that every possible future from each state can be considered.

The logic is obtained by restricting CTL* so that every temporal operator is immediately preceded by a path quantifier (and vice versa). The syntax can be given in BNF as

$$\begin{array}{lll} \phi & ::= & \mathsf{AP} \mid \neg\,\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \\ & & \mathbf{AX}\,\phi \mid \mathbf{EX}\,\phi \mid \mathbf{AF}\,\phi \mid \mathbf{EF}\,\phi \mid \mathbf{AG}\,\phi \mid \mathbf{EG}\,\phi \mid \\ & & \mathbf{A}[\phi\,\mathbf{U}\,\phi] \mid \mathbf{E}[\phi\,\mathbf{U}\,\phi] \mid \mathbf{A}[\phi\,\mathbf{R}\,\phi] \mid \mathbf{E}[\phi\,\mathbf{R}\,\phi] \end{array}$$

3.2.4 Linear- versus Branching-time

There has been a significant amount of discussion in the literature over the past several decades about the choice between CTL and LTL [e.g. Pnueli, 1977; Lamport, 1980; Emerson and Clarke, 1980; Ben-Ari et al., 1981, 1983; Emerson and Halpern, 1983, 1986; Pnueli, 1985; Emerson and Lei, 1985, 1987; Clarke and Draghicescu, 1988; Vardi, 1998a,b, 2001; Nain and Vardi,

2007].[2] There are two main points of discussion — complexity and expressibility.

For a system of size n and formula of size m, a CTL model checking algorithm will run in time $O(nm)$ [Clarke et al., 1983, 1986], whilst an LTL model checking algorithm will run in time $n2^{O(m)}$ [Lichtenstein and Pnueli, 1985].[3] This indicates that in the worst case, LTL model checking is exponentially more expensive than CTL. This worst case behaviour occurs infrequently in practice though, and most properties that are checked are relatively small. As such, there is generally no notable difference in performance between such model checkers. (Indeed, in some cases LTL model checkers outperform CTL ones on the same problem [see e.g. Beaudenon et al., 2010].)

It can be readily shown that LTL and CTL are expressively incomparable — they express differing (though overlapping) sets of properties. CTL can describe so-called 'reset' properties, which express that at every state there is at least one possible execution that returns to an initial state. For example, if p represents a property of an initial state, reset can be expressed by the CTL formula $\mathbf{AG}(\mathbf{EF}\,p)$, which has no equivalent in LTL. On the other hand, LTL is able to describe fairness properties expressing that certain properties hold (or do not hold) infinitely often. For example, the LTL formula $\mathbf{AFG}\,q$ requires q to hold in every state from some point in the future, and has no equivalent in CTL.

In this thesis I prefer LTL when possible, as a more "natural" logic of the two for reasoning about the behaviour of algorithms. However, in Section 5.3 I define properties that can only be expressed in one of LTL or CTL and thus use that particular logic for the definition.

3.3 Model Checking Paradigms

There are a great number of different algorithms for deciding the model checking problem for temporal logics. As model checkers are fully automatic, a user can treat them as a 'black box' and use them successfully without understanding the underlying algorithm. It is useful however, to have a general understanding of the underlying approach. There are three main paradigms of model checking algorithms — explicit state

[2] Appendix B of Holzmann's Spin book [Holzmann, 2004] contains a brief and balanced summary of this "debate".

[3] See Schnoebelen [2003] for an overview of model checking complexity.

model checking based on automata-theoretic methods, BDD-based symbolic model checking, and SAT/SMT-based bounded model checking.

3.3.1 Explicit State Model Checking

The most straightforward approach for state space exploration is to perform a depth- or breadth-first search, explicitly storing each visited state in memory. Temporal logic properties can be checked using automata-based methods, first proposed by Vardi and Wolper [1986], to exploit the fact that the logic LTL expresses a subset of the ω-regular properties, to produce an on-the-fly model checking algorithm.[4] The ω-regular properties are those that are accepted by ω-automata, i.e. finite automata over infinite words such as Büchi automata [Perrin and Pin, 2004].

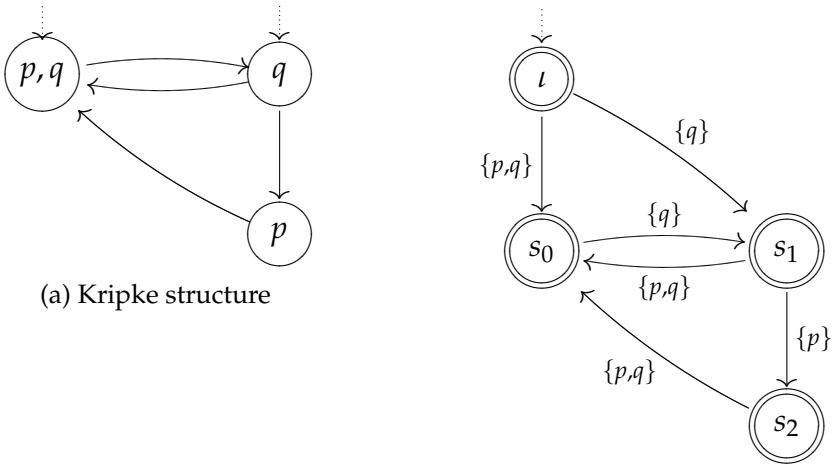
A finite automaton $\mathcal{A}$ over finite words is defined by a tuple $\langle \Sigma, Q, \Delta, Q_0, F \rangle$, where Σ is an *alphabet* (a finite set of symbols), Q is a finite set of *states*, $\Delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*, $Q_0 \subseteq Q$ is a set of *initial states*, and $F \subseteq Q$ is a set of *final* or *accepting states*. If $v \in \Sigma^*$ is a word of length $|v|$, then a *run* of $\mathcal{A}$ on v is a mapping $\rho : \{0, 1, \ldots, |v|\} \mapsto Q$ such that $p(0) \in Q_0$ and, for all $0 \leqslant i < |v|$, $\langle \rho(i), v(i), \rho(i+1) \rangle \in \Delta$. A run is *accepting* if $\rho(|v|) \in F$, i.e. it ends at an accepting state, and an automaton $\mathcal{A}$ *accepts* a word v if, and only if, there exists an accepting run of $\mathcal{A}$ on v. The *language* of $\mathcal{A}$, $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$, consists of all the words accepted by $\mathcal{A}$ [Clarke et al., 1999].

A Büchi automaton has the same components as a finite automaton; a run of a Büchi automaton $\mathcal{A}$ on an infinite word $v \in \Sigma^\omega$ is defined in almost the same way, except that $|v| = \omega$. A run is *accepting* if at least one of the accepting states occurs infinitely often. Büchi automata are closed under intersection and complementation [Büchi, 1962].

A Kripke structure $\mathcal{M} = \langle S, Init, R, L \rangle$ can be represented by a Büchi automaton $\mathcal{A}_\mathcal{M} = \langle \Sigma, S \cup \{\iota\}, \Delta, \{\iota\}, S \cup \{\iota\} \rangle$ where

- $\Sigma = 2^{\mathsf{AP}}$,
- $\langle s, \alpha, s' \rangle \in \Delta$ iff $\langle s, s' \rangle \in R$ and $\alpha = L(s')$, and
- $\langle \iota, \alpha, s \rangle \in \Delta$ iff $s \in Init$ and $\alpha = L(s)$.

[4] There has also been research on automata theoretic methods for branching time logics [Bernholtz et al., 1994; Kupferman et al., 2000].



(a) Kripke structure

(b) Büchi automaton

Figure 3.3: Two equivalent system representations

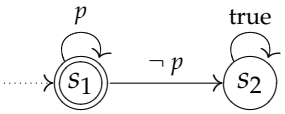


Figure 3.4: A Büchi automaton for **AG**p

Figure 3.3 shows a simple Kripke structure and its corresponding Büchi automaton [from Clarke et al., 1999, p. 123]. There also exist algorithms for translating any LTL formula into a Büchi automaton [e.g. Gerth et al., 1995; Gastin and Oddoux, 2001; Giannakopoulou and Lerda, 2002]. Figure 3.4 shows a Büchi automaton that corresponds to the invariant formula $\mathbf{AG}p$. The model checking problem $\mathcal{M} \models \varphi$ can be solved by the language inclusion problem $\mathcal{L}(\mathcal{A}_\mathcal{M}) \subseteq \mathcal{L}(\mathcal{A}_\varphi)$, which is equivalent to checking whether $\mathcal{L}(\mathcal{A}_\mathcal{M}) \cap \overline{\mathcal{L}(\mathcal{A}_\varphi)} = \varnothing$. Complementing Büchi automaton is a complicated procedure, so it is common to construct $\mathcal{A}_{\neg\varphi}$ instead of $\overline{\mathcal{A}_\varphi}$ [Clarke et al., 1999].

It is possible to construct the intersection automaton that accepts the language $\mathcal{L}(\mathcal{A}_\mathcal{M}) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi})$ directly, without first completely constructing the two component automata separately. This results in "on-the-fly" verification, as the emptiness of the language is checked as the automata is constructed. If the language is found to be non-empty then a counterexample falsifying the specification has been discovered and there is no need to construct the rest of the automaton. Thus, this approach has the advantage that finding bugs in even very large systems can be relatively inexpensive in many cases. Additionally, if the automaton is too large to be constructed with available resources, and no counterexample has been encountered, then the property has at least been verified for some portion of the statespace. The coverage can be increased by repeating the partial verification with different strategies for constructing the automaton so that different parts of the statespace are explored each time [Holzmann et al., 2008].

3.3.2 BDD-Based Symbolic Model Checking

Explicit state model checking stores every reachable state in memory — as a result the memory use increases linearly with the number of states that are encountered. Alternative approaches instead store a symbolic representation of the visited states; the symbolic representation can use much less memory, leading to a reduction in overall memory use. Symbolic model checking was first proposed by McMillan et al. [Burch et al., 1990, 1992; McMillan, 1992], exploiting the compact representation that can often be gained from the use of ordered binary decision diagrams (OBDDs).

A *binary decision tree* is a rooted, directed tree used for representing all possible truth assignments of a boolean formula, akin to a truth table. The

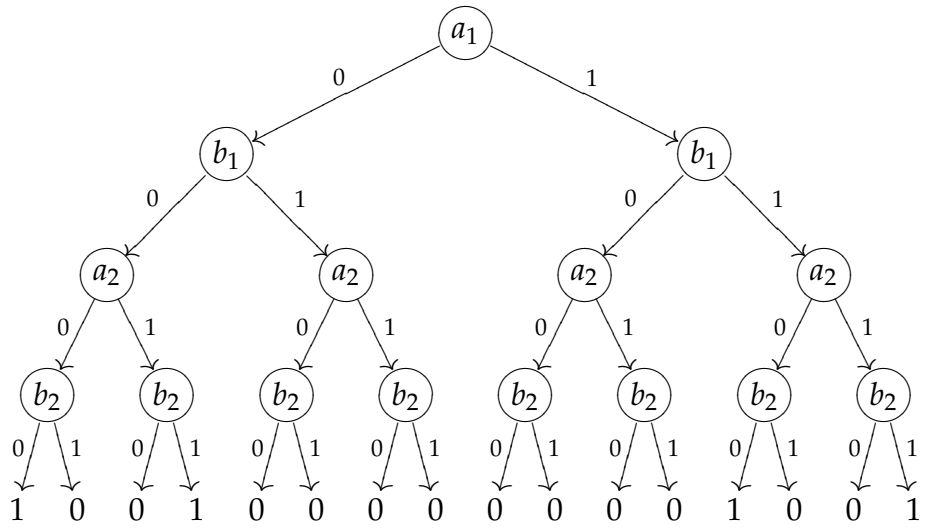


Figure 3.5: A binary decision tree for a two-bit comparator

tree's non-terminal nodes are labelled with a boolean variable and have a *low* and *high* successor (corresponding to the assignment of false or true to the variable, respectively), and the terminal nodes are labelled with either 0 or 1 (for false and true). The truth of a particular assignment to the formula's variables can be ascertained by following the appropriate path through to the tree — it is true if the terminal node is 1. Figure 3.5 contains a binary decision tree for a two-bit comparator given by the formula $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$.

A *binary decision diagram* (BDD) [Bryant, 1986] is a modification of a binary decision tree in a more compact form by merging duplicate terminal nodes and isomorphic subgraphs, resulting in a rooted, directed acyclic graph. Additionally, an *ordered binary decision diagram* (OBDD) requires the variables to occur in a specific order, providing a canonical representation for boolean formulas. The size of an OBDD can depend critically on the variable ordering — Figure 3.6 shows two OBDDs for the two-bit comparator example; each has a different variable ordering, resulting in a different number of nodes. Finding an optimal ordering is intractable, as even checking that a given ordering is optimal is an NP-complete problem, though several heuristics exist for finding good orderings [Bryant, 1992].

State space exploration is achieved by constructing OBDD representa-

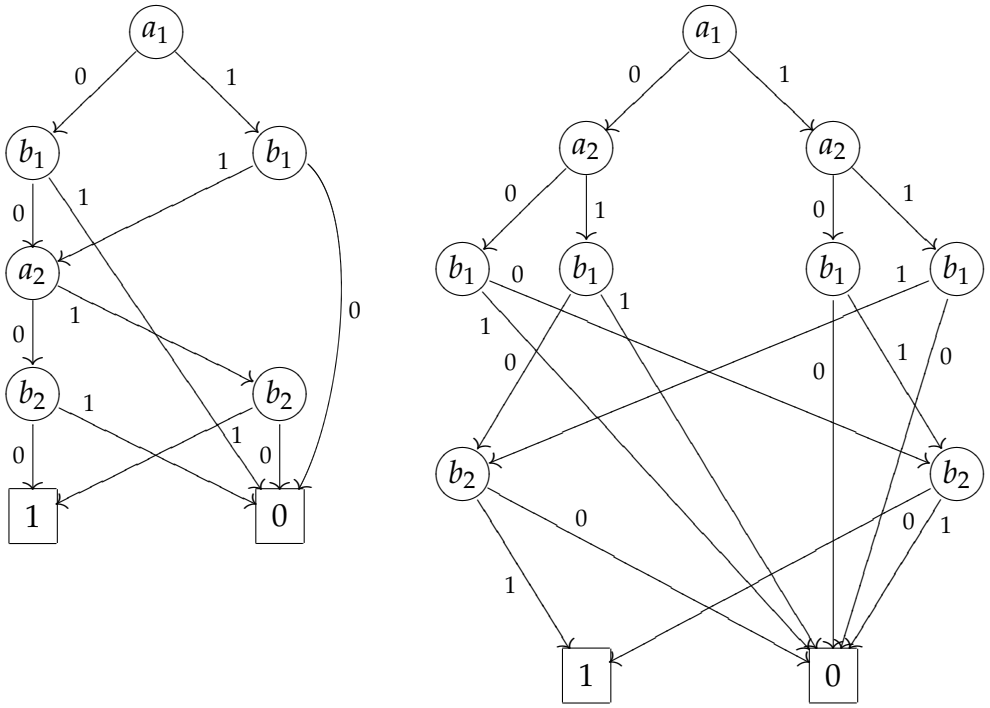


Figure 3.6: OBDDs for a two-bit comparator

tions of the set of initial states and the transition relation. They are composed to produce another OBDD representation of states, and the transition relation is repeatedly composed until a fixpoint is reached, indicating that all reachable states have been found. This approach is most commonly used for checking CTL, though it can be applied to LTL as well [Bérard et al., 2001].

3.3.3 Bounded Model Checking

Another approach to LTL symbolic model checking, introduced by Biere et al. [1999a,b], uses efficient satisfiability (SAT) decision procedures to search for counterexamples up to a bounded length. A boolean formula is constructed in polynomial time that is satisfied iff there is a counterexample of a given length k or less. For example, a formula expressing a counterexample of length three to a safety property $\mathbf{AG}\varphi$ has the form

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge T(s_2, s_3) \wedge \\ (\neg\, \varphi(s_0) \vee \neg\, \varphi(s_1) \vee \neg\, \varphi(s_2) \vee \neg\, \varphi(s_3))$$

where $I(s_i)$ asserts that s_i is an initial state, $T(s_i, s_j)$ asserts that s_i and s_j are related by the transition relation, and $\neg\, \varphi(s_i)$ asserts that φ is false at s_i.

Bounded model checking is complete for finite-state systems as it is sufficient to check up to the *diameter* of the system — the minimal path length that can reach every state from the initial state. However, computing the diameter is not a trivial problem — SAT is NP-complete and LTL model checking is PSPACE-complete [Sistla and Clarke, 1985], so if it could be found in polynomial time then NP = PSPACE. For this reason, bounded model checking is generally used as a fast method for finding counterexamples, rather than for verification.

An extension to this approach is to use a satisfiability modulo theories (SMT) solver [de Moura et al., 2007] instead of a SAT solver, which enables bounded model checking of certain infinite state systems, such as those using real numbers [de Moura et al., 2002].

3.4 Tackling the State Explosion Problem

One of the most significant issues in model checking is the so-called "state explosion problem" [Clarke and Grumberg, 1987]. The size of the state

space of a system is exponential in the number of threads (and other parameters) and is thus a major factor in the size of models that can be verified with current computational resources.[5] Much work has gone into investigating ways of reducing the state space to counteract the state explosion problem. In this section mention bitstate hashing (3.4.1), partial order reduction (3.4.2), slicing (3.4.3) and symmetry reduction (3.4.4).

3.4.1 Bitstate hashing

Bitstate hashing, due to Holzmann [1987, 1995, 1998], is a technique that greatly increases the efficiency of explicit state model checking, though at the cost of the guarantee of complete statespace coverage.

Most explicit state model checking tools store visited states in a hashtable. Each element of the hashtable contains a list of states; when a new state is visited, its hash value is computed and it is inserted into the list if not already present.

An alternative approach is to simply have a boolean value at each element in the hashtable, which is initialised to false and set to true when a state with that hash value is visited. This induces an equivalence relation on states based on their hash values. The benefit is that memory requirements are drastically reduced: each equivalence class of states only requires one bit of storage in total, rather than tens, hundreds or even thousands for each state. The downside is that for each equivalence class only the successors of the first visited state will be explored — every subsequently encountered state in the class will be treated as previously visited. The states in each equivalence class have no semantic relation, thus arbitrary parts of the statespace may not be analysed.

Statespace coverage can be approximated by the ratio of visited states to the size of the hashtable (collisions are more likely as the table gets fuller); confidence can be improved by rerunning the analysis with different hash functions.

3.4.2 Partial order reduction

In most concurrent systems there are many places within the statespace where a number of 'independent' transitions performed by different threads

[5] Demri et al. [2002, 2006] investigate the parametric complexity [Downey and Fellows, 1999] of model checking due to the state explosion problem.

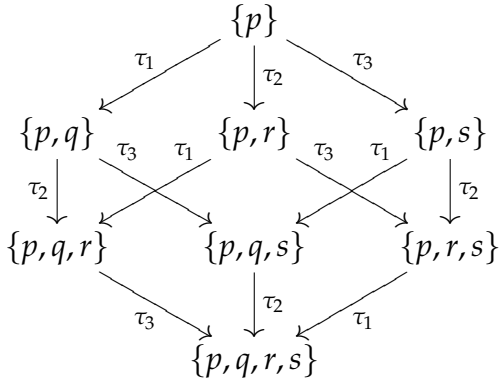


Figure 3.7: Independent transition execution

are enabled at the same time. The interleaving concurrency model means that all permutations of execution orderings will be explored in the statespace, though it may only be necessary to consider a single one in order to verify a property.

By 'independent', we mean that the transition sequences have the same result, no matter which order the transitions are executed in. For example, consider the execution fragment in Figure 3.7 of three independent transitions, where the states consist of sets of independent propositional variables. The transitions arrive at the same state, no matter which order they are executed in, and 12 transitions, with 6 intermediate states, must be considered. If the ordering of the transitions does not affect the property being checked, which here is the case for $\mathbf{AG}p$ but is not for $\mathbf{AG}q$, then it is possible to reduce the statespace by considering only one interleaving (say, $\tau_1\tau_2\tau_3$), so only 3 transitions and 2 intermediate states need to be considered.

A reduced state graph, where only one from each set of equivalent transition sequences is present, can be automatically computed using the technique of partial order reduction [Peled, 1994, 1996; Holzmann and Peled, 1995; Godefroid, 1996]. The reduction can be computed on the fly, for both explicit state and symbolic model checkers, whilst guaranteeing that the property is valid in the original system iff it is valid in the reduced system. Clarke et al. [1999, Chapter 10] provide a solid technical overview.

3.4.3 Slicing

Slicing, or cone of influence reduction, is a technique with wide applications throughout computer science [Tip, 1995; Xu et al., 2005], and has been applied successfully in model checking [e.g. Millett and Teitelbaum, 2000; Vasudevan et al., 2005]. Slicing focusses on the variables in the property being checked, by removing the variables from the system that do not influence the property variables. This can result in a smaller statespace to verify.

3.4.4 Symmetry reduction

In many concurrent systems there are symmetries that can be exploited to reduce the size of the models to be analysed, and there have been many investigations on how this can be achieved [e.g. Clarke et al., 1993, 1996; Emerson and Sistla, 1993, 1996; Ip and Dill, 1993, 1996; Emerson and Trefler, 1999; Bošnački et al., 2000, 2002; Sistla et al., 2000; Iosif, 2001, 2002, 2004; Emerson and Wahl, 2003; Donaldson et al., 2005a,b; Sistla, 2004; Sistla and Godefroid, 2004; Barner and Grumberg, 2005; Donaldson and Miller, 2005, 2006].

The underlying observation is that the symmetry implies the existence of nontrivial permutation groups (Clarke et al. [1999, Chapter 14] provide a general overview), which can be used to define an equivalence relation on the statespace. Then a quotient model can be produced that has only one state in each equivalence class; this model is smaller than the original but they are bisimilar, so they have exactly the same CTL* properties.

Constructing this model is not easy though — determining whether two states are in the same equivalence class, or orbit, is as hard as the graph isomorphism problem, for which no polynomial time algorithm is known [Clarke et al., 1996].

Certain symmetries can be identified by using the *scalarset* datatype introduced by Ip and Dill [1993, 1996]. A scalarset is a finite unordered set with restricted operations — notably that values can only be compared for equality. Any two states that differ only by a permutation of scalarset values are in the same equivalence class; the model checking procedure is the same except that a canonical permutation function is used when comparing the current state with the previously visited states.

3.5 Abstraction

The techniques for addressing the state explosion problem described in the previous section are all inherently aimed at finite state systems — they can make a finite statespace smaller, but they cannot (generally) turn an infinite statespace into a finite one. An alternative, and largely orthogonal, approach is to apply some form of abstraction.

An abstraction involves "forgetting" some part of each state so that (often infinitely) many states are indistinguishable. Instead of asking whether a (*concrete*) model $\mathcal{M}$ satisfies a (concrete) specification φ, "$\mathcal{M} \models \varphi$?", we construct an abstract model $\mathcal{M}_A$ and an abstract specification φ_A and ask "$\mathcal{M}_A \models \varphi_A$?" Abstract models are constructed from techniques that map individual concrete states to abstract states. An abstract specification is constructed by applying the same state abstraction technique to the state variables in the concrete specification.

There are many abstraction techniques in the literature, including:

- Data abstraction [Clarke et al., 1994], where individual data types are mapped to abstract versions, e.g. from integers to the 3-element set $\{\text{negative}, \text{zero}, \text{positive}\}$.
- Counter abstraction [Pnueli et al., 2002], where only one copy of each thread configuration (based on the location and variable assignments) is stored, along with a "0-1-∞" counter to record how many copies there are.
- Predicate abstraction [Graf and Saïdi, 1997], where abstract states consist of boolean variables, each recording a property of the concrete state.
- Canonical abstraction [Sagiv et al., 2002], where abstract states are 3-valued logical structures, as described in Chapter 4.

Ideally, an abstraction will *strongly preserve* properties, i.e.

$$\mathcal{M}_A \models \varphi_A \Leftrightarrow \mathcal{M} \models \varphi$$

and the result (whether verification or refutation) is guaranteed to hold for the original concrete system. It is usually necessary though to use abstractions that only *weakly preserve* properties, i.e.

$$\mathcal{M}_A \models \varphi_A \Rightarrow \mathcal{M} \models \varphi$$

In these cases, if the abstract property is shown to hold in the abstract model then the concrete property is guaranteed to hold in the concrete model. If the abstract property does not hold in the abstract model then the concrete property may or may not hold in the concrete model. However, the model checker will provide a counterexample, which can be examined against the concrete model to determine whether the error is genuine or spurious. If the counterexample is spurious then it may be possible to revise the abstraction and try again. This step ofter requires manual intervention, but techniques exist to automate it in some contexts [Clarke et al., 2000, 2003].

Properties of the logic ACTL* (recall from Section 3.2.1) can be weakly preserved by constructing an abstract model that is an *over-approximation* of the concrete model, i.e. that it contains more behaviours than the concrete model. In contrast, properties of the logic ECTL* can be weakly preserved by constructing an abstract model that is an *under-approximation* of the concrete model, i.e. that it contains fewer behaviours than the concrete model.

3.5.1 Constructing Abstract Models

Given an abstraction technique that can be used to construct abstract states, it is not clear how the abstract transition relation should be constructed, nor how properties are preserved. These issues can be dealt with by using the framework of *abstract interpretation*, which uses a Galois connection to relate the concrete and abstract states. Abstract interpretation was originally developed for compiler optimization [Cousot and Cousot, 1977, 1979], and has been extended to handle properties of computations as well as properties of states [Loiseaux et al., 1995; Dams et al., 1997]. We will give a brief overview of the construction of abstract systems that weakly preserve the logics ACTL* (and hence LTL), ECTL* and CTL*. For more detail see e.g. Grumberg's survey [2002a].

We first revise the definition of Kripke structures from Section 3.1 so that states are labelled, not by atomic propositions, but by a set of *literals*, $Lit = \mathsf{AP} \cup \{\neg\, p \mid p \in \mathsf{AP}\}$. The Kripke structures are the same, except that the labelling function $L : S \to 2^{Lit}$ is required to satisfy

$$p \in L(s) \Rightarrow \neg\, p \notin L(s) \text{ and } \neg\, p \in L(s) \Rightarrow p \notin L(s)$$

Note that it is possible that neither p nor $\neg\, p$ at s.

A *Galois connection* from a partially ordered set $\langle C, \leqslant_C \rangle$ to another $\langle A, \leqslant_A \rangle$ is a pair of functions $\alpha : C \to A$ (the abstraction function) and $\gamma : A \to C$ (the concretisation function) such that

- α and γ are total and monotonic,
- for all $c \in C$, $\gamma(\alpha(c)) \geqslant_C c$, and
- for all $a \in A$, $\alpha(\gamma(a)) \leqslant_A a$.

Additionally, if $\leqslant_A$ is defined by $\leqslant_C$ such that

$$a \leqslant_A a' \quad \text{iff} \quad \gamma(a) \leqslant_C \gamma(a')$$

then $\langle \alpha, \gamma \rangle$ is a *Galois insertion*.

For a given Kripke structure $\mathcal{M}$, we use $\langle 2^S, \subseteq \rangle$ as the concrete domain. A set of abstract states, $\widehat{S}$, is chosen and we use the Galois insertion so that the partial order on $\widehat{S}$ is determined by $a \leqslant a'$ iff $\gamma(a) \subseteq \gamma(a')$. To construct the abstract Kripke structure, $\widehat{\mathcal{M}}$, the set of initial states is defined as

$$\widehat{S_0} = \{\alpha(\{s\}) \mid s \in S_0\}$$

and the labelling function is defined as

$$p \in \widehat{L}(a) \quad \text{iff} \quad \forall\, s \in \gamma(a) \bullet p \in L(s)$$

The transition relation is not so straightforward. First we define two relations that will be used. For any A and B and relation $R \subseteq A \times B$, the relations $R^{\forall\exists}, R^{\exists\exists} \subseteq 2^A \times 2^B$ are defined as

$$R^{\forall\exists} = \{\langle X, Y \rangle \mid \forall\, x \in X \bullet \exists\, y \in Y \bullet x\, R\, y\}$$
$$R^{\exists\exists} = \{\langle X, Y \rangle \mid \exists\, x \in X \bullet \exists\, y \in Y \bullet x\, R\, y\}$$

These relations can be used to define abstract systems that preserve different properties. The Kripke structure $\left\langle \widehat{S}, \widehat{S_0}, \widehat{R^E}, \widehat{L} \right\rangle$ preserves the logic ACTL* and the Kripke structure $\left\langle \widehat{S}, \widehat{S_0}, \widehat{R^A}, \widehat{L} \right\rangle$ preserves the logic ECTL*, where the transition relations are defined as

$$a\, \widehat{R^E}\, b \quad \text{iff} \quad b \in \left\{ \alpha(Y) \,\middle|\, Y \in \min \left\{ Y' \mid \gamma(a)\, R^{\forall\exists}\, Y' \right\} \right\}$$
$$a\, \widehat{R^A}\, b \quad \text{iff} \quad b \in \left\{ \alpha(Y) \,\middle|\, Y \in \min \left\{ Y' \mid \gamma(a)\, R^{\exists\exists}\, Y' \right\} \right\}$$

In order to construct an abstract system that preserves the full logic CTL* we define a Kripke structure with a *mixed transition relation*. The system is defined as $\widehat{\mathcal{M}} = \left\langle \widehat{S}, \widehat{S}_0, \widehat{R^A}, \widehat{R^E}, \widehat{L} \right\rangle$ and has two types of paths — A-paths, defined along $\widehat{R^A}$ transitions, and E-paths, defined along $\widehat{R^E}$ transitions. Finally, we modify the semantics of CTL* from Section 3.2.1 so that

$$\begin{array}{lll} \widehat{\mathcal{M}}, s \models \mathbf{E}\psi & \text{iff} & \widehat{\mathcal{M}}, \pi \models \psi \text{ for some E-path } \pi \\ \widehat{\mathcal{M}}, s \models \mathbf{A}\psi & \text{iff} & \widehat{\mathcal{M}}, \pi \models \psi \text{ for every A-path } \pi \end{array}$$

3.6 Tools

In this section we describe three model checking tools — Spin (3.6.1), SAL (3.6.2) and TVLA (3.6.3).

3.6.1 Spin

Spin[6] [Holzmann, 1997, 2004], developed by Gerard Holzmann at Bell Labs, is an explicit-state on-the-fly model checker that uses automata-theoretic methods. Its input language Promela[7] is procedural, with visual similarity with C, and is largely based on Dijkstra's guarded command language. It can check ω-automata properties, including translations of LTL formulas, and allows embedded assertions.

Developed since 1989, Spin is a stable and widely used program with active development and support, and was the winner of the 2001 ACM System Software Award. Spin is written in C and runs on Unix and Windows; its source code is freely available for noncommercial use.

Spin supports a number of statespace reduction techniques, including partial order reduction, state compression (to reduce memory requirements at the expense of time), and bitstate hashing. Recent advanced features include embedded C statements to aid model checking software, and support for utilising multiple processors [Holzmann and Bošnački, 2007].

Spin has no provision for symmetry reduction. An extension based on scalarsets, SymmSpin, was developed [Bošnački et al., 2000, 2002], but is no longer available. Another extension, dSpin (dynamic Spin) [Demartini

[6] `http://www.spinroot.com`

[7] Promela was originally an acronym for PROcess MEta LAnguage; Spin was an acronym of Simple Promela INterpreter.

et al., 1999], was developed that allowed dynamic creation and deletion of heap objects, and implemented garbage collection [Iosif and Sisto, 2000] and symmetry reduction [Iosif, 2001, 2002, 2004]. dSpin was based on version 3.2.0 (from April 1998) and the code it produces does not compile with recent versions of `gcc`. The extension TopSpin [Donaldson and Miller, 2006] automatically infers symmetry using a separate group theory tool; this extension is compatible with recent versions of Spin and `gcc` though it has restrictions on the Promela constructs that can be used.

Spin is used for the analyses in Chapter 6.

3.6.2 SAL

SAL,[8] the Symbolic Analysis Laboratory from SRI International, comprises a declarative language [de Moura et al., 2001] and toolset [de Moura et al., 2004]. The tools include a deadlock checker, BDD-based symbolic model checkers for both LTL and CTL, and both SAT- and SMT-based bounded model checkers.

SAL 3.0 was released under the GPL licence in December 2006. It is written in Scheme and runs on Unix systems (including Windows in the Cygwin environment).

SAL was used for preliminary work, which is discussed at the start of Chapter 6.

3.6.3 TVLA/3VMC

TVLA[9] (Three Valued Logic Analyzer) [Lev-Ami and Sagiv, 2000; Lev-Ami, 2000; Lev-Ami et al., 2004; Bogudlov et al., 2007a] is a prototype static analysis tool developed at Tel Aviv University, implementing canonical abstraction (see Chapter 4). It includes the extension 3VMC (3 Valued Model Checker) [Yahav, 2001, 2004], though this has not been updated to take advantage of recent improvements in TVLA (see Section 4.4).

As a prototype tool it has only had three alpha releases (2002-7), and does not contain any additional reduction mechanisms such as partial order reduction. TVLA is written in Java and bytecode is available for academic use.

TVLA is used for the analyses in Chapters 7 and 8.

[8] `http://sal.csl.sri.com`

[9] `http://www.cs.tau.ac.il/~tvla/`

Chapter 4

Canonical Abstraction

This chapter describes the technique of *canonical abstraction* [Reps et al., 2004a; Sagiv et al., 2005], a powerful approachfor abstracting systems with different degrees of granularity. Additionally, it allows reasoning about properties involving transitive closure, and can be used for automated analyses with the prototype TVLA tool (see Section 3.6.3). Canonical abstraction will be used in Part III to construct finite state abstract models from infinite state models of nonblocking concurrent data structures. Section 4.1 introduces the concept of representing concrete and abstract states using 2- and 3-valued logical structures, and explains how they are related by canonical abstraction. Section 4.2 describes integrity rules and instrumentation predicates as means of refining abstractions, and Section 4.3 describes how abstract transitions are constructed from concrete semantics. Section 4.4 describes how threads can be abstracted to enable unbounded model checking. Finally, Section 4.5 outlines some improvements in analysing canonically abstract systems.

4.1 Canonical Abstraction

Generalising previous work in shape analysis of programs [e.g. Jones and Muchnick, 1979, 1982; Larus and Hilfinger, 1988; Horwitz et al., 1989; Chase et al., 1990; Stransky, 1992; Aßmann and Weinhardt, 1993; Plevyak et al., 1993; Wang, 1994; Sagiv et al., 1996, 1998] that represents the program heap as a directed graph, Sagiv et al. [1999, 2002] represent states as logical structures, where predicates describe relationships between objects. Concrete states are represented using 2-valued structures. Abstract states

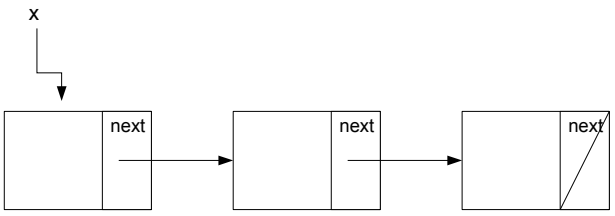


Figure 4.1: A list of length three

are represented using 3-valued structures, which allow multiple concrete objects to be represented by a single abstract "summary object" (or "summary node" [Chase et al., 1990]). Since a summary object can represent two or more concrete objects, an abstract state with summary objects can represent an infinite number of concrete states.

4.1.1 States as Logical Structures

First, a finite set of predicates $\mathcal{P} = \{\mathsf{eq}, \mathsf{p}_1, \ldots, \mathsf{p}_n\}$ is fixed for the analysis, and we define $\mathcal{P}_k$ to be the set of k-ary predicates in $\mathcal{P}$ (the equality predicate eq has arity 2). Then, a *concrete configuration* $S^\natural = \langle U^\natural, \iota^\natural \rangle$ has a *universe* $U^\natural$ that is a (finite or infinite) set of objects and an *interpretation* $\iota^\natural$ over the logical values true (1) and false (0). For each k-ary predicate p,

$$\iota^\natural(\mathsf{p}) : (U^\natural)^k \to \{0, 1\}$$

Additionally, for each $u_1, u_2 \in U^\natural$ where $u_1 \neq u_2$, $\iota^\natural(\mathsf{eq})(u_1, u_1) = 1$ and $\iota^\natural(\mathsf{eq})(u_1, u_2) = 0$.

Example Consider a program that has a pointer variable x to a list of Node objects that are linked by the Nodes' *next* fields. We might use a unary predicate node to indicate the type of Node objects, a unary predicate x to indicate which object is pointed to by x, and a binary predicate next to represent the relationship of the *next* field.

$$\mathcal{P} = \{\mathsf{eq}, \mathsf{node}, \mathsf{x}, \mathsf{next}\}$$

Figure 4.1 shows a state with x pointing to a list of length 3. The concrete configuration of this state has three objects and an interpretation as

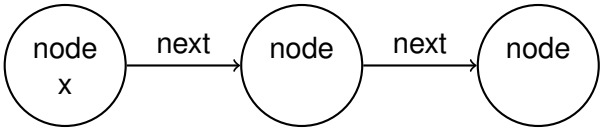


Figure 4.2: Graph of concrete configuration

follows, where in each case $1 \leqslant i, j \leqslant 3$:[1]

$$
\begin{aligned}
&U^\natural = \{u_1, u_2, u_3\} \\
&\iota^\natural(\mathsf{eq}) = \{\langle u_i, u_j\rangle \mapsto 1 \mid i = j\} \cup \{\langle u_i, u_j\rangle \mapsto 0 \mid i \neq j\} \\
&\iota^\natural(\mathsf{node}) = \{u_i \mapsto 1\} \\
&\iota^\natural(\mathsf{x}) = \{u_1 \mapsto 1, u_2 \mapsto 0, u_3 \mapsto 0\} \\
&\iota^\natural(\mathsf{next}) = \{\langle u_1, u_2\rangle \mapsto 1, \langle u_2, u_3\rangle \mapsto 1\} \cup \{\langle u_i, u_j\rangle \mapsto 0 \mid \langle i, j\rangle \notin \{\langle 1, 2\rangle, \langle 2, 3\rangle\}\}
\end{aligned}
$$

■

It is often helpful for comprehension to represent logical structures using graphs. For concrete configurations we use:

- graph nodes to represent objects in the universe,[2]
- labels on nodes to represent unary predicates (present if true, absent if false, for each object), and
- labelled arrows to represent binary predicates (present if true, absent if false, for each pair of objects).

Additionally, the equality predicate eq is not explicitly represented. Figure 4.2 shows the graphical representation for the configuration in the above example.

The definition of an *abstract configuration* $S = \langle U, \iota\rangle$ is similar to that of a concrete configuration, but the interpretation is over the truth values true (1), false (0) and unknown ($\frac{1}{2}$). For each k-ary predicate p,

$$\iota(\mathsf{p}) : U^k \to \{1, 0, \tfrac{1}{2}\}$$

Note that a concrete configuration is also trivially an abstract configuration.

[1] The names u_i have no meaning other than for ease of identification.

[2] We will generally say "objects" of the graph, to avoid confusion with the nodes of linked lists being represented.

An object u, for which $\iota(\mathsf{eq})(u,u)$ is unknown, is called a *summary object*. As we will see in the next section, these may represent more than one object in a given concrete state.

The graphical representation for abstract configurations is similar to concrete ones, but:

- summary nodes (where $\iota(\mathsf{eq})(u,u) = \frac{1}{2}$) have a double line,
- unary predicates with an unknown interpretation have an addition to the label (e.g. "$\mathsf{x} = \frac{1}{2}$"), and
- binary predicates with an unknown interpretation have dotted rather than solid arrows.

4.1.2 Embeddings

Intuitively, an abstract configuration represents a concrete one if it contains the same information, except for some conservative information loss. In other words, it has the same universe of objects, though some may have been merged together into summary objects, and it has the same predicate interpretations, though some may have become unknown. This is formalised by the notion of embedding, which relates configurations (concrete or abstract[3]) that are related by conservative information loss.

We say that a configuration $S_1 = \langle U_1, \iota_1 \rangle$ *embeds* into an abstract configuration $S_2 = \langle U_2, \iota_2 \rangle$ if there exists a surjective function $f : U_1 \to U_2$ such that for every k-ary predicate p, and $u_1, \ldots, u_k \in U_1$,

$$\iota_1(\mathsf{p})(u_1, \ldots, u_k) \sqsubseteq \iota_2(\mathsf{p})(f(u_1), \ldots, f(u_k))$$

where, for $l_1, l_2 \in \{1, 0, \frac{1}{2}\}$, $l_1 \sqsubseteq l_2$ iff $l_1 = l_2$ or $l_2 = \frac{1}{2}$.

Example One abstract configuration that is an embedding of the linked list state in the previous example has two objects, one of which is a summary object, as shown in Figure 4.3. Since the summary object represents the second two list nodes, the next predicate is unknown, as e.g. the x-node's next field points to one but not the other. Given the abstract configuration's universe as $U = \{u_{\{1\}}, u_{\{2,3\}}\}$ and the embedding function

[3]Since 2-valued configurations are trivially 3-valued configurations also, we will assume that configurations are 3-valued unless otherwise noted.

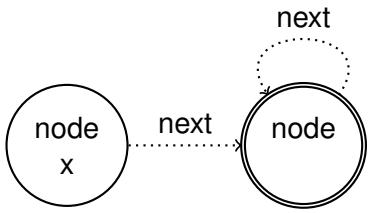


Figure 4.3: Graph of abstract configuration

$f = \{u_1 \mapsto u_{\{1\}}, u_2 \mapsto u_{\{2,3\}}, u_3 \mapsto u_{\{2,3\}}\}$, the interpretation is as follows:

$$\begin{aligned}
\iota(\mathsf{eq}) &= \left\{\left\langle u_{\{1\}}, u_{\{1\}}\right\rangle \mapsto 1, \left\langle u_{\{2,3\}}, u_{\{2,3\}}\right\rangle \mapsto \tfrac{1}{2},\right. \\
&\qquad \left.\left\langle u_{\{1\}}, u_{\{2,3\}}\right\rangle \mapsto 0, \left\langle u_{\{2,3\}}, u_{\{1\}}\right\rangle \mapsto 0\right\} \\
\iota(\mathsf{node}) &= \left\{u_{\{1\}} \mapsto 1, u_{\{2,3\}} \mapsto 1\right\} \\
\iota(\mathsf{x}) &= \left\{u_{\{1\}} \mapsto 1, u_{\{2,3\}} \mapsto 0\right\} \\
\iota(\mathsf{next}) &= \left\{\left\langle u_{\{1\}}, u_{\{2,3\}}\right\rangle \mapsto \tfrac{1}{2}, \left\langle u_{\{2,3\}}, u_{\{2,3\}}\right\rangle \mapsto \tfrac{1}{2},\right. \\
&\qquad \left.\left\langle u_{\{1\}}, u_{\{1\}}\right\rangle \mapsto 0, \left\langle u_{\{2,3\}}, u_{\{1\}}\right\rangle \mapsto 0\right\}
\end{aligned}$$

Note that not only is this 2-object list abstract configuration an embedding of the previous 3-object list concrete configuration in Figure 4.2, it is an embedding of many configurations with a list of *two or more* objects. The latter $n-1$ node objects get mapped to the summary node object, and the next predicates are set to unknown — thus we can embed a wide range of lists, whether or not they are connected or acyclic, though no node's next field can point to the x-node, as this would not be conservative information loss. A 1-object list configuration cannot embed into this abstract configuration, as a function between the universes would not be a surjection. ■

Example Every configuration $S = \langle U, \iota\rangle$ trivially embeds into the abstract configuration $S' = \langle U' = \{u_0\}, \iota'\rangle$ with one object and universally indefinite interpretation. For every $u \in U$, the embedding function maps $u \mapsto u_0$; and for every predicate p, $\iota'(\mathsf{p})(u_0, \ldots, u_0) = \frac{1}{2}$. ■

We further define a *tight embedding* to be one that minimises information loss, i.e. a predicate interpretation only becomes unknown if two objects are being merged together, one which has a true interpretation and

the other a false interpretation. Formally, there exists a surjective function $f : U_1 \to U_2$ such that for every k-ary predicate p, and $u_1, \ldots, u_k \in U_2$,

$$\iota_2(\mathsf{p})(u_1, \ldots, u_k) = \begin{cases} 1 & \text{if } \forall\, u'_1 \in f^{-1}(u_1), \ldots, u'_k \in f^{-1}(u_k) \bullet \\ & \qquad \iota_1(\mathsf{p})(u'_1, \ldots, u'_k) = 1 \\ 0 & \text{if } \forall\, u'_1 \in f^{-1}(u_1), \ldots, u'_k \in f^{-1}(u_k) \bullet \\ & \qquad \iota_1(\mathsf{p})(u'_1, \ldots, u'_k) = 0 \\ \frac{1}{2} & \text{otherwise} \end{cases}$$

Example The abstract configuration S in Figure 4.3 is actually a tight embedding of the configuration $S^\natural$ in Figure 4.2. However, if

$$\iota(\mathsf{next})(u_{\{2,3\}}, u_{\{1\}}) = \tfrac{1}{2}$$

then the information loss would not be minimal, as both

$$\iota^\natural(\mathsf{next})(u_2, u_1) = 0$$
$$\iota^\natural(\mathsf{next})(u_3, u_1) = 0$$

Thus it would still be an embedding, but not tight. ■

Canonical Abstraction

Canonical abstraction is a method for constructing tight embeddings. Given a subset of the unary predicates $\mathcal{A} \subseteq \mathcal{P}_1$, called the *abstraction predicates*, we map objects in the original configuration to the same abstract object if they have the same interpretations over the abstraction predicates. The interpretation in the abstract configuration is constructed as per the definition of tight embeddings above. We say that a configuration is canonically abstract, with respect to $\mathcal{A}$, if it is the canonical abstraction of itself.

One way of defining the unique embedding function from a structure $S = \langle U, \iota \rangle$ to its canonical abstraction $S' = \langle U', \iota' \rangle$ is to label every object in U' with the abstraction predicates that are true for it, i.e. $U' \subseteq \{u_A \mid A \subseteq 2^{\mathcal{A}}\}$. Now we can define the embedding function f as follows:

$$f = \{u \mapsto u_A \mid \begin{array}{l} u \in U \wedge u_A \in U' \wedge \\ (\forall\, \mathsf{p} \in A \bullet \iota(\mathsf{p})(u) = 1) \wedge \\ (\forall\, \mathsf{p} \in \mathcal{A} - A \bullet \iota(\mathsf{p})(u) \neq 1)\} \end{array}$$

Example The abstract configuration in Figure 4.3 is the canonical abstraction of the concrete configuration in Figure 4.2, with $\mathcal{A} = \mathcal{P}_1$. For both $u_2, u_3 \in U^\sharp$, $\iota^\sharp(\mathsf{node})(\_) = 1$ and $\iota^\sharp(\mathsf{x})(\_) = 0$ so they are both mapped to $u_{\{2,3\}} \in U$. The object $u_1 \in U^\sharp$ differs, as $\iota^\sharp(\mathsf{x})(u_1) = 1$, so it is mapped to the different abstract object $u_{\{1\}}$.

Following the above definition of the canonical abstraction embedding function, we would label the abstract objects $u_{\{1\}}$ and $u_{\{2,3\}}$ as $u_{\{\mathsf{node},\mathsf{x}\}}$ and $u_{\{\mathsf{node}\}}$, respectively. ■

Canonical abstraction has a number of important properties:

- Every configuration has a single canonical abstraction, as each object has a single canonical mapping in the embedding function.
- Since there are a finite number of predicates, it follows that there is a finite bound on the number of objects in the universe of a canonically abstract configuration, and thus a finite bound on the number of potential states in an abstract system.
- Given the canonical abstraction function α for a set of abstraction predicates, we can define a *concretisation* function

  $$\gamma(S) = \{S^\sharp \mid \alpha(S^\sharp) = S\}$$

  which maps an abstract configuration to the set of concrete configurations that abstract to it. These two functions form a Galois insertion.

4.1.3 Properties

We describe properties of configurations using first order logic with transitive closure (FOTC) [Sagiv et al., 2002].

Syntax

We define formulas over a set of predicates $\mathcal{P}$ and set of variables *Var* inductively as follows:

- The logical literals 0 and 1 are atomic formulas with no free variables.
- For every predicate $\mathsf{p} \in \mathcal{P}^k$, $\mathsf{p}(v_1, \ldots, v_k)$ is an atomic formula with free variables $\{v_1, \ldots, v_k\} \subseteq \mathit{Var}$.

- If φ_1 and φ_2 are formulas with sets of free variables $V_1 \subseteq \mathit{Var}$ and $V_2 \subseteq \mathit{Var}$ respectively, then $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ and $\neg\ \varphi_1$ are formulas with free variables $V_1 \cup V_2$, $V_1 \cup V_2$ and V_1 respectively.
- If φ_1 is a formula with free variables $\{v_1, \ldots, v_k\} \subseteq \mathit{Var}$, then $\exists\, v_1 \bullet \varphi_1$ and $\forall\, v_1 \bullet \varphi_1$ are both formulas with free variables $\{v_2, \ldots, v_k\}$.
- If φ_1 is a formula with free variables $V \subseteq \mathit{Var}$, such that $v_1, v_2 \in V$ and $v_3, v_4 \notin V$, then $(\mathrm{TC}\, v_1, v_2 \bullet \varphi_1)(v_3, v_4)$ is a formula with free variables $(V - \{v_1, v_2\}) \cup \{v_3, v_4\}$.

A *closed formula* is one with no free variables.

Semantics

Let $S = \langle U, \iota \rangle$ be a configuration, and $Z : \{v_1, \ldots\} \to U$ be a mapping from free variables to objects, called an *assignment*. The (3-valued) *meaning* of a formula φ, relative to S and Z, is written $[\![\varphi]\!]^S_3(Z)$, and is defined inductively below.

Atomic formulas If $\varphi = l \in \{1, 0\}$, then

$$[\![l]\!]^S_3(Z) = l$$

If $\varphi = \mathsf{p}(v_1, \ldots, v_k)$, for some $\mathsf{p} \in \mathcal{P}_k$, then

$$[\![\mathsf{p}(v_1, \ldots, v_k)]\!]^S_3(Z) = \iota(\mathsf{p})(Z(v_1), \ldots, Z(v_k))$$

Logical connectives If φ is built from subformulas φ_1 and φ_2 then

$$[\![\varphi_1 \wedge \varphi_2]\!]^S_3(Z) = \min([\![\varphi_1]\!]^S_3(Z), [\![\varphi_2]\!]^S_3(Z))$$
$$[\![\varphi_1 \vee \varphi_2]\!]^S_3(Z) = \max([\![\varphi_1]\!]^S_3(Z), [\![\varphi_2]\!]^S_3(Z))$$
$$[\![\neg\ \varphi_1]\!]^S_3(Z) = 1 - [\![\varphi_1]\!]^S_3(Z)$$

Quantifiers If φ is built from subformula φ_1 then[4]

$$[\![\forall\, v_1 \bullet \varphi_1]\!]^S_3(Z) = \min_{u \in U} [\![\varphi_1]\!]^S_3(Z[v_1 \mapsto u])$$
$$[\![\exists\, v_1 \bullet \varphi_1]\!]^S_3(Z) = \max_{u \in U} [\![\varphi_1]\!]^S_3(Z[v_1 \mapsto u])$$

[4]The notation $Z[v_i \mapsto u]$ is an assignment update, i.e. $\{v_j \mapsto u' \mid j \neq i \wedge Z(v_j) = u'\} \cup \{v_i \mapsto u\}$.

Transitive closure If φ is built from subformula φ_1 then

$$\begin{aligned}&[\![(\mathrm{TC}\, v_1, v_2 \bullet \varphi_1)(v_3, v_4)]\!]^S_3(Z) = \\ &\quad \max_{n \geqslant 1;\, u_1,\ldots,u_{n+1} \in U;\, Z(v_3)=u_1;\, Z(v_4)=u_{n+1}} \\ &\qquad \min_{i=1}^{n} [\![\varphi_1]\!]^S_3(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])\end{aligned}$$

We say that S and Z *potentially satisfy* φ, written $S, Z \models \varphi$, if $[\![\varphi]\!]^S_3(Z) = 1$ or $[\![\varphi]\!]^S_3(Z) = \frac{1}{2}$, and write $S \models \varphi$ if $S, Z \models \varphi$ for all Z.

We define the operators for implication, binary predicate transitive closure, and binary predicate reflexive transitive closure in the following ways:

$$\begin{aligned}\varphi_1 \rightarrow \varphi_2 &\equiv \neg\, \varphi_1 \vee \varphi_2 \\ \mathsf{p}^+(v_3, v_4) &\equiv (\mathrm{TC}\, v_1, v_2 \bullet \mathsf{p}(v_1, v_2))(v_3, v_4) \\ \mathsf{p}^*(v_3, v_4) &\equiv \mathsf{eq}(v_3, v_4) \vee \mathsf{p}^+(v_3, v_4)\end{aligned}$$

Embedding Theorem

The soundness of the canonical abstraction approach rests upon the Embedding Theorem of Sagiv et al. [2002, Theorem 4.9]. Informally, this says that if a structure S embeds into a structure S', then any information extracted from S' via a formula φ is a conservative approximation of the information extracted from S via φ. Alternatively, if we prove a property φ true or false in S', then we know it has the same value in S.

To formalise this, we extend functions on objects to act on assignments. If $f : U \rightarrow U'$ is a function over universes U and U', and Z is an assignment over U, then $f \circ Z$ is an assignment over U', where $(f \circ Z)(v) = f(Z(v))$.

Now, let $S = \langle U, \iota \rangle$ and $S' = \langle U', \iota' \rangle$ be structures, and $f : U \rightarrow U'$ a function that embeds S into S'. Then, for every formula φ and assignment Z,

$$[\![\varphi]\!]^S_3(Z) \sqsubseteq [\![\varphi]\!]^{S'}_3(f \circ Z)$$

The inductive proof is given by Sagiv et al. [2002, Appendix B].

4.2 Refining Abstractions

Canonical abstraction is a very coarse approach, and many predicates are lost on abstraction, i.e. they evaluate to unknown in the abstract state. This

can be addressed by making the abstraction less coarse, through the use of *integrity rules* and derived *instrumentation predicates*. The former describe properties of states, and are used to specify invariants that hold in the system. The latter describe properties of objects: they allow such properties to be explicitly recorded in an abstract state when they would otherwise evaluate to unknown, and they can be used as additional abstraction predicates to change the canonical abstraction.

4.2.1 Integrity Rules

Example In the running list example, the predicates x and next represent a global variable and a field, respectively. As such, we expect various properties to exist, for example that x is unique, and that next is functional. We can see though that the functional property is lost, as the formula

$$\forall\, v_1, v_2, v_3 \bullet \mathsf{next}(v_1, v_2) \wedge \mathsf{next}(v_1, v_3) \rightarrow \mathsf{eq}(v_2, v_3)$$

which evaluates to true in the concrete configuration in Figure 4.2, evaluates to unknown in the abstract configuration in Figure 4.3. Indeed, Figure 4.4 shows a configuration $S^\natural$ where next is not functional and has the same canonical abstraction S as in Figure 4.3. Three objects are mapped to the canonical object $u_{\{\mathsf{node}\}}$ so it is a summary object, but only one is mapped to $u_{\{\mathsf{x,node}\}}$ so it is a non-summary object. Now

$$\iota(\mathsf{next}(u_{\{\mathsf{x,node}\}}, u_{\{\mathsf{x,node}\}}) = \iota(\mathsf{next}(u_{\{\mathsf{node}\}}, u_{\{\mathsf{x,node}\}}) = 0$$

because next is false for each of the relevant concrete pairings, and

$$\iota(\mathsf{next}(u_{\{\mathsf{x,node}\}}, u_{\{\mathsf{node}\}}) = \iota(\mathsf{next}(u_{\{\mathsf{node}\}}, u_{\{\mathsf{node}\}}) = \tfrac{1}{2}$$

because next is true for some of the relevant concrete pairings and false for others. Clearly, this situation will happen whenever next is unknown and points to a summary object. ■

We can address such situations by imposing global invariants on configurations. We define a set F of bound FOTC formulas, called the *integrity rules*, and only consider configurations that potentially satisfy these formulas. Thus, we assume that concretisation is redefined as:

$$\gamma[F](S) = \{S^\natural \mid \alpha(S^\natural) = S \wedge \forall\, \varphi \in F \bullet S^\natural \models \varphi\}$$

Table 4.1 gives some example integrity rules that may be used to enforce properties of a predicate p, including uniqueness and functionality.

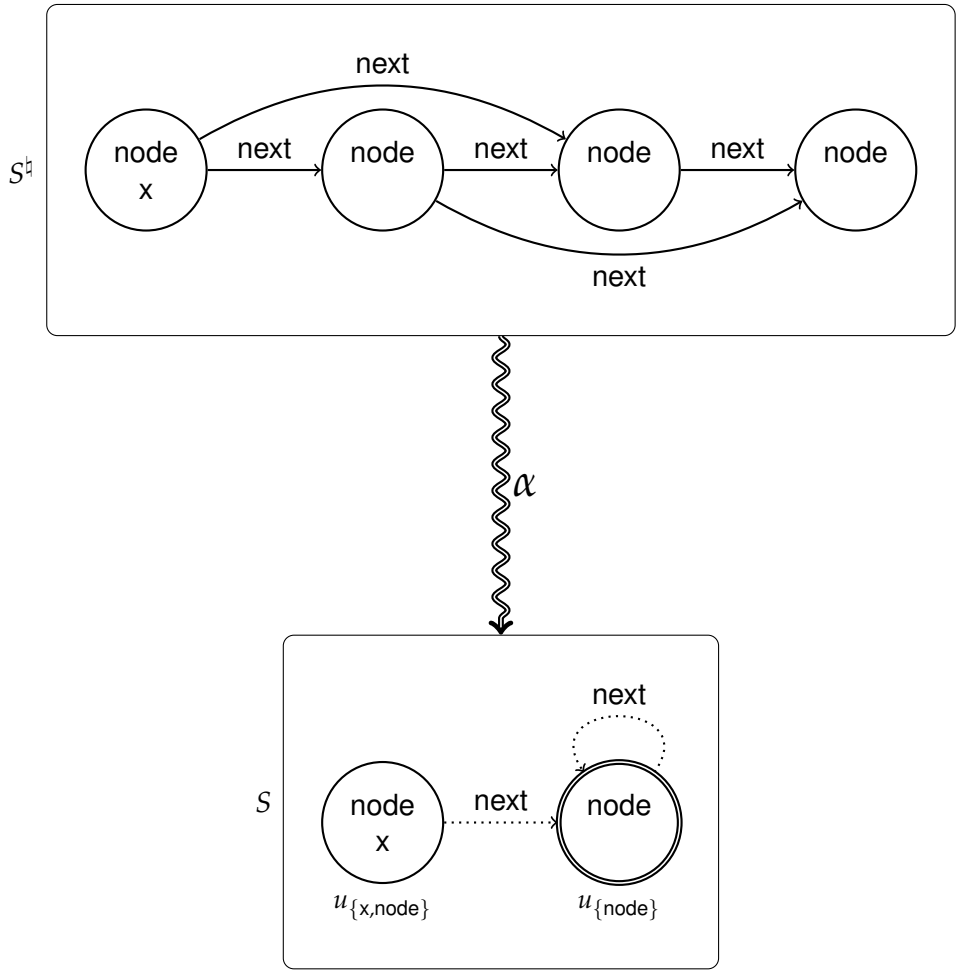


Figure 4.4: Canonical abstraction of a non-functional field

4.2.2 Instrumentation Predicates

An alternative way to refine the abstraction is to introduce additional predicates that record properties derived from the other predicates.

Example Consider the example configurations from Figure 4.2, which we will call $S_1^\natural$, and Figure 4.3, which we will call S_1. The concrete configuration $S_1^\natural$ represents a list of length 3, and the abstract configuration S_1 represents all lists of length 3 or more.[5] Connectedness and circularity are

[5]In Section 4.1.2 we said that lists of length 2 or more embed into S_1. Here we are

uniqueness:	$\forall v_1, v_2 \bullet \mathsf{p}(v_1) \wedge \mathsf{p}(v_2) \rightarrow \mathsf{eq}(v_1, v_2)$
functionality:	$\forall v_1, v_2, v_3 \bullet \mathsf{p}(v_1, v_2) \wedge \mathsf{p}(v_1, v_3) \rightarrow \mathsf{eq}(v_2, v_3)$
inverse functionality:	$\forall v_1, v_2, v_3 \bullet \mathsf{p}(v_1, v_3) \wedge \mathsf{p}(v_2, v_3) \rightarrow \mathsf{eq}(v_1, v_2)$
symmetry:	$\forall v_1, v_2 \bullet \mathsf{p}(v_1, v_2) \rightarrow \mathsf{p}(v_2, v_1)$

Table 4.1: Example integrity rules

often important properties to consider when analysing lists, so since $S_1^\natural$ represents a connected acyclic list, we might expect that S_1 similarly only represents connected and acyclic lists. However, when we consider these properties as logical formulas, we see that they have indefinite meanings in S_1, as the non-false next interpretations are all unknown. First, we check that each object can be reached from an x object by following 0 or more next fields:

$$[\![\forall v_2 \bullet \exists v_1 \bullet \mathsf{x}(v_1) \wedge \mathsf{next}^*(v_1, v_2)]\!]_2^{S_1^\natural} = 1$$
$$[\![\forall v_2 \bullet \exists v_1 \bullet \mathsf{x}(v_1) \wedge \mathsf{next}^*(v_1, v_2)]\!]_3^{S_1} = \tfrac{1}{2}$$

Second, we check that it is not possible to form a cycle of 1 or more next fields:

$$[\![\forall v_1 \bullet \neg\, \mathsf{next}^+(v_1, v_1)]\!]_2^{S_1^\natural} = 1$$
$$[\![\forall v_1 \bullet \neg\, \mathsf{next}^+(v_1, v_1)]\!]_3^{S_1} = \tfrac{1}{2}$$

Indeed, we can see in Figure 4.5 that S_1 is also the canonical abstraction of an unconnected acyclic list ($S_2^\natural$), and a connected cyclic list ($S_3^\natural$). ■

We can address such situations by introducing additional predicates that are defined by a formula using other predicates. These *instrumentation predicates* (or *derived predicates*) add no new information to concrete configurations, but may allow more definite information to be extracted from abstract configurations. We partition the set of predicates $\mathcal{P}$ into sets $\mathcal{C}$, containing the (non-derived) *core predicates*, and $\mathcal{I}$, containing the instrumentation predicates.

Example To the previous example we can add the following two instrumentation predicates, which are defined by the formulas we were unable

using "represent" to mean "is the canonical abstraction of"; a list of length 2 does not tightly embed into S_1.

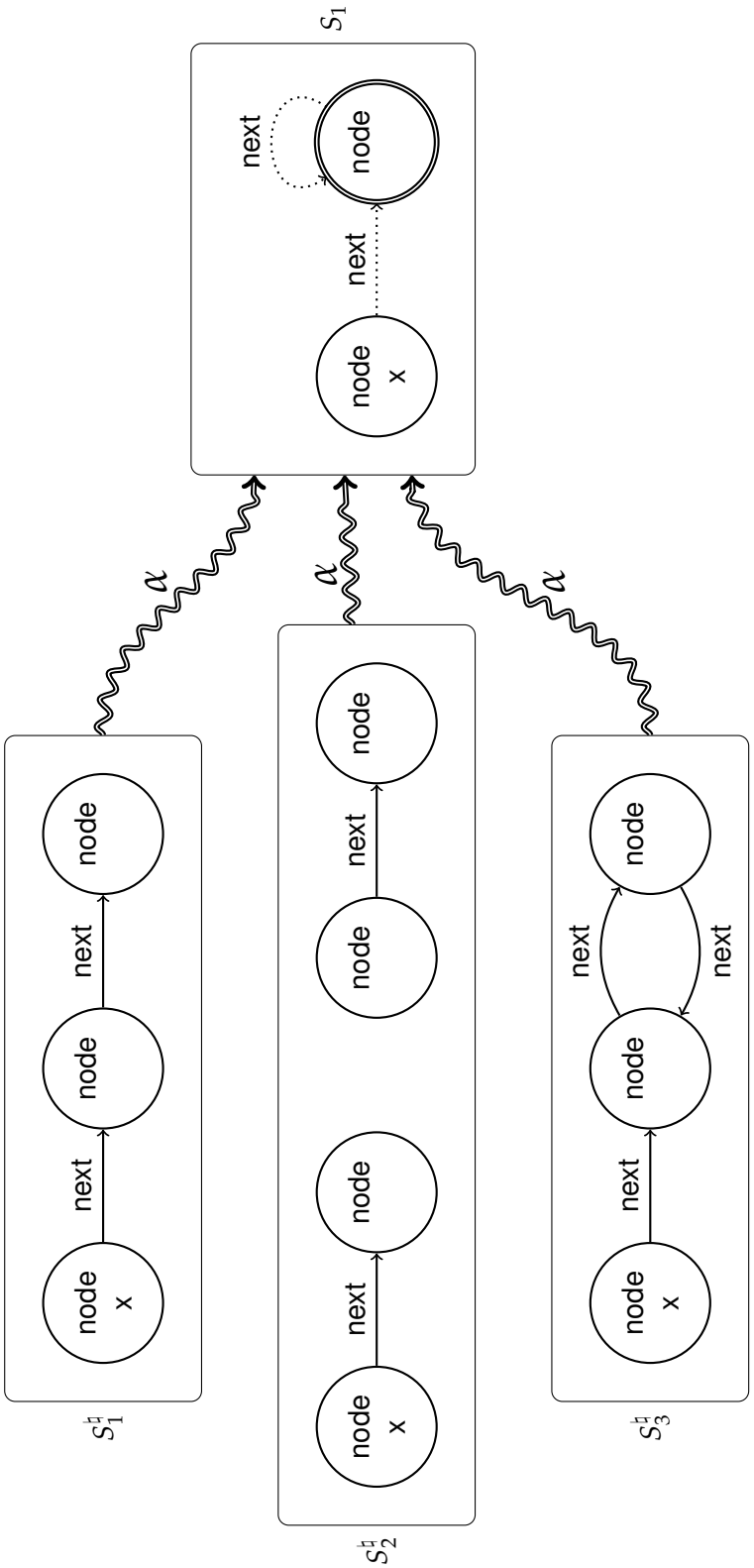
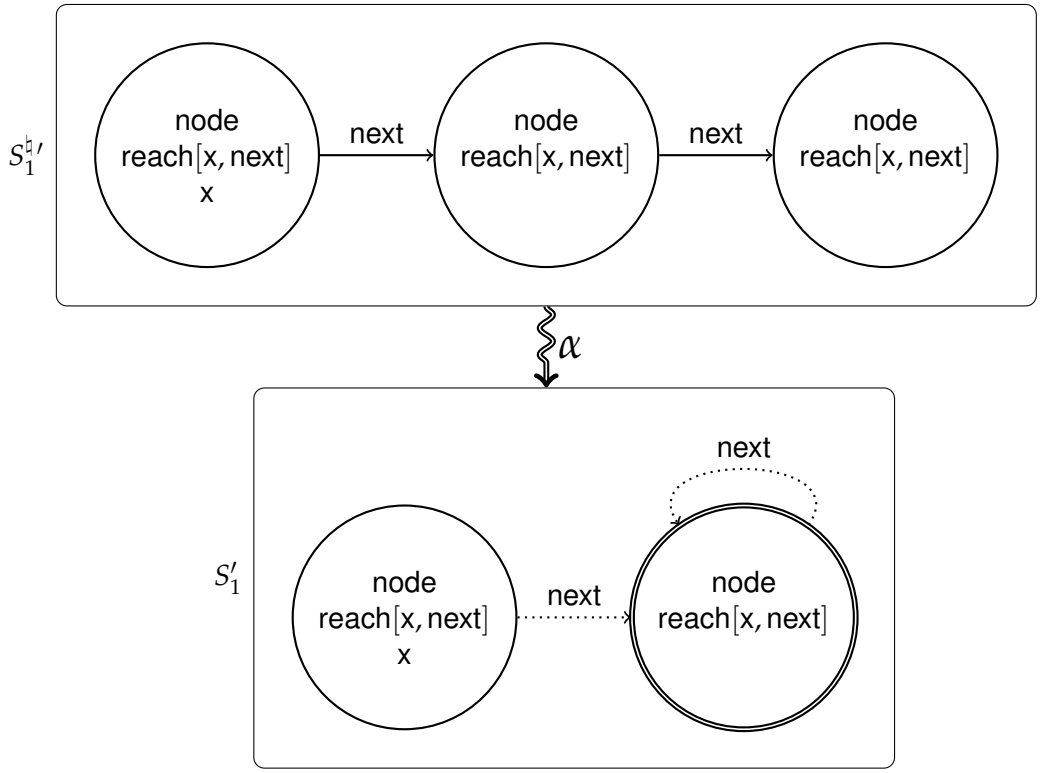


Figure 4.5: Three different lists have the same canonical abstraction



(a) Connected, acyclic

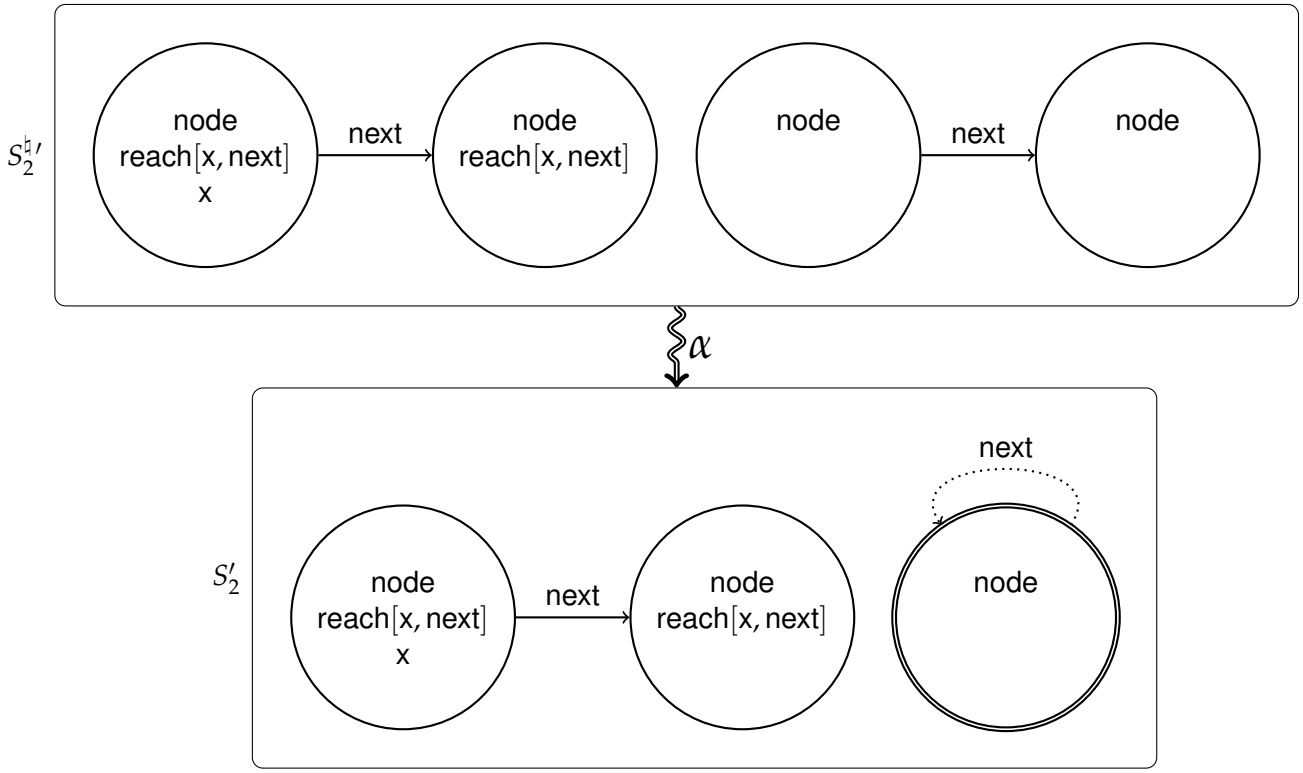
Figure 4.6: Using instrumentation predicates to distinguish three different list structures

to evaluate definitely:[6]

$$\mathsf{reach}[\mathsf{x},\mathsf{next}](v) = \exists u \bullet \mathsf{x}(u) \wedge \mathsf{next}^*(u,v)$$
$$\mathsf{circ}[\mathsf{next}](v) = \mathsf{next}^+(v,v)$$

Figure 4.6 (4.6a, 4.6b, 4.6c) shows the same concrete configurations from Figure 4.5 augmented with these instrumentation predicates. With these predicates added to the set of abstraction predicates these concrete configurations all have different canonical abstractions. S_1' is similar to S_1, but since $\mathsf{reach}[\mathsf{x},\mathsf{next}]$ is true for all nodes, we know that the list is connected. Also, since $\mathsf{circ}[\mathsf{next}]$ is false for all nodes, we know that the list is

[6]The square brackets have no meaning other than being a visual indicator of which core predicates are used in the definition. (TVLA allows parametrised definitions of sets of predicates in this way — e.g. to define $\mathsf{reach}[\mathsf{y},\mathsf{next}]$ and $\mathsf{reach}[\mathsf{z},\mathsf{next}]$ at the same time.)



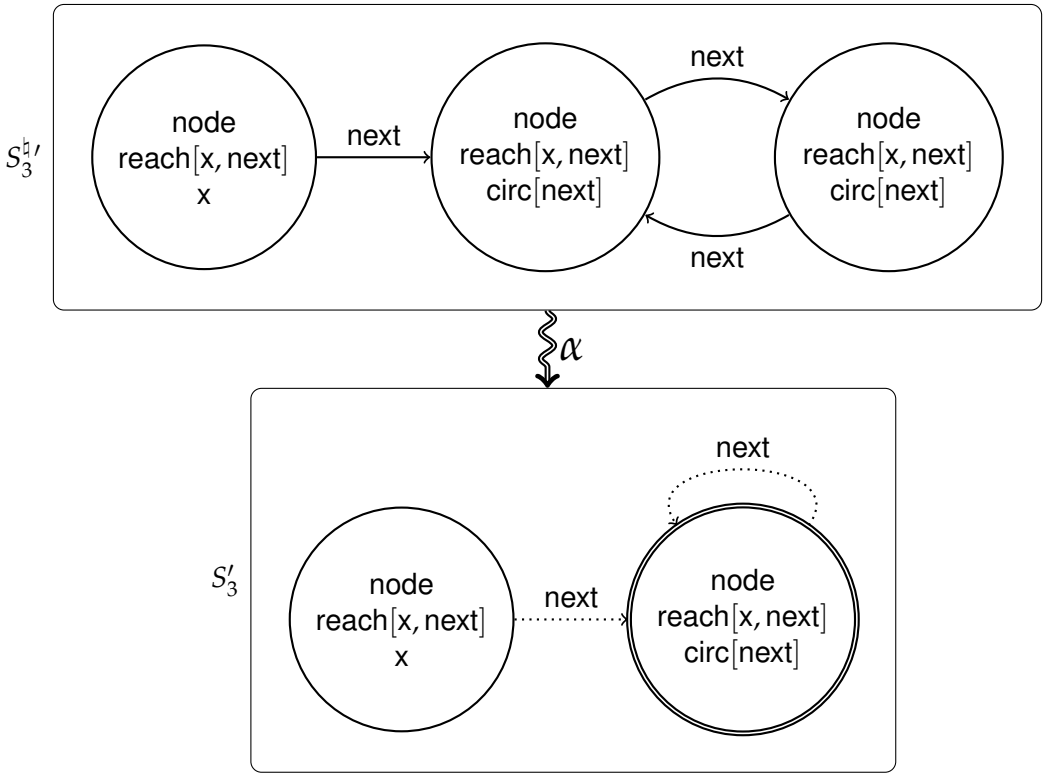
(b) Disconnected, acyclic

Figure 4.6: Using instrumentation predicates to distinguish three different list structures

acyclic. Thus, S_1' represents all connected acyclic lists of length 3 or more. Similarly, S_2' represents all acyclic lists with a connected part of length 2 and an unconnected part of length 2 or more; S_3' represents all connected lists of length 3 or more with a cycle from the second node.

These three abstract configurations illustrate how instrumentation predicates are able to reduce the amount of information loss that happens in canonical abstraction. First, each instrumentation predicate records definite meanings for the defining formulas, which may still have an unknown meaning when evaluated directly. For example in S_1', as with S_1, if we evaluate the defining formulas directly, we still get an indeterminate result:

$$[\![\forall v_2 \bullet \exists v_1 \bullet \mathsf{x}(v_1) \wedge \mathsf{next}^*(v_1, v_2)]\!]_3^{S_1'} = \tfrac{1}{2}$$
$$[\![\forall v_1 \bullet \neg\, \mathsf{next}^+(v_1, v_1)]\!]_3^{S_1'} = \tfrac{1}{2}$$



(c) Connected, cyclic

Figure 4.6: Using instrumentation predicates to distinguish three different list structures

However, if we evaluate the instrumentation predicates we discover that these properties are in fact true:

$$[\![\forall v_2 \bullet \mathsf{reach}[\mathsf{x},\mathsf{next}](v_2)]\!]_3^{S_1'} = 1$$
$$[\![\forall v_1 \bullet \neg\, \mathsf{circ}[\mathsf{next}](v_1)]\!]_3^{S_1'} = 1$$

Second, instrumentation predicates are able to be used as abstraction predicates and can prevent certain objects being merged together, as happened in S_2'. ■

Table 4.2 lists a number of instrumentation predicates that have been defined by other authors [see Sagiv et al., 2002; Yahav, 2001; Yahav and Sagiv, 2010]. Choosing the right instrumentation predicates is an important part of using canonical abstraction, and is the basis of the approaches described in Chapter 7 and Chapter 8.

Property	Predicate	Defining Formula
Does an object have a non-null p field?	$\mathsf{has}[\mathsf{p}](v)$	$\exists u \bullet \mathsf{p}(v,u)$
Is an object pointed to by a p field?	$\mathsf{r\_by}[\mathsf{p}](v)$	$\exists u \bullet \mathsf{p}(u,v)$
Is an object pointed to by 2 or more p fields?	$\mathsf{shared}[\mathsf{p}](v)$	$\exists u_1, u_2 \bullet \mathsf{p}(u_1,v) \wedge \mathsf{p}(u_2,v) \wedge \neg\, \mathsf{eq}(u_1,u_2)$
Is an object reachable from a p-object, following q fields?	$\mathsf{reach}[\mathsf{p},\mathsf{q}](v,u)$	$\exists u \bullet \mathsf{p}(u) \wedge \mathsf{q}^*(u,v)$
Is an object on a cycle of p fields?	$\mathsf{circ}[\mathsf{p}](v)$	$\mathsf{p}^+(v,v)$

Table 4.2: Example instrumentation predicates

4.3 Abstract Transitions

Now we have seen how to use canonical abstraction to construct abstract representations of individual states, we will explore how to construct abstract representations of systems, which preserve ACTL* properties.[7] Since canonical abstraction induces a Galois insertion, it follows that it is an instance of the abstract interpretation framework. In Section 3.5.1, we saw how abstract interpretation can be used to construct an abstract transition system that preserves ACTL* properties by defining a transition τ_α between two states S_1 and S_2 when each has a state in its concretisation (S_1' and S_2', respectively) that are related by the concrete transition τ, i.e.

$$\begin{array}{ccc} S_1' & \xrightarrow{\tau} & S_2' \\ {\scriptstyle\gamma}\uparrow & & \uparrow{\scriptstyle\gamma} \\ S_1 & & S_2 \end{array} \quad \text{implies} \quad S_1 \xrightarrow{\tau} S_2$$

This does not, however, provide a practical algorithm for constructing the abstract transitions, as the concretisation of an abstract state may contain an infinite number of concrete states.

We can approximate this though by using an approach that constructs a *partial* concretisation instead. In the partial concretisation, only the parts

[7] As far as I am aware, canonical abstraction has not been used in the literature to check ECTL* properties; in Part III we only consider ACTL* properties.

of the state that are relevant to the transition are concretised, i.e. the state is concretised only enough for the precondition and updates of the transition to evaluate to definite values [Sagiv et al., 2002, §6]. The precise steps are:

- *Focus*, which performs an optimistic partial concretisation, so that a given set of formulas all evaluate to definite values;
- *Coerce*, which removes the inconsistent states generated by the optimistic Focus operation;
- *Update*, which applies the operational semantics of the concrete transition; and
- *Blur*, which performs canonical abstraction.

Focus and Coerce are *semantic reductions* [Cousot and Cousot, 1979], which means that they take a set of abstract states and return a more precise set that represent the same set of concrete states. From this and the Embedding Theorem, it follows that this approach is safe [Sagiv et al., 2002, Theorem 6.29].

An abstract system based on canonical abstraction will always be finite, as there is a finite bound on the number of abstract states (due to the finite number of predicates used, as mentioned earlier in Section 4.1.2). Such an abstract system constructed using the approach described here may not be as precise as theoretically possible, i.e. by fully concretising an abstract state, applying a concrete transition to each of the concrete states, and then reabstracting the resulting states.

In the remainder of this section we examine in more detail the Focus operation in Section 4.3.1, the Coerce operation in Section 4.3.2 and the Update operation in Section 4.3.3. Section 4.3.4 contains an example of an abstract transition, which shows how each of these steps is applied, and demonstrates the potential imprecision. In Section 4.3.5 we discuss computing the most precise abstract transitions.

4.3.1 Focus Operation

The first step is the *Focus* operation, which "focusses" some of the indefinite information in a structure. Given an abstract configuration S and a set of logical formulas Φ (the *focus formulas*), it returns the minimum set of abstract configurations that together represent the same set of concrete

configurations as the original, i.e. $\gamma(S) = \gamma(\text{Focus}(S, \Phi))$, but in which all the focus formulas evaluate to definite values. So for every configuration $S' \in \text{Focus}(S, \Phi)$, formula $\varphi \in \Phi$ and assignment Z:

$$[\![\varphi]\!]_3^{S'}(Z) \in \{0, 1\}$$

Sagiv et al. [2002, §6.3.1] give an algorithm for a restricted class of formulas. Lev-Ami [2000, §6] gives a general algorithm, with conservative checks for the generation of an infinite number of structures. The basic idea of Lev-Ami's algorithm is to convert each formula into conjunctive normal form, and then 'focus' each atomic component in turn. When a predicate and assignment on free variables are encountered that evaluate to unknown, Focus produces either two or three modified copies:

1. a configuration where the predicate is set to true;
2. a configuration where the predicate is set to false; and
3. if the predicate is binary and exactly one of the objects is a summary object, then a configuration where there are two copies of the summary object — one with the predicate set to true and the other with it set to false.

Figure 4.7 shows an example of focussing a binary predicate p, and the three states that result. Note that focussing a binary predicate with two summary objects could produce an infinite number of configurations — the algorithm fails with an error in this case.

There is no set method for constructing focus formulas for each transition. In general, the intention is to concretise the parts of the structure that are read or modified by (the precondition and update formulas of) the transition — this is discussed by Sagiv et al. [2002, §6.3.2]. Any choice is safe, but as expected if not enough information is focussed it may make the analysis too coarse.

4.3.2 Coerce Operation

The Coerce operation "coerces" structures to be more precise. It takes an abstract configuration and a set of constraints (defined below), and returns either the empty set, or a singleton set containing the input configuration with zero or more indefinite predicate interpretations set to definite values

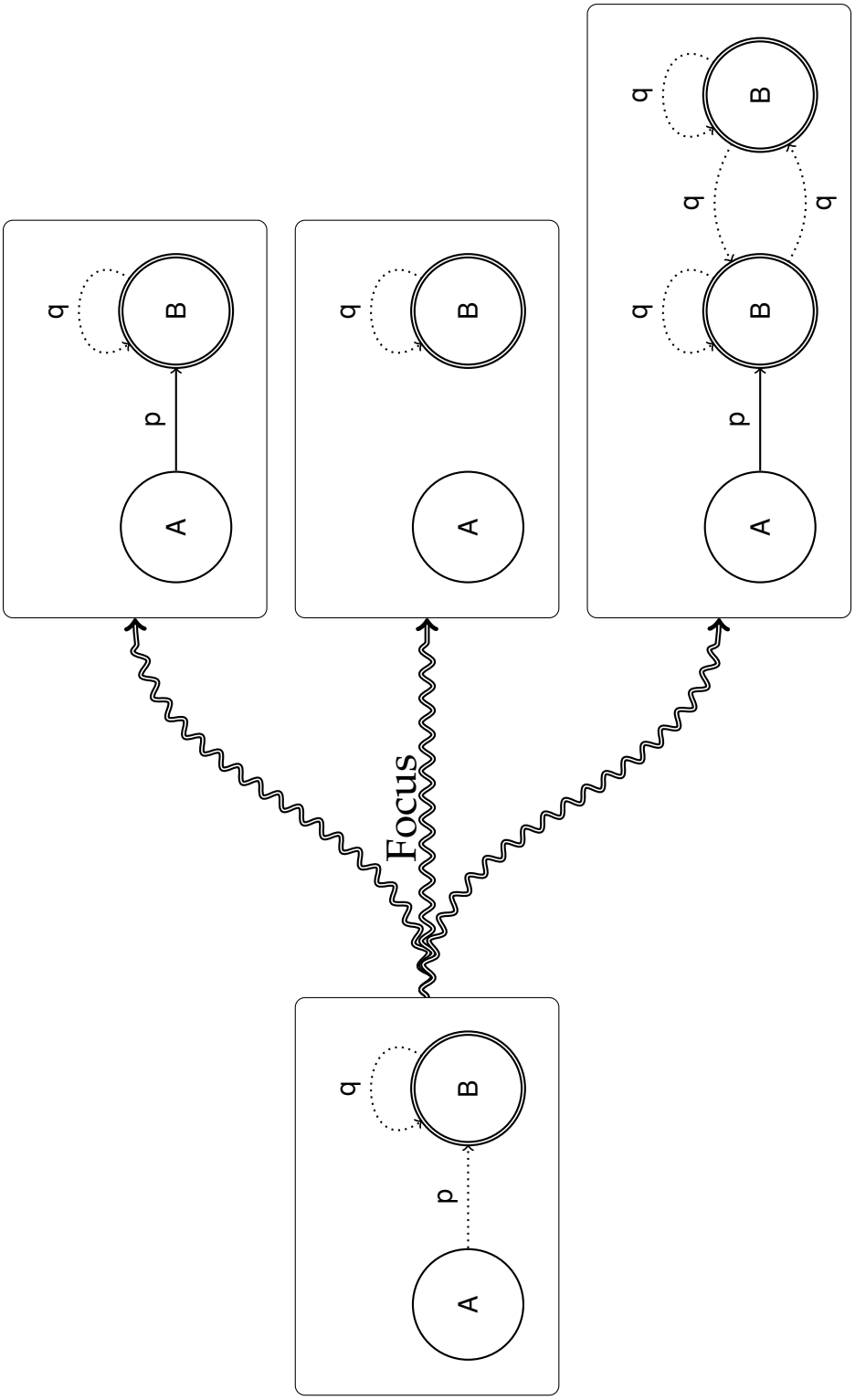


Figure 4.7: Focus on a single predicate: Focus($\mathsf{p}(v_1, v_2)$)

(true or false). Intuitively, the resulting configuration represents exactly the same set of concrete configurations that the initial one does. In the latter case, the indefinite interpretations have the same definite interpretation in every embedding concrete configuration. In the former case, the structure is logically inconsistent.

Compatibility constraints

A *compatibility constraint* is a formula of the form:

$$\varphi_1 \rhd \varphi_2$$

where the *head* φ_2 is an atomic FOTC formula (a literal or predicate) or the negation of an atomic FOTC formula, and the *body* φ_1 is an arbitrary FOTC formula. The novel operator $\rhd$ is defined semantically as follows:

$$S, Z \models \varphi_1 \rhd \varphi_2 \quad \text{iff} \quad [\![\varphi_1]\!]^S_3(Z) = 1 \text{ implies } [\![\varphi_2]\!]^S_3(Z) = 1$$

In 2-valued configurations, $\varphi_1 \rhd \varphi_2$ has exactly the same meaning as $\varphi_1 \rightarrow \varphi_2$. In 3-valued configurations though it has a stronger meaning; if $[\![\varphi_1]\!]^S_3(Z) = 1$ and $[\![\varphi_2]\!]^S_3(Z) = \frac{1}{2}$ then $\varphi_1 \rightarrow \varphi_2$ is satisfied but $\varphi_1 \rhd \varphi_2$ is not.

The constraints used in an analysis come from the integrity rules and instrumentation predicates, which were discussed in Section 4.2. To the set of integrity rules, we add the following two formulas for each instrumentation predicate p, where k is the arity of p and φ_{p} is its defining formula:

$$\forall v_1 \cdots v_k \bullet \varphi_{\mathsf{p}} \rightarrow \mathsf{p}(v_1, \ldots, v_k)$$
$$\forall v_1 \cdots v_k \bullet \neg\, \varphi_{\mathsf{p}} \rightarrow \neg\, \mathsf{p}(v_1, \ldots, v_k)$$

From these formulas we construct the set of constraints using the following translation. Let φ be a closed formula, and a be an atomic formula or the negation of an atomic formula. Then, the constraint generated from φ is:

$$\begin{array}{rl} \varphi_1 \rhd a & \text{if } \varphi = \forall v_1 \cdots v_k \bullet \varphi_1 \rightarrow a \\ \neg\, \varphi \rhd 0 & \text{otherwise} \end{array}$$

Additionally, we include the *extended Horn clause closure* of each constraint. For every constraint of the form

$$a_1 \wedge \cdots \wedge a_n \rhd a_0$$

where each a_i is an atomic formula or the negation of an atomic formula, we also include the following constraints, for all $1 \leqslant i \leqslant n$:[8]

$$\neg\, a_0 \wedge a_1 \wedge \cdots \wedge a_{i-1} \wedge a_{i+1} \wedge \cdots \wedge a_n \rhd \neg\, a_i$$

Coerce algorithm

The Coerce algorithm given by Sagiv et al. [2002, §6.4.4] is fairly straightforward. Briefly, each constraint is evaluated for every assignment over free variables. For each case where the body evaluates to true and the head evaluates to unknown, the predicate in the head is set to true (or false, if the head contains a negated predicate). If a case is reached where the body evaluates to true and the head evaluates to false, then the structure is discarded and Coerce returns the empty set.

Coerce is performed in between Update and Blur, in order to ensure that the structures are as precise as possible before re-abstracting them. In practice it is more efficient to also perform it between Focus and Update, so that unnecessary computation is not wasted performing updates on inconsistent configurations.

4.3.3 Update

The Update operation that is performed during the construction of an abstract transition is simply the application of the concrete transition to a partially concretised state.

The concrete operational semantics of a transition are given in two parts: the precondition formula ψ_{pre} and the set of update formulas Ψ. The precondition ψ_{pre} can be any FOTC formula. The set Ψ contains a formula of the form $\mathsf{p}(v_1, \ldots, v_k) = \varphi_{\mathsf{p}}$ for every predicate p, where k is the arity of p and φ_{p} can be any FOTC formula with free variables that are a subset of $\{v_1, \ldots, v_k\}$.

We say that a transition $\tau = \langle \psi_{pre}, \Psi \rangle$ holds between concrete states (2-valued configurations) $S_1^\natural$ to $S_2^\natural$ iff their universes are the same, i.e. $U_1^\natural = U_2^\natural$, there is an assignment that makes the precondition true in the first state $S_1^\natural$, i.e.

$$\exists\, Z \bullet [\![\psi_{pre}]\!]_2^{S_1^\natural}(Z) \neq 0$$

[8] The construction is slightly more complicated, as we must ensure that the free variables of the body and head are the same; Sagiv et al. [2002, §6.4.2] have more details.

and every predicate has the same value in the second state $S_2^\natural$ as its update formula has in the first, i.e.

$$\forall \mathsf{p} \bullet \forall Z \bullet [\![\mathsf{p}(v_1, \ldots, v_k)]\!]_2^{S_2^\natural}(Z) = [\![\varphi_\mathsf{p}]\!]_2^{S_1^\natural}(Z)$$

During the construction of of an abstract transition, we apply the concrete transition to partially concretised abstract states, rather than concrete states. However, we can easily redefine the definition using 3-valued semantics. We say that a transition τ hold between abstract states (3-valued configurations) S_1 and S_2 iff

$$\begin{aligned}
&U_1 = U_2 \\
&\exists Z \bullet [\![\psi_{pre}]\!]_3^{S_1}(Z) \neq 0 \\
&\forall \mathsf{p} \bullet \forall Z \bullet [\![\mathsf{p}(v_1, \ldots, v_k)]\!]_3^{S_2}(Z) = [\![\varphi_\mathsf{p}]\!]_3^{S_1}(Z)
\end{aligned}$$

New objects

The requirements for the Update operation state that the universes of the pre and post states must be identical, which would preclude transitions that create or destroy objects. Such transitions can be represented with the use of an additional predicate and operation.

For transitions that create a single object, the operation New is performed immediately before Update. New makes two modifications to the state — it adds an object to the universe and sets a unary predicate is_New to be true for (only) that object.

The new object can be specified in the update formulas of the succeeding Update operation via the is_New predicate. The update formula for the predicate is always $\mathsf{is\_New}(v) = 0$.

4.3.4 Example

To illustrate how abstract transitions are created using Focus and Coerce, we will look at applying a transition to the running example. So far, we have considered an abstract configuration that represents connected acyclic lists of length 2 or more (see S_1' in Figure 4.6a). To this state we will apply a transition that advances the variable x along the list, i.e.

$$x := x.next$$

First, let us consider the aspects that are not transition dependent. In this simple example we use only four predicates — two core predicates representing the global variable and the next field, and two instrumentation predicates recording circularity and reachability:

$$\mathcal{C}_1 = \{\mathsf{x}\}$$
$$\mathcal{C}_2 = \{\mathsf{next}\}$$
$$\mathcal{I}_1 = \{\mathsf{circ}[\mathsf{next}], \mathsf{reach}[\mathsf{x}, \mathsf{next}]\}$$

There are eight constraints used with this abstraction. Four come from the integrity rules that enforce the uniqueness of x and the functionality of next (two directly from the rules, and two from their extended Horn clause closures):

$$\mathsf{x}(v_1) \wedge \mathsf{x}(v_2) \quad \rhd \quad \mathsf{eq}(v_1, v_2) \tag{C.1}$$
$$\exists\, v_2 \bullet \neg\, \mathsf{eq}(v_1, v_2) \wedge \mathsf{x}(v_2) \quad \rhd \quad \neg\, \mathsf{x}(v_1) \tag{C.2}$$
$$\exists\, v_1 \bullet \mathsf{next}(v_1, v_2) \wedge \mathsf{next}(v_1, v_3) \quad \rhd \quad \mathsf{eq}(v_2, v_3) \tag{C.3}$$
$$\exists\, v_3 \bullet \neg\, \mathsf{eq}(v_2, v_3) \wedge \mathsf{next}(v_1, v_3) \quad \rhd \quad \neg\, \mathsf{next}(v_1, v_2) \tag{C.4}$$

The remaining four come from the definitions of the instrumentation predicates:

$$\mathsf{next}^+(v_1, v_1) \quad \rhd \quad \mathsf{circ}[\mathsf{next}](v_1) \tag{C.5}$$
$$\neg\, \mathsf{next}^+(v_1, v_1) \quad \rhd \quad \neg\, \mathsf{circ}[\mathsf{next}](v_1) \tag{C.6}$$
$$\exists\, v_1 \bullet \mathsf{x}(v_1) \wedge \mathsf{next}^*(v_1, v_2) \quad \rhd \quad \mathsf{reach}[\mathsf{x}, \mathsf{next}](v_2) \tag{C.7}$$
$$\forall\, v_1 \bullet \neg\, \mathsf{x}(v_1) \vee \neg\, \mathsf{next}^*(v_1, v_2) \quad \rhd \quad \neg\, \mathsf{reach}[\mathsf{x}, \mathsf{next}](v_2) \tag{C.8}$$

Initial State

The initial state we will apply the transition to is S_1' in Figure 4.6, and we will label it S_0 here. The universe has two objects, and the interpretation is

as follows:

$U = \{u_1, u_2\}$

ι		u_1	u_2
x		1	0
circ[next]		0	0
reach[x, next]		1	1
next	u_1	0	$\frac{1}{2}$
	u_2	0	$\frac{1}{2}$
eq	u_1	1	0
	u_2	0	$\frac{1}{2}$

Focus

To apply the transition we require that the x node and its successor are both concrete (see the Update operation), so we use the single focus formula

$$\mathsf{x}(v_1) \wedge \mathsf{next}(v_1, v_2)$$

Now $[\![\mathsf{x}(u_2)]\!]_3^{S_0} = 0$, so

$$[\![\mathsf{x}(u_2) \wedge \mathsf{next}(u_2, v_2)]\!]_3^{S_0}(Z) = 0$$

for any Z, even though $[\![\mathsf{next}(u_2, u_2)]\!]_3^{S_0} = \frac{1}{2}$. Then

$$[\![\mathsf{x}(u_1) \wedge \mathsf{next}(u_1, u_1)]\!]_3^{S_0} = 0$$

However,

$$[\![\mathsf{x}(u_1) \wedge \mathsf{next}(u_1, u_2)]\!]_3^{S_0} = \tfrac{1}{2}$$

when it is required to be a definite value. Since u_2 is a summary node ($[\![\mathsf{eq}(u_2, u_2)]\!]_3^{S_0} = \frac{1}{2}$) the Focus operation produces three modified states where the focus formula evaluates only to definite values, as in Figure 4.7. We will label these S_1, S_2 and S_3.

States S_1 and S_2 are exactly the same as S_0 except that the indefinite predicate has been assigned a definite value, i.e.

$$[\![\mathsf{next}(u_1, u_2)]\!]_3^{S_1} = 1$$
$$[\![\mathsf{next}(u_1, u_2)]\!]_3^{S_2} = 0$$

thus for all Z

$$[\![\mathsf{x}(v_1) \wedge \mathsf{next}(v_1, v_2)]\!]_3^{S_1}(Z) \in \{0, 1\}$$
$$[\![\mathsf{x}(v_1) \wedge \mathsf{next}(v_1, v_2)]\!]_3^{S_2}(Z) = 0$$

For S_3 the summary object gets duplicated so that each copy gets a different definite value for the predicate:

$U = \{u_1, u_2, u_3\}$

ι		u_1	u_2	u_3
x		1	0	0
circ[next]		0	0	0
reach[x, next]		1	1	1
next	u_1	0	1	0
	u_2	0	$\frac{1}{2}$	$\frac{1}{2}$
	u_3	0	$\frac{1}{2}$	$\frac{1}{2}$
eq	u_1	1	0	0
	u_2	0	$\frac{1}{2}$	0
	u_3	0	0	$\frac{1}{2}$

Now $[\![\mathsf{x}(v_1) \wedge \mathsf{next}(v_1, v_2)]\!]_3^{S_3}(Z) \in \{0, 1\}$ for all Z.

Coerce

First State In S_1, all the constraint formulas are satisfied except for C.3, as

$$[\![\exists\, v_1 \bullet \mathsf{next}(v_1, u_2) \wedge \mathsf{next}(v_1, u_2)]\!]_3^{S_1} = 1$$

but

$$[\![\mathsf{eq}(u_2, u_2)]\!]_3^{S_1} = \tfrac{1}{2}$$

Coerce replaces S_1 with a state S_1' that differs only by making the second node non-summary, i.e. $\iota(\mathsf{eq}(u_2, u_2)) = 1$. Now all constraints are satisfied, notably

$$S_1' \models \exists\, v_1 \bullet \mathsf{next}(v_1, v_2) \wedge \mathsf{next}(v_1, v_3) \rhd \mathsf{eq}(v_2, v_3)$$

Second State In S_2, all the constraint formulas are satisfied except for C.8, as

$$[\![\forall v_1 \bullet \neg\, \mathsf{x}(v_1) \vee \neg\, \mathsf{next}^*(v_1, u_2)]\!]_3^{S_2} = 1$$

but

$$[\![\neg\, \mathsf{reach}[\mathsf{x}, \mathsf{next}](u_2)]\!]_3^{S_2} = 0$$

There is no way to satisfy this constraint, so Coerce discards S_2.

Third State In S_3, as for S_1, all the constraint formulas are satisfied except for C.3. Again, as for S_1, Coerce replaces S_3 with a state S_3' that differs only by making the second node non-summary, i.e. $\iota(\mathsf{eq}(u_2, u_2)) = 1$.

Update

The precondition of the transition is the formula

$$\mathsf{x}(v_1) \wedge \mathsf{next}(v_1, v_2)$$

Both S_1' and S_3' satisfy this precondition with the assignment

$$[v_1 \mapsto u_1, v_2 \mapsto u_2]$$

The update formulas for x and the instrumentation predicate that uses it are

$$\begin{aligned} \mathsf{x}(v_1) &= \exists v_2 \bullet \mathsf{x}(v_2) \wedge \mathsf{next}(v_2, v_1) \\ \mathsf{reach}[\mathsf{x}, \mathsf{next}](v_1) &= \mathsf{reach}[\mathsf{x}, \mathsf{next}](v_1) \wedge (\neg\, \mathsf{x}(v_1) \vee \mathsf{circ}[\mathsf{next}](v_1)) \end{aligned}$$

None of the other predicates are altered so their update formulas are of the form $\mathsf{p}(v_1, \ldots, v_k) = \mathsf{p}(v_1, \ldots, v_k)$.

New states S_4 and S_5 are constructed from S_1' and S_3', respectively, sharing the same universes. The interpretations are constructed using the update formulas and are the same as the previous states except for the following:

S_5	ι	u_1	u_2
x		0	1
reach[x, next]		0	1

S_5	ι	u_1	u_2	u_3
x		0	1	0
reach[x, next]		0	1	1

Coerce

Both S_4 and S_5 satisfy all eight constraint formulas, so Coerce makes no changes to either state.

Blur

Blur also makes no changes to either S_4 or S_5. Both states are already canonically abstract, as all (respectively two and three) nodes have distinct abstraction (unary) predicate interpretations.

4.3.5 Computing the Best Abstract Transition

As mentioned earlier in this section, the construction of abstract transitions described here is not always as precise as is theoretically possible, though it is practical and always terminates. An alternative approach has been investigated [Reps et al., 2004b; Yorsh et al., 2004, 2007] that constructs the "best" abstract transitions (equivalent to fully concretising abstract states, applying the concrete transition, and reabstracting). It follows the approach from predicate abstraction [Graf and Saïdi, 1997] of using an automated theorem prover to construct the abstract transitions, and encodes 3-valued abstract states using 2-valued formulas. This approach has the advantage of constructing more precise abstract systems, but has the disadvantage that the procedure may not terminate, as first order logic is undecidable. The results reported are currently slower than TVLA and do not allow transitive closure, though further work has investigated first order simulation of transitive closure in some circumstances [Lev-Ami et al., 2009].

4.4 Concurrent Systems

Canonical abstraction can be extended from representing the heap of sequential systems to representing concurrent systems by treating threads in a similar way to the heap objects. This approach was introduced by Yahav for model checking Java programs [Yahav, 2001, 2004; Yahav and Sagiv, 2010].

4.4.1 States

The key idea of this approach is to represent threads in the same way as any other object. Each thread is represented by an object in the universe and has the unary predicate is_thread true. The fields of the threads are represented by binary predicates, as for other objects' fields.[9] The threads have a finite set of locations, which are represented by unary predicates of the form at[label], for each location label. A further core predicate is introduced, called tr_scheduled, which records which thread object is performing a transition.

Example Figure 4.8 shows the graph of the canonical abstraction of a possible state of the stack algorithm from Figure 2.7 (in fact, of an infinite number of possible states). The stack contains three or more elements, and is distinguished by the unary predicate (representing the global variable) Head and the previously introduced instrumentation predicate reach[Head, next]. The data values are marked by the unary predicate is_data, but there is nothing further to distinguish them so they are all abstracted to a single summary object. On the right, there are two or more threads performing a pop operation, which have read a snapshot at line 11 that is now stale (i.e. ss does not point to the node marked by Head) and which are waiting to perform line 12. Note that because ss is unknown for these threads, the stack snapshot could be either a node later in the list or it could be null (i.e. if ss is false). On the left, there is a single thread performing a push operation, which is about to perform the CAS step at line 7. This thread is the next scheduled to perform a transition, as tr_scheduled is true, and the CAS will succeed because ss does point to the Head node. ■

Yahav [2001, 2004] also discusses the representation of locks and threads that can be created, destroyed, and made inactive; these are not used in this thesis.

4.4.2 Transitions

Each algorithm transition operates on the thread that has the unique unary predicate tr_scheduled true. Nondeterministically assigning this predicate

[9] Yahav defines a predicate rv[fld] for each field *fld*, but we will just define them as fld.

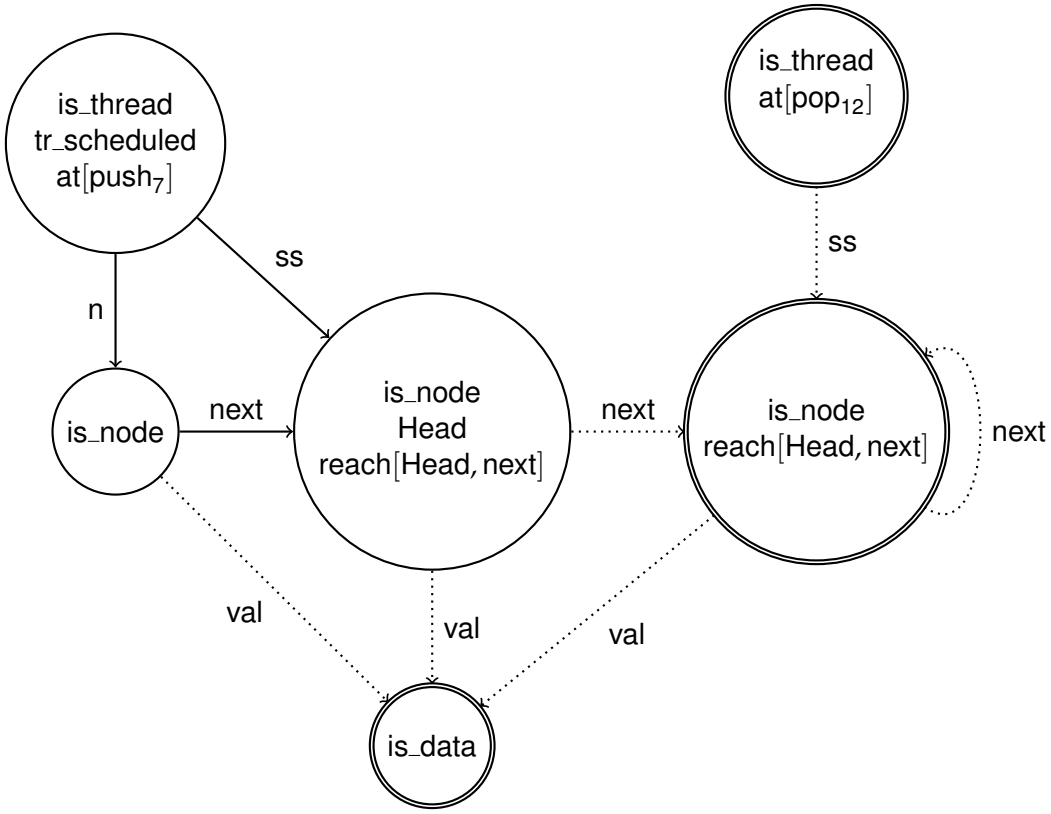


Figure 4.8: Graph of a stack algorithm state with concurrent threads

(and hence thread execution order) is achieved by two additional transitions, which "unschedule" the thread that has just performed a transition by potentially scheduling every thread, and then "schedule" the succeeding thread by focusing. This process is handled implicitly by 3VMC (see Section 3.6.3) or it can be explicitly implemented in a standard TVLA model.

Unschedule

This transition is performed immediately after a thread transition. It sets the value of tr_scheduled to unknown for all threads based on the following update formula:

$$\mathsf{tr\_scheduled}(v) = \mathsf{is\_thread}(v) \wedge \tfrac{1}{2}$$

Schedule

This transition is performed immediately before a thread transition. It performs no update, and performs Focus with the formula $\mathsf{tr\_scheduled}(v)$. The predicate $\mathsf{tr\_scheduled}$ has "unique" integrity rules, which ensure that a single thread is identified to perform the next step.

4.5 Improvements

A number of techniques have been proposed and implemented to improve the efficiency of analyses using canonical abstraction. In this section we describe summary predicates (4.5.1), partially disjunctive analysis (4.5.2), semi-naive evaluation and multi-constraints for Coerce (4.5.3), finite differencing (4.5.4) and graph decomposition (4.5.5).

4.5.1 Summary Predicate

As explained above in Section 4.1, summary objects are represented using the eq equality predicate. However, maintaining a binary predicate in order to represent a unary property is more expensive than necessary. Sagiv et al. [2002] define a unary summary predicate to use instead of eq, which can be defined as an instrumentation predicate:

$$\mathsf{sm}(v) = \neg\, \mathsf{eq}(v, v)$$

In an abstract state, $\mathsf{sm}(u)$ is unknown if u is a summary node, and is false if u is a non-summary node.

However, if sm is used in place of eq, then it cannot be defined as a standard instrumentation predicate to be used with canonical abstraction. In any concrete state it is false for every object — in any canonically abstract state it is therefore false for every object too, though it needs to be unknown to denote the summary objects.

The predicate sm can be used if its behaviour is explicitly specified in the theory. Sagiv et al. [2002] use this explicit extension in all of their definitions and proofs, showing that the summary predicate can be used as a more efficient approach for implementing canonical abstraction. Implementing this theory, the TVLA tool uses sm in place of eq.

4.5.2 Partially Disjunctive Analysis

Manevich et al. [2004] describe an approach for storing abstract states in the statespace analysis that is less precise but in practice just as effective and much more efficient than the original approach of Sagiv et al. [2002]. Rather than comparing to see whether a state is isomorphic to any stored state they compare to see if it is "partially isomorphic" based on the notion of *universe congruence*, which induces an equivalence class on abstract states. Two states are universe congruent if they have the same universe and have the same interpretation over the abstraction predicates.

If a state S_1 is universe congruent with a previously visited state S_2, the two are merged to form a new state S_3, where $U_1 = U_2 = U_3$ and for every k-ary predicate p with objects $u_1, \ldots, u_k$

$$\iota_3(\mathsf{p})(u_1, \ldots, u_k) = \begin{cases} \iota_1(\mathsf{p})(u_1, \ldots, u_k) & \text{if } \iota_1(\mathsf{p})(u_1, \ldots, u_k) = \\ & \quad \iota_2(\mathsf{p})(u_1, \ldots, u_k) \\ \frac{1}{2} & \text{otherwise} \end{cases}$$

I used partially disjunctive analysis in all the results in Chapters 7 and 8.

4.5.3 Coerce

The Coerce operation is expensive, as it has to evaluate all of the many compatibility constraints. Bogudlov et al. [2007a,b] detail several improvements that make Coerce much more efficient.

Most transition updates only affect a small part of the configuration. Borrowing ideas from database theory, Bogudlov et al. define a semi-naive evaluation for Coerce that only evaluates the constraints affected by predicates that could have been changed by Focus or Update.

The compatibility constraints often include groups of formulas that are symmetric, having the same atoms. Bogudlov et al. introduce the concept of a multi-constraint, which can be used to represent such a set of constraints, and can be used to evaluate all the constraints at once, at a comparable cost to evaluating one alone.

These improvements are implemented in TVLA 3.0α, which I used for all the results in Chapters 7 and 8.

4.5.4 Instrumentation Predicate Updates

Defining the update formulas for instrumentation predicates adds an extra amount of manual effort to constructing a model. Reps et al. [2003, 2010] describe an algorithm called 'finite differencing' for automatically computing an update formula for an instrumentation predicate based on the definition formula of the predicate. The update is based on the defining formula of the instrumentation predicate and the update formulas of the predicates used in the definition.

I did not use finite differencing in this thesis — all the instrumentation predicate updates in the analyses reported in Chapters 7 and 8 were constructed by hand.

4.5.5 Graph Decomposition

Manevich et al. [2007, 2008] describe an alternative implementation of canonical abstraction that stores manually specified subgraphs of each state graph, rather than the entire state graph as a whole. This approach exploits the insight that there can be many *independent* parts of a state graph: the graph is decomposed into these parts, which are stored separately. The subgraphs are not disjoint, so together the subgraphs use more space than the full state graph, however large efficiencies can be gained from only storing one copy of subgraphs that are found in multiple states.

I did not use graph decomposition for any of the analyses in this thesis.

Part II

Modelling and Testing

Chapter 5

Model Checking Nonblocking Algorithms

In this chapter we examine how to represent the concepts presented in Chapter 2 — data structures algorithms and their linearisability and nonblocking properties — in the languages and logics of model checkers, as presented in Chapter 3. In Section 5.1 we consider how to model the data structures themselves. In Section 5.2 we consider how to specify linearisability. In Section 5.3 we consider how to specify nonblocking properties.

The approaches described in Sections 5.1 and 5.2 are similar or identical to published approaches by other authors, though I arrived at much of it independently. The novel contributions of this chapter are the formalisations of nonblocking properties in Section 5.3, and the survey presentation of the approaches as a whole.

5.1 Modelling Data Structures

In this section, we will present the representation of nonblocking data structures for model checking that is used through the rest of the thesis. In Section 5.1.1, we discuss transition systems, relating the languages of the model checkers to the underlying Kripke structures. In Section 5.1.2, we discuss how to create finite sized models. In Section 5.1.3, we discuss some approaches for manually reducing the statespace.

5.1.1 Transition Systems

In Section 3.1, we introduced Kripke structures as a formalism for describing systems. In this thesis we do not construct Kripke structures directly — instead we use the language of each model checker to describe a system; internally, the model checker uses this to construct a Kripke structure, or equivalent representation. Effectively, the states are labelled by atomic propositions representing the global variables and the threads' local variables, including program counters. Some modelling languages (e.g. Promela) define the program counters implicitly, whilst others require them to be defined explicitly (e.g. SAL). We will assume, unless noted otherwise, that the program counter is a variable called *location*. Thus, thread t_1 is at location 'idle' when $t_1.location = \mathsf{idle}$.

Each step of a thread does not (necessarily) represent a single transition in the Kripke structure, but represents transitions between any two pairs of states where the first state satisfies the step's precondition — notably considering the value of the program counter — and the second state is the result of applying the step changes to the first.

We say that a step is enabled at a state if the precondition is satisfied. We say that a thread is enabled at a state if at least one of the thread's steps is enabled. Thus, for concurrent data structures each thread is always enabled, unless it is blocked.

One reason threads may be blocked is for a reduction in interleaving. We may place some of the steps within an atomic block, which prevents interleaving of steps from other threads. This behaviour can be implemented implicitly by model checkers in one of two ways. Either, the transitions of the steps are merged into a single transition (operationally, the intermediate states are not stored). Or, other threads are blocked until the end of the atomic block. Promela provides options for both (`d_step` and `atomic`, respectively). TVLA only provides the latter approach to restrict interleaving, but if desired, we can merge steps by manually defining an equivalent step to replace the atomic block.

Later in this chapter we will consider "traces" of a system, a notion from labelled transition systems (which we mentioned in Section 3.1 to be equivalent to Kripke structures). For our purposes here, we will assume that the "output" of the system is recorded by a shared variable *trace*. This variable is updated by every step, containing a (possibly empty) sequence from a fixed set of values. The trace of an execution is the sequence concatenating all of the sequences stored in *trace* along the execution.

For the remainder of the thesis we will not discuss Kripke structures again, and will generally take a pseudocode-centric view of using "transition" interchangeably with "step" to refer to such meta-transitions of the modelling language.

5.1.2 Creating Finite Systems

As mentioned in Chapter 1, model checking is a technique applicable to finite state systems, yet models of nonblocking data structure algorithms have infinite statespaces. In any real implementation there will be physical constraints placed upon the size of the algorithm, however there are no *a priori* limits on the number of threads, the number of distinct data values that can be used, or the amount of memory that can be used for the list structure (i.e. arrays or linked list nodes).

Bounded Parameters

In Chapter 6, I parameterise the algorithms so that each model has a fixed number of threads, data values and memory locations. Every finite instantiation of the parameters produces a finite model, which can be analysed by a model checker.

An alternative approach is to place a bound on the number of operations, which in turn bounds the three parameters above. It also restricts to finite executions, so is not suitable for checking progress properties. Whilst the smaller statespace might make it quicker to find some bugs, if no bugs are found it is harder to draw any general inferences.

Unbounded Abstract Models

In Chapters 7 and 8, I use canonical abstraction (see Chapter 4) to construct an 'equivalent' finite state model, which allows 'unbounded' verification. The models being abstracted have a fixed collection of threads and of data values, and the sizes of these are not necessarily recorded explicitly. The model is initialised with an empty data structure, and additional nodes for the linked list may be created without bound. Again, the number of nodes in a particular concrete state is not necessarily recorded explicitly in the abstract state.

Since the numbers of threads, data values and nodes in a particular abstract state may not be explicitly recorded, the abstract state may represent

an infinite number of concrete states. See Chapter 4 and Sections 7.1–7.2 for more details.

5.1.3 Manual Statespace Reduction

As a consequence of the state explosion problem (see Section 3.4), model checking is very sensitive to the representation of the model — a linear increase in the thread's statespace has an exponential effect on the size of the system's statespace. I have employed two manual techniques to reduce the size of the statespace — resetting unused values and merging transitions.

Resetting Unused Values

Consider the stack algorithm from Figure 2.7 — when a Pop operation detects conflict with other threads at line 17 (if the CAS fails) it restarts the loop. Going through the loop a second (or subsequent) time the thread rereads all of the local variables (*ss*, *ssnext*, *lv*) before using them again, thus it is effectively in the same state as when it begins the operation. However, a model checker does not necessarily see them as the same state — when the operation is begun these variables are initialised as null; when the loop is restarted they contain whatever values were read during the loop. Not only will the model checker separately store states beginning and restarting the loop, it will separately store many versions of states restarting the loop, corresponding to each of the (probably exponentially many) possible values that the local variables can take.

To address this problem I modify the Spin models in Chapter 6 to atomically reset all local variables that are modified in the loop.

I do not make this modification to the TVLA models used in Chapters 7 and 8, as canonical abstraction can represent all such states with a single abstract state.

Reducing Interleaving

In Section 3.4.2, we introduced partial order reduction, an automatic technique that reduces the size of the explored statespace by eliminating thread interleavings that do not affect the property being checked. However, some model checkers do not implement partial order reduction, and the

Type: Node $= \{val : \mathsf{T};\ next : \mathsf{Node}\}$
Shared: $Head : \mathsf{Node} := \text{null}$

```
 1: operation PUSH(lv:T)
 2:     n := new(Node)
 3:     n.val := lv
 4:     repeat
 5:         atomic
 6:             ss := Head
 7:             n.next := ss
 8:         end atomic
 9:     until CAS(Head, ss, n)
10: end operation
```

```
11: operation POP()
12:     repeat
13:         atomic
14:             ss := Head
15:             if ss = null then
16:                 return empty
17:             end if
18:         end atomic
19:         ssnext := ss.next
20:         lv := ss.val
21:     until CAS(Head, ss, ssnext)
22:     ss.next := null
23:     ss.val := null
24:     return lv
25: end operation
```

Figure 5.1: A lock-free stack algorithm with merged transitions

range of possible partial order reduction algorithms that can be implemented have varying effectiveness. Some of the benefits of partial order reduction can be achieved manually by using atomic blocks to prevent unnecessary interleaving.

The linearisability and nonblocking properties of concurrent data structures can be affected by the interleaving of thread steps that read or write to shared variables and objects, but are not affected by the interleaving of thread steps that *only* read or write to local variables and private objects. Thus, we can group such local steps together for efficiency — the reduction in transition interleavings reduces both the time and memory required for analysis.

Figure 5.1 shows the stack algorithm from Figure 2.7 extended with atomic blocks that group transitions to prevent unnecessary interleavings. In each case, the transitions grouped are ones that only read or write to local variables. For the Push operation, the initialisation of private node n at lines 2 and 3 involves only the local variables n and lv, so can be merged. Subsequently, line 7 assigns the local variable ss to a field of the

private node n so can be merged with the previous line that assigns *ss*. Similarly, line 15 of the Pop operation tests the local variable *ss* and can be merged with the previous line that assigns it. Line 19 cannot be merged as well — whilst *ss* is a local variable it refers to a shared node.

5.2 Specifying Linearisability

In this section, we consider how to specify linearisability so that it can be checked by a model checker. In Section 5.2.1, we discuss constructing concurrent specifications from sequential specifications. In Section 5.2.2, we discuss identifying linearisation points for operations. In Section 5.2.3, we discuss the use of simulation relations for determining trace inclusion of the implementation with the concurrent specification. In Section 5.2.4, we discuss checking the trace inclusion directly. In Section 5.2.5, we discuss discuss cases where linearisation points may be determined by the future behaviour of other threads. In Section 5.2.6, we discuss merging the specification and implementation systems together.

5.2.1 Concurrent Specifications

As explained in Section 2.2, a concurrent data structure implementation is linearisable if every history of operation invocations and responses is a permutation (with some ordering constraints) of a history of the sequential specification. One way of determining this is to construct a *concurrent specification* — a system that is linearisable by construction — and show that the set of invocation/response histories generated by the implementation is a subset of the set of histories generated by the concurrent specification. This turns linearisability into a language containment or trace inclusion question.

Each operation of the concurrent specification consists of three transitions: a call step, which invokes the operation; an apply step, which atomically performs the entire operation; and a return step, which performs the operation's response. Figure 5.2 shows a diagram of the structure of the stack concurrent specification's ith thread, and Figure 5.3 enumerates the possible operations, where $\{v_1, \ldots, v_n\}$ is the set of possible data values. From the initial idle state, it nondeterministically invokes either a pop or a push operation; there is only one pop call transition, but there is one push

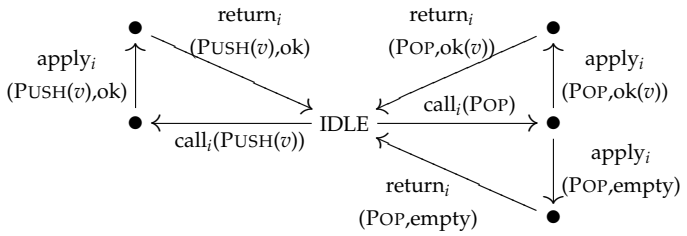


Figure 5.2: Diagram of thread i in the concurrent stack specification

$$
\begin{array}{ccccc}
\mathrm{call}_i(\textsc{Push}(v_1)) & - & \mathrm{apply}_i(\textsc{Push}(v_1), \mathrm{ok}) & - & \mathrm{return}_i(\textsc{Push}(v_1), \mathrm{ok}) \\
\vdots & & \vdots & & \vdots \\
\mathrm{call}_i(\textsc{Push}(v_n)) & - & \mathrm{apply}_i(\textsc{Push}(v_n), \mathrm{ok}) & - & \mathrm{return}_i(\textsc{Push}(v_n), \mathrm{ok}) \\
\mathrm{call}_i(\textsc{Pop}) & - & \mathrm{apply}_i(\textsc{Pop}, \mathrm{empty}) & - & \mathrm{return}_i(\textsc{Pop}, \mathrm{empty}) \\
\mathrm{call}_i(\textsc{Pop}) & - & \mathrm{apply}_i(\textsc{Pop}, \mathrm{ok}(v_1)) & - & \mathrm{return}_i(\textsc{Pop}, \mathrm{ok}(v_1)) \\
\vdots & & \vdots & & \vdots \\
\mathrm{call}_i(\textsc{Pop}) & - & \mathrm{apply}_i(\textsc{Pop}, \mathrm{ok}(v_n)) & - & \mathrm{return}_i(\textsc{Pop}, \mathrm{ok}(v_n))
\end{array}
$$

Figure 5.3: Operations of thread i in the concurrent stack specification

call transition for every possible data value. Then it performs the specification operation with an apply transition. For a push operation, the data value of the apply transition must match that of the preceding call transition. For a pop operation, the apply transition taken — an empty pop, or the pop of a value — depends on the contents of the stack. Finally, it performs a return transition that matches the preceding apply transition, and returns to the idle state.

Concurrent specifications are linearisable — every invocation-response pair can be moved to the position where the operation is performed (the apply step), constructing a correct sequential execution. Thus, the concurrent specification generates every possible linearisable invocation-response history.

For both the concurrent specification and the implementation system, we set *trace* at each transition as follows. For the first transition of an operation, the output is a singleton containing the appropriate invocation (see Section 2.2). For example, the *trace* output of $\mathrm{call}_2(\textsc{Push}(a))$ is $\langle \mathrm{inv}_2(\textsc{Push}, a) \rangle$. For the final transition of an operation, the output is a

singleton containing the appropriate response. For example, the *trace* output of $\text{return}_7(\text{POP}, \text{empty})$ is $\langle \text{resp}_7(\text{POP}, \text{empty}) \rangle$. For all other transitions, the *trace* output is the empty sequence $\langle\rangle$. Then the implementation is linearisable with respect to the specification iff its set of traces is a subset of the set of traces of the concurrent specification.

Determining this trace inclusion is a complex problem, and cannot be verified solely using CTL* model checking — Alur et al. [1996, 2000] have shown that it is in EXPSPACE, whilst CTL* model checking is in PSPACE [Sistla and Clarke, 1985]. Liu et al. [2009] have used FDR-style refinement to allow model checking.

5.2.2 Linearisation Points

One of the principle causes of the complexity of the above approach is the large number of permutation of orderings that have to be considered for every set of concurrent operations. These permutations can be eliminated if we can specify a point in the code where an operation can always be assumed to take effect. This approach of using *linearisation points* requires more human effort but is a tradeoff used in most formal linearisation analyses.

It is not always possible to pick one transition from each operation and say "when this transition is executed it is the linearisation point for its operation." Linearisation points can be more complicated:

- An operation's linearisation point may be a transition of another operation [e.g. Colvin et al., 2006].
- One transition may be the linearisation point for many operations [e.g. Colvin et al., 2006].
- A transition may be the linearisation point for an operation, depending on the *future* behaviour of *other* operations (see Section 5.2.5).

Choosing linearisation points can be quite challenging, as the vast majority of points are not suitable. With automated analyses, incorrect choices will always be identified; with manual analyses there is the risk that a mistake could "prove" linearisability with incorrect linearisation points.[1]

[1] For example, Colvin et al. [2006] explain how one linearisation point choice of Vafeiadis et al. [2006] in their manual analysis of the lazy list set algorithm [Heller et al., 2005, 2007] is incorrect.

We expand the set of values that can be output to *trace* to include linearisation points, of the form $\mathrm{lp}_p(\mathrm{OP}, val^*)$, congruent with the invocations and responses (see Section 2.2). These values are output at the linearisation point transitions of the implementation system and the apply transitions of the concurrent specification system; the latter will only write singleton sequences to *trace*, but the implementation system transitions may write longer sequences.

Including linearisation points in traces simplifies the analysis, as it determines exactly one sequential ordering for each trace, though it does require further effort to refute linearisability. If a particular implementation trace is shown to not be a specification trace, it does not necessarily follow that the execution and system are not linearisable — the chosen linearisation points may be incorrect, so the trace must be searched for other linearisable orderings.

5.2.3 Simulation

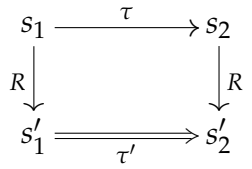
One approach to verifying trace inclusion, and thus linearisability, is to show that there is a *simulation relation* [Lynch and Vaandrager, 1995] between the implementation and specification. There are two types of simulations, for reasoning forwards and backwards through the statespace. A relation R between the states of an implementation and a specification system is a *forward* (or *downward*) *simulation* iff

- every initial state of the implementation is related to an initial state of the specification; and
- given a transition τ between two implementation states s_1 and s_2, and a specification state s'_1 related to s_1, i.e.

$$\begin{array}{ccc} s_1 & \xrightarrow{\quad\tau\quad} & s_2 \\ R \big\downarrow & & \\ s'_1 & & \end{array}$$

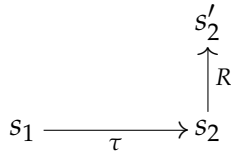
  then there is a specification state s'_2 related to s_2, which is reachable from s'_1 by a sequence τ' of 0 or more specification transitions that has the same trace (of invocations, linearisation points and responses) as

the implementation transition, i.e.

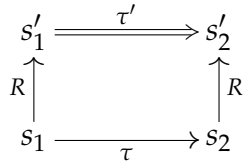


Alternatively, R is a *backward* (or *upward*) *simulation* iff

- every implementation state is related to a specification state;
- every specification state related to an implementation initial state is a specification initial state; and
- given a transition τ between two implementation states s_1 and s_2, and a specification state s'_2 related to s_2, i.e.



then there is a specification state s'_1 related to s_1, which can reach s'_2 via a sequence τ' of 0 or more specification transitions that has the same trace (of invocations, linearisation points and responses) as the implementation transition, i.e.



It is important to note that the "states" referred to in the above definitions are specifically the *reachable states*, not all possible states generated by permutations of values that the state variables can take.

Neither forward nor backward simulation alone is complete for showing trace inclusion — it may be necessary to construct an intermediate system and show a forward simulation from the implementation to the intermediate system and a backward simulation from the intermediate to the specification system [He et al., 1986]. For nonblocking data structures forward simulation is not sufficient in cases where a linearisation point depends upon future behaviour of other threads, such as a non-empty Dequeue for the queue algorithm presented in Section 2.6.2.

Use of Simulation

A number of algorithms have been verified using the theorem prover PVS [Owre et al., 1998] by representing the implementation and specification systems as input/output automata [Lynch, 1996] and showing that there is a simulation relation between them [Doherty et al., 2004b; Colvin and Groves, 2005; Colvin et al., 2005, 2006].

Hesselink [2007] describes an alternative approach, also using PVS. Derrick et al. [2007, 2008, 2011a,b] have used a different proof approach, using the theorem prover KIV [Balser et al., 1998], to show simulation relations (and hence linearisability) between Z systems [Spivey, 1992].

Smith and Derrick [2005, 2006] have considered model checking simulations of Z systems.[2] Their approach encodes the simulation conditions as relatively large CTL formulas (quantification over transitions must be explicitly expanded as CTL is a propositional logic). Rather than initialising the model to initial states and exploring the combined statespace from there, they initialise the model to any states of the implementation and specification systems and explore only two transitions deep to check the simulation conditions. This has the advantage of keeping the depth of the analysis shallow, but the breadth of initialising to every possible state is inefficient. Smith and Derrick [2006] acknowledge that they have not yet investigated any optimisations that would make the approach more widely applicable. They do not give results (e.g. memory or time) for the small example they use, but it does appear that further optimisations would be needed for the approach to be practically applicable to nonblocking data structures.

5.2.4 Direct Trace Inclusion

Using simulation to show trace inclusion involves an extra layer of manual work to define the relation R. Additionally, the relations required are relatively verbose[3] — the complexity of model checking is dependent on

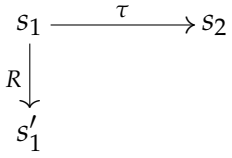
[2] Their simulation definitions differ slightly from those presented here, primarily by including an extra condition relating when implementation and specification transitions are enabled. We assume that transitions are always enabled, so this is not necessary. They also specify that an implementation transition is matched by a single specification transition (rather than a sequence) and by default do not restrict to reachable states (though they do mention this issue in discussion).

[3] Colvin et al. [2005] explain in detail the forward simulation relation they use to verify the stack from Section 2.6.1.

the size of the formula (see Section 3.2.4) and the approach of Smith and Derrick [2005, 2006] requires R to be repeated several times in the CTL formulas.

For model checking nonblocking data structures we can avoid this additional complexity of defining a simulation relation if we ensure that at every state in the concurrent specification, and for each output value, there is at most one enabled transition that can write that value to *trace*. If this is the case, then when trace inclusion holds, there is a single execution of the specification system to match each execution of the implementation system.

Simulation needs to account for nondeterminism in the trace. For example, when we have the following situation:



there could be two specification transition sequences $s'_1 \stackrel{\tau'_1}{\Longrightarrow} s'_2$ and $s'_1 \stackrel{\tau'_2}{\Longrightarrow} s'_3$ with the same trace as the implementation transition, where $R(s_2, s'_3)$ holds but $R(s_2, s'_2)$ does not. Potentially, both must be checked, which is why Smith and Derrick [2005, 2006] use CTL's existential path quantifiers.

Concurrent specifications would exhibit this behaviour too if we had defined the values for *trace* more simply. For example, if the invocations, linearisation points and responses did not contain the thread index or data value, then e.g. the transitions $\text{call}_1(\text{PUSH}(a))$, $\text{call}_2(\text{PUSH}(a))$, and $\text{call}_2(\text{PUSH}(b))$ would all have the same output to *trace*: $\text{inv}(\text{PUSH})$. There are other possible ways that a concurrent specification could have two enabled transitions with the same *trace* output, such as a nondeterministic choice of 'new' nodes. We restrict our analyses to specifications where such nondeterminism does not occur, or where it can be replaced with a deterministic choice.

Now, the above observation — that each execution of the implementation has at most one concurrent execution with the same trace — means that we can check trace inclusion by using the trace of the implementation as input to the specification, determining which transition will be performed at each point in time. If trace inclusion holds, then the specification will always be able to perform transitions to match the trace output that

it receives as input from the implementation. Otherwise, if trace inclusion does not hold, then there will be a state where the specification is not able to perform a transition that produces the same trace as it received as input.

We combine the implementation and specification systems into a new system. The implementation transitions write to *trace* as before, but the preconditions are modified so that the transitions are only enabled when *trace* is empty (i.e. contains $\langle\rangle$). The specification transitions' preconditions are modified to only be enabled when the sequence in *trace* is not empty, and the first item matches the value that output to *trace* in the standalone specification. In the combined system, *trace* is updated by removing the first item in the sequence.

There are two ways to detect a failure of trace inclusion. As described so far, the system will halt if the specification receives an input from *trace* that it can't match; some model checking tools have efficient deadlock detectors so this may be preferable. Alternatively, an additional specification transition can be defined that is enabled only if none of the others are. We add an auxiliary boolean variable *noerror*, which is initialised to true, and then set to false by this transition. Then the combined system can be checked against the LTL/CTL formula **AG** *noerror*.

5.2.5 Future Nondeterminism

The approach outlined above in Section 5.2.4 is analogous to forward simulation; it does not account for cases where backward simulation is needed, i.e. when the decision to output a linearisation point to *trace* depends upon the future behaviour of other threads. I have explored three approaches for addressing these cases: backward analysis analogous to backward simulation, which has a number of difficulties; prophesy variables, which are an easier approach for model checking; and allowing more than one linearisation point for some operations.

Backwards analysis

One possible approach to handling future nondeterminism is to apply the same approach described in Section 5.2.4 but to examine the statespace in reverse. For each transition τ in both the implementation and specification systems, we construct a transition τ_r, such that $s_2 \xrightarrow{\tau_r} s_1$ iff $s_1 \xrightarrow{\tau} s_2$. In the combined system the (reverse) implementation outputs the trace of

each transition to the (reverse) specification, which performs the unique sequence of transitions that generate the same trace.

Constructing the reverse transition semantics is an extra layer of manual work, though it could conceivably be automated. The process is not immediately straightforward, as a naive approach would allow unreachable states that cause the reverse implementation system to deadlock (particularly important for some model checking algorithms that require the transition relation to be total). This happens because the reverse systems are not nonblocking — it is possible for a thread to become blocked, waiting for actions of other threads to modify the shared memory. For example, in the stack algorithm from Figure 2.7, a thread can perform a response transition for a Push of any value whenever it is in the idle state, but it can then perform the linearisation point CAS at line 7 only when the stack is non-empty and the head node's value matches the value from the previous response transition. If the stack is in some other configuration then the thread is blocked, waiting for other threads to Push and Pop operations until the stack is in an appropriate configuration for it to proceed. Consider then a state where the stack is empty and all threads are idle — if every thread performs a Push response transition then each one will be blocked and the system will deadlock as there is no other thread to perform a dequeue and make the stack non-empty. I addressed this problem by defining auxiliary boolean variables that recorded whether there was at least one other thread in an unblocked state — transitions that could make a thread become blocked were modified to only be enabled when the thread would not become blocked (e.g. for a Push response if the value matched the head node's value) or the auxiliary variables indicated that there was at least one other unblocked thread.

An important aspect of this approach is that the backward analysis must cover all of the (forward) reachable states. For a forward analysis the (forward) reachable states are computed by constructing the transition graph from the initial state(s). For a backward analysis there is no comparable final state(s) to begin the transition graph from, and the set of (forward) reachable states is not easily computed by any other means. I used the same initial states for the forward and backward systems, an approach that is only complete if the sets of backward reachable and forward reachable states are the same. This is the case for many nonblocking data structures — there is usually a sequence of Pop / Dequeue operations that will return the system to the initial state, which has an empty list and idle threads. However, it does exclude linked-list based implementations that

do not free memory to the system — the set of backward reachable states (where no memory is allocated, i.e. there have been no Push / Enqueue operations) is a small subset of the forward reachable states. This property can be determined by examining the (forward) implementation to see whether every (forward) reachable state can return to the initial state, i.e. model checking the CTL formula

AGEF *init*

where *init* is an auxiliary boolean variable that is only true in the initial state.

As is the case for simulation, a forward or backward analysis alone may not be sufficient for verifying trace inclusion — it may be necessary to define an intermediate system and show trace inclusion of the implementation by the intermediate system using a forward analysis and trace inclusion of the intermediate system by the specification using a backward analysis. The definition of an intermediate system is another source of manual effort, which requires a detailed understanding of the implementation algorithm.

I attempted to implement this approach in SAL to investigate its practicality, checking trace inclusion of the intermediate queue system that Doherty et al. [2004b] defined and a concurrent queue specification. The preliminary implementation appeared promising but given the above mentioned extra costs of manual work required, and the availability of an alternative approach, I did not pursue this approach any further.

Prophecy variables

An alternative technique for dealing with future nondeterminism is to bring the nondeterministic choice forward by using *prophecy variables*, first described by Abadi and Lamport [1991]. These auxiliary variables record information about a future event, and the transitions are modified to ensure that the "prediction" comes true.

Example As described in Section 2.6.2, the queue algorithms from Figure 2.8 have a future nondeterministic choice at the linearisation point for an empty Dequeue operation; let us consider the second algorithm, from Figure 2.8c. The linearisation point occurs when the snapshot *ssnext* is read at line 43 — more specifically, at the last such occurrence if the loop is repeated. However, it is not possible when executing that step to know

whether the loop will be repeated again, as other threads may modify *Head* before it is checked against *sshead* at the next step. Using prophecy variables we can nondeterministically make the decision when reading *ssnext* whether it is a linearisation point or not. If the step is decided to be a linearisation point, we ensure that no other thread modifies *Head* until it is tested against *sshead*. Otherwise the thread waits for another thread to modify *Head* and then restarts the loop. Thus we need to record three possibilities:

- the potential linearisation point has not been reached;
- the point has been reached and is not a linearisation point, so the loop will be repeated; or
- the point has been reached and is a linearisation point, so the loop will not be repeated.

Figure 5.4 contains the Dequeue operation from Figure 2.8c, augmented with two boolean prophecy variables. The variables are initialised to false at the start of the loop (lines 3 and 4). At the point where *ssnext* is read (lines 6–14), and it is null, a nondeterministic choice is made as to whether it is a linearisation point or not — if so then *observedEmptyWontLoop* is set to true, and if not then *observedEmptyWillLoop* is set to true. In the latter case the loop must be restarted so at line 15 the thread blocks until *Head* is modified.[4] In the former case, the loop must not be restarted, so other threads must be prevented from modifying *Head*. This is achieved by blocking the steps that modify *Head* — the successful CAS step at line 22 and the successful CAS in the Enqueue operation (not shown) — when another thread has *observedEmptyWontLoop* true. After performing the test at line 16, *observedEmptyWontLoop* is set to false as it is not possible for a thread that has observed an empty queue to restart the loop from here, and it is not necessary to block other threads from modifying *Head*. ■

Prophecy variables are a practical approach for model checking nonblocking data structures with linearisation points that depend on the future behaviour of other threads. Their implementation does require some manual work, but less than constructing a backward analysis.

[4]The pseudocode "**when** ϕ τ" means that transition τ is only enabled when formula ϕ is true.

```
1:  operation DEQUEUE()
2:      loop
3:          observedEmptyWontLoop := false
4:          observedEmptyWillLoop := false
5:          sshead := Head
6:          atomic
7:              ssnext := sshead.next
8:              if sshead = Head then
9:                  either
10:                     observedEmptyWontLoop := true                    // LP
11:                 or
12:                     observedEmptyWillLoop := true                    // not LP
13:             end if
14:         end atomic
15:         when ¬ (observedEmptyWillLoop ∧ Head = sshead)
16:             if sshead = Head then
17:                 observedEmptyWontLoop := false
18:                 if ssnext = null then
19:                     return empty
20:                 else
21:                     lv := ssnext.val
22:                     when  Head ≠ sshead ∨
                                  ¬ ∃ t • t.observedEmptyWontLoop
23:                         if CAS(Head, sshead, ssnext) then
24:                             ssnext.val := null
25:                             sstail := Tail
26:                             if sshead = sstail then
27:                                 CAS(Tail, sstail, ssnext)
28:                             end if
29:                             break
30:                         end if
31:                 end if
32:             end if
33:         return lv
34:     end loop
35: end operation
```

Figure 5.4: A lock-free linked list based queue with prophecy variables

Multiple Linearisation Points

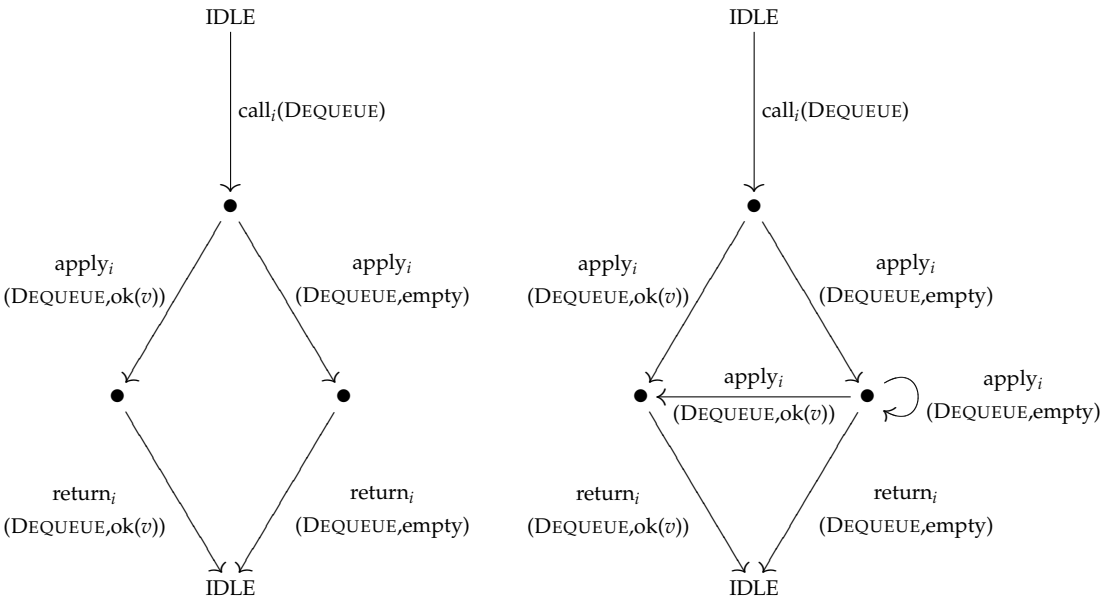
Some data structure operations, such as a push, modify the shared memory; others, such as an empty pop, have no global side effects. Both possibilities exist for operations that have future nondeterminism when deciding the linearisation point — they may modify shared memory [e.g. Harris et al., 2002; Hendler et al., 2004] or may, like the empty dequeue of the example queues, have no side effects.

In the cases where there are no side effects, we can avoid the need to use prophecy variables by relaxing the constraint in the specification that there be exactly one linearisation point for these operations. We mark the potential linearisation point every time it occurs, which allows it to occur more than once within the operation, and we allow the specification to repeat the application of the operation multiple times. Repeating a linearisation point that modifies the shared memory would have the incorrect effect of performing the operation twice, but this is not a concern for operations that don't modify shared memory — it simply provides several points where the operation can be linearised, each of which is a valid choice.

Example The future nondeterminism in the queue explored in the previous example (originally described in Section 2.6.2) occurs when deciding the linearisation point for an empty Dequeue — an operation that does not modify the shared memory. Using multiple linearisation points in the implementation is straightforward — we mark the step where *ssnext* is read (line 43 in Figure 2.8c) as the linearisation point when it is null, ignoring the possibility that it may occur several times during the operation.

Figure 5.5a shows a diagram of the original Dequeue steps of thread i in the concurrent queue specification (compare with the concurrent stack specification in Figure 5.2).[5] Figure 5.5b shows a diagram of the modified version to allow multiple linearisation points for an empty Dequeue operation. Since the implementation is able to repeat the loop and output the linearisation point to the trace more than once during and during an empty Dequeue operation, the specification must be able to loop, performing additional apply_i(dequeue,empty) transitions before the return transition. Also, the implementation is able to perform the empty Dequeue linearisation point and then after restarting the loop perform

[5]The two "IDLE" states in each diagram are the same state repeated for clarity.



(a) Original (b) With multiple linearisation points

Figure 5.5: Diagrams of Dequeue operation of Thread i of a concurrent stack specification

a non-empty Dequeue; thus the specification must be able to perform apply$_i$(DEQUEUE,ok(v)) after performing apply$_i$(DEQUEUE,empty).

The modified concurrent specification can be simply argued to be linearisable. There is exactly one apply$_i$(ENQUEUE(v),ok) step in an (unmodified) Enqueue operation and exactly one apply$_i$(DEQUEUE,ok(v)) step in a non-empty Dequeue operation; these are the linearisation points. A non-empty Dequeue operation has at least one apply$_i$(DEQUEUE,empty) step and can be linearised at any of these, as they each observe an empty queue. ■

I applied prophecy variables successfully in preliminary work, but I use multiple linearisation points for the models constructed in Chapters 6–8, as the future nondeterminism in the algorithms considered occurs only in operations with no global side effects.

5.2.6 Merging the Specification

In the combined systems for checking trace inclusion, described in Sections 5.2.4 and 5.2.5, the operations in the specification system perform their invocation, linearisation point and response actions immediately after the corresponding action in the implementation system. We make the observation that effectively the specification system is ensuring that there is exactly one sequential specification operation performed during each implementation operation (or at least one, for side-effect free operations when using the approach in the last part of Section 5.2.5). We can achieve the same result by merging the specification system into the implementation system, gaining increased efficiency by removing extra overhead.

Since the specification invocations and responses make no modifications, the principal result is that the apply / linearisation point actions (which are the sequential specification operations) are performed atomically at the linearisation points of the merged system. To ensure that exactly one linearisation point is performed per operation we introduce an auxiliary Boolean variable *doneLP* for each thread, which is set to false at each invocation and true at each linearisation point. The system enters an error state if it is already true at a linearisation point (unless multiple linearisation points are allowed), or if it is false at a response.

Example Figure 5.6 contains the stack algorithm from Figure 2.7 with the concurrent specification merged in. A notable difference in display is that the conditional CAS steps have been expanded within atomic blocks to allow the specification steps to be combined (Push lines 8–16; Pop lines 37–45). To accommodate this the **repeat**. . . **until** structure has been rearranged into a **loop**. . . **break** structure.

In the Push operation, the auxiliary variable *doneLP* is initialised to false (line 2) and then checked to be true at the end (line 18). The checks that may result in an error state are represented as assertions — these can be expressed directly in some model checker languages (e.g. Promela) but in other languages (e.g. SAL) it will be necessary to define separate error transitions as described in Section 5.2.4.

At the Push linearisation point (lines 8–16), the successful CAS is performed (lines 9–10) and then *doneLP* is checked to be false (line 11) before being set to true (line 12). Finally a Push operation is performed on the specification stack with the same value that was used in the implementation (line 13).

```
 1: operation PUSH(lv)
 2:     doneLP := false
 3:     n := new(Node)
 4:     n.val := lv
 5:     loop
 6:         ss := Head
 7:         n.next := ss
 8:         atomic
 9:             if Head = ss then
10:                 Head = n
11:                 assert(¬ doneLP)
12:                 doneLP := true
13:                 SPECPUSH(lv)
14:                 break
15:             end if
16:         end atomic
17:     end loop
18:     assert(doneLP)
19: end operation
20: operation POP( )
21:     doneLP := false
22:     loop
23:         atomic
24:             ss := Head
25:             if ss = null then
26:                 assert(¬ doneLP)
27:                 doneLP := true
28:                 assert(SPECISEMPTY())
29:             end if
30:         end atomic
31:         if ss = null then
32:             assert(doneLP ∧ lv = null)
33:             return empty
34:         end if
35:         ssnext := ss.next
36:         lv := ss.val
37:         atomic
38:             if Head = ss then
39:                 Head := ssnext
40:                 assert(¬ doneLP)
41:                 doneLP := true
42:                 assert(lv = SPECPOP())
43:                 break
44:             end if
45:         end atomic
46:         lv := null
47:     end loop
48:     ss.next := null
49:     ss.val := null
50:     assert(doneLP ∧ lv ≠ null)
51:     return lv
52: end operation
```

Figure 5.6: A lock-free stack algorithm with merged specification

The Pop operation begins by initialising *doneLP* to false. The two linearisation points (lines 23–30 and 37–45) both perform the implementation steps, assert that *doneLP* is false, set *doneLP* to true and then perform the specification operation. In both cases the result of the specification operation must be checked to make sure that it matches that of the implementation. Immediately before the response steps, *doneLP* is checked to be true (lines 32 and 50), ensuring that a linearisation point has been performed, and the value of *lv* is checked to ensure that the linearisation point (empty or non-empty) matches the response. ■

As well as ensuring that there is exactly one (or at least one) linearisation point per operation, we must ensure that the linearisation point of the implementation matches both the specification operation and the implementation response, in terms of the types of operation and the value consumed or returned. In the previous example we were able to do this using the local thread variable *lv*, but in general it may be necessary to use additional auxiliary variables.

Example Figure 5.7 contains the queue algorithm from Figures 2.8a and 2.8b, with the Dequeue operation modified to allow multiple empty linearisation points, and with the concurrent specification merged in.

The Enqueue operation in Figure 5.7a is modified similarly to the stack Push operation in Figure 5.6. The auxiliary variable *doneLP* is initialised to false and checked to be true at the end. The linearisation point is a conditional CAS step, which is expanded within an atomic block that checks that *doneLP* is false, sets *doneLP* to true and performs the specification Enqueue operation.

The Dequeue operation in Figure 5.7b has additional modifications — because it allows multiple empty dequeue linearisation points we cannot use the approach from the stack Pop operation of using the variable *lv* to record whether a past linearisation point was empty or non-empty. Instead, we use the auxiliary variable *doneLP* to record non-empty linearisation points and introduce another one called *doneELP* to record empty linearisation points; both are initialised to false.

For an empty Dequeue, the linearisation point (lines 32–39) first checks that *doneLP* is false, as it is not possible in the specification (see Figure 5.5) to perform an empty linearisation point after performing a non-empty one. The variable *doneELP* does not need to be checked as we allow multiple linearisation points to be performed. Then *doneELP* is set to true and

```
 1: operation ENQUEUE(lv:T)
 2:     doneLP := false
 3:     n := new(Node)
 4:     n.val := lv
 5:     loop
 6:         sstail := Tail
 7:         ssnext := sstail.next
 8:         if sstail = Tail then
 9:             if ssnext = null then
10:                 atomic
11:                     if sstail.next = ssnext then
12:                         sstail.next := n
13:                         assert(¬ doneLP)
14:                         doneLP := true
15:                         SPECENQUEUE(lv)
16:                         break
17:                     end if
18:                 end atomic
19:             else
20:                 CAS(Tail, sstail, ssnext)
21:             end if
22:         end if
23:     end loop
24:     CAS(Tail, sstail, n)
25:     assert(doneLP)
26: end operation
```

(a) Enqueue operation

Figure 5.7: A lock-free queue algorithm with merged specification

```
27: operation DEQUEUE()
28:     doneLP, doneELP := false
29:     loop
30:         sshead := Head
31:         sstail := Tail
32:         atomic
33:             ssnext := sshead.next
34:             if ssnext = null then
35:                 assert(¬ doneLP)
36:                 doneELP := true
37:                 assert(SPECISEMPTY())
38:             end if
39:         end atomic
40:         if sshead = Head then
41:             if sshead = sstail then
42:                 if ssnext = null then
43:                     assert(doneELP ∧ ¬ doneLP)
44:                     return empty
45:                 end if
46:                 CAS(Tail, sstail, ssnext)
47:             else
48:                 lv := ssnext.val
49:                 atomic
50:                     if Head = sshead then
51:                         Head := ssnext
52:                         assert(¬ doneLP)
53:                         doneLP := true
54:                         assert(lv = SPECDEQUEUE())
55:                         break
56:                     end if
57:                 end atomic
58:             end if
59:         end if
60:     end loop
61:     assert(doneLP)
62:     return lv
63: end operation
```

(b) Original dequeue operation

Figure 5.7: A lock-free queue algorithm with merged specification

```
64: operation DEQUEUE()
65:     doneLP, doneELP := false
66:     loop
67:         sshead := Head
68:         atomic
69:             ssnext := sshead.next
70:             if ssnext = null then
71:                 assert(¬ doneLP)
72:                 doneELP := true
73:                 assert(SPECISEMPTY())
74:             end if
75:         end atomic
76:         if sshead = Head then
77:             if ssnext = null then
78:                 assert(doneELP ∧ ¬ doneLP)
79:                 return empty
80:             else
81:                 lv := ssnext.val
82:                 atomic
83:                     if Head = sshead then
84:                         Head := ssnext
85:                         assert(¬ doneLP)
86:                         doneLP := true
87:                         assert(lv = SPECDEQUEUE())
88:                         break
89:                     end if
90:                 end atomic
91:             end if
92:         end if
93:     end loop
94:     sstail := Tail
95:     if sshead = sstail then
96:         CAS(Tail, sstail, ssnext)
97:     end if
98:     assert(doneLP)
99:     return lv
100: end operation
```

(c) Simplified dequeue operation

Figure 5.7: A lock-free queue algorithm with merged specification

the empty Dequeue specification operation is performed. Immediately before the empty response (line 43) we check that *doneELP* is true, ensuring that at least one empty linearisation point has been performed, and that *doneLP* is false, ensuring that no non-empty linearisation point has been performed.

For a non-empty Dequeue, the linearisation point (lines 49–57), as for the Pop operation in Figure 5.6, checks that *doneLP* is false before setting it to true and performing the specification Dequeue operation, checking that the value returned matches that for the implementation. At the end of the operation *doneLP* is checked to be true (line 61). These checks ensure that exactly one non-empty linearisation point is performed during the operation. The value of *doneELP* does not need to be checked because an empty linearisation point is allowed to occur before a non-empty one, and the check at line 35 ensures that an empty linearisation point cannot occur after a non-empty one. ■

This same approach was taken by Vafeiadis [2008, 2009]. A similar approach was taken by Gao and Hesselink [2004], though they counted the number of linearisation points performed in each operation, checking that it totalled 1 at the response — a strategy that does not catch erroneous nonterminating operations.

5.3 Specifying Nonblocking Properties

In Section 2.4 we presented the three nonblocking properties wait-, lock- and obstruction-freedom, and defined them informally with natural language descriptions. In this section we formalise these definitions using the logics LTL and CTL, and compare with the formalisations of other authors.

5.3.1 Wait-freedom

Recall that an algorithm is wait-free if every thread is able to complete its operation in a finite number of (its own) steps. The core of this property can be expressed as

> if a thread is performing an operation, then it will eventually complete it.

A thread's location is "idle" iff it is not performing an operation, so we can formalise this as

$$t.location \neq \mathsf{idle} \rightarrow \mathbf{F}(t.location = \mathsf{idle})$$

We expect this property to hold in every state of the algorithm and for every thread. Thus we construct the following (quantified) LTL formula:

$$\mathbf{AG}(\forall\, i \bullet t_i.location \neq \mathsf{idle} \rightarrow \mathbf{F}(t_i.location = \mathsf{idle}))$$

To restrict this to propositional LTL we could explicitly expand the universal quantifier, though only if there is a finite bound on the number of threads. However, noting that the threads are fully symmetric, we can check the property for one thread only. Without loss of generality we choose the first thread:

$$\mathbf{AG}(t_1.location \neq \mathsf{idle} \rightarrow \mathbf{F}(t_1.location = \mathsf{idle})) \qquad \text{(WF-1)}$$

One important part of the definition is easy to miss — "of *its own* steps". An execution is a counterexample to wait-freedom when a thread takes an infinite number of steps and does not complete an operation; an (infinite) execution in which a thread takes only a finite number of steps and does not complete an operation does not (necessarily) violate the property. Thus, for model checking wait-freedom we must restrict analysis to executions where the thread or threads take an infinite number of steps, otherwise checking the formula WF-1 will capture spurious counterexamples.

One approach to restricting the analysis is to impose *weak fairness* assumptions [Manna and Pnueli, 1991]. These do not allow executions where threads that can take steps do not, thus t_1 will only stop executing indefinitely if it is deadlocked. Some model checkers have built-in options to restrict analysis to weak fairness executions, but some do not. If not, we must alter the formula.

Without weak fairness assumptions, there are three possible futures from a state where t_1 is not idle: either it will eventually complete its operation (consistent with wait-freedom), or it will never return to idle; in the latter case, t_1 will either take an infinite number of steps (a violation of wait-freedom) or take a finite number of steps (consistent with wait-freedom). By defining an auxiliary variable called *last_step* to record which thread performed the immediately preceding step, then these futures can be specified by the following formulas:

- $\mathbf{F}(t_1.location = \mathsf{idle})$
- $\mathbf{G}(t_1.location \neq \mathsf{idle}) \wedge \mathbf{GF}(last\_step = t_1)$
- $\mathbf{G}(t_1.location \neq \mathsf{idle}) \wedge \mathbf{FG}(last\_step \neq t_1)$

Possibly t_1 completes its operation, which is the desired wait-free behaviour. Possibly t_1 never completes its operation despite taking an infinite number of steps, which is not wait-free behaviour. Alternatively, t_1 may never complete its operation as it only takes a finite number of steps along the infinite path; this is still classed as wait-free behaviour. Thus the wait-free future behaviour can be described as

$$\mathbf{F}(t_1.location = \mathsf{idle}) \vee \mathbf{FG}(last\_step \neq t_1)$$

Note that the *last_step* auxiliary variable increases the statespace by recording which of the n threads lead to each state. Since we are only concerned whether the last step was performed by t_1 or not, it is much more efficient to use an auxiliary boolean variable *lastStepTOne* instead. Thus the full wait-free property is expressed in LTL as:

$$\mathbf{AG}(t_1.location \neq \mathsf{idle} \rightarrow (\mathbf{F}(t_1.location = \mathsf{idle}) \vee \mathbf{FG}(\neg\, lastStepTOne))) \quad \text{(WF-2)}$$

5.3.2 Lock-freedom

Recall that an algorithm is lock-free if some thread always completes an operation after a finite number of steps of the system. The core of this property is the completion of an operation, which can be specified by comparing the locations of each thread in successive states:

$$\exists\, i \bullet t_i.location \neq \mathsf{idle} \wedge \mathbf{X}(t_i.location = \mathsf{idle})$$

However, we would have to explicitly expand the quantifier, as discussed in the previous section. Instead, we can introduce an auxiliary boolean variable *was_response* that is set to true in response steps and false in all others. Now we can very compactly specify lock-freedom:

$$\mathbf{AGF}\, was\_response \quad \text{(LF-1)}$$

Thread-level view

The formula LF-1 specifies lock-freedom as a property of the system, but it is also possible to specify it as a property of individual threads. Though it yields a more complicated formula, there are two motivations for doing so. First, it allows a more direct comparison with wait-freedom, which is also specified as a property of individual threads. Second, for the pragmatic reason that the formula LF-1 relies on the implicit assumption that there are a finite number of active threads. Note that if there are an infinite number of active threads then it is possible for each one to take only a single (e.g. invocation) step during an infinite path, so that no operation is ever completed. This assumption is reasonable, as such systems do not exist in practice; however, in canonical abstraction, a summary thread object may represent an infinite number of threads, so this assumption is lost.

Lock-freedom sacrifices wait-freedom's individual thread guarantees of progress/completion for a system guarantee of progress. Thus a thread in a lock-free algorithm generally proceeds unless it detects conflict and retries. In order for the system guarantee to hold, the conflicting thread must have completed, or be able to complete its operation — otherwise two threads could mutually detect conflict and repeat forever, with neither completing an operation. Thus we can divide an operation into two parts — the second part is wait-free, whilst the first part is only prevented from transitioning to the second part if other threads make the transition infinitely often.

An examination of the algorithms in Section 2.6 shows that the transitions occur at the linearisation points of the operations. For the stack in Figure 2.7 both operations may loop an infinite number of times, but the linearisation points for a push (line 7) and a non-empty pop (line 17) are the successful CAS steps that exit the loop, which are followed by 0 and 2 steps (respectively) before the operation completes. Comparatively, the linearisation point for an empty pop (line 11) only occurs when *Head* is null, so there are only two steps (lines 12–13) before the operation completes. Thus each operation is wait-free from the linearisation point. Before the linearisation points, the loop can only be restarted when the CAS steps fail, which can only happen because successful CAS steps (i.e. linearisation points) of other threads have modified *Head*. Thus the operations loop infinitely only if there are infinite linearisation points performed by other threads.

We can reuse the auxiliary local boolean variable *doneLP* introduced in

Section 5.2.6, which is initialised to false and set to true at the linearisation points. Using this we can specify that the operations are wait-free after the linearisation points:[6]

$$\mathbf{AG}(t_1.location \neq \mathsf{idle} \wedge t_1.doneLP \rightarrow \mathbf{F}(t_1.location = \mathsf{idle})) \quad \text{(LF-2.1)}$$

We introduce a global auxiliary boolean variable *wasLP* that is set to true at steps that are linearisation points, and false at others. This allows us to express that if an operation has not reached its linearisation point it will either do so in the future, or others will an infinite number of times:

$$\mathbf{AG}(t_1.location \neq \mathsf{idle} \wedge \neg\, t_1.doneLP \rightarrow \mathbf{F}(t_1.doneLP) \vee \mathbf{GF}(wasLP)) \quad \text{(LF-2.2)}$$

As with the formula WF-1, these formulas implicitly rely on weak fairness of the execution. This can be addressed explicitly, as was done for WF-2:

$$\mathbf{AG}(t_1.location \neq \mathsf{idle} \wedge t_1.doneLP \rightarrow \mathbf{F}(t_1.location = \mathsf{idle}) \vee \mathbf{FG}(\neg\, lastStepTOne)) \quad \text{(LF-3.1)}$$

$$\mathbf{AG}(t_1.location \neq \mathsf{idle} \wedge \neg\, t_1.doneLP \rightarrow \mathbf{F}(t_1.doneLP) \vee \mathbf{GF}(wasLP) \vee \mathbf{FG}(\neg\, lastStepTOne)) \quad \text{(LF-3.2)}$$

These pairs of formulas imply lock-freedom. They have the disadvantage of being longer than LF-1 — and hence more expensive to verify — so they are only preferable when the assumption of finite threads cannot be relied upon. They also have the disadvantage of requiring linearisation points to be determined, though this is required for the specification of linearisability we have suggested, so is no further effort if linearisability has already been verified.

5.3.3 Obstruction-freedom

Recall that an algorithm is obstruction-free if any operation is able to complete in a finite number of steps if run in isolation. This means that at any state there is a possible execution path for each non-idle thread where only

[6] As in the previous section, these are properties of *all* threads, but due to the thread symmetry we are able to choose just one (the first: t_1).

that thread takes steps until its operation is complete. This is expressed by the following CTL formula:

$$\mathbf{AG}(t_1.location \neq \mathsf{idle} \rightarrow \mathbf{E}[lastStepTOne\,\mathbf{U}\,t_1.location = \mathsf{idle}]) \quad \text{(OF-1)}$$

This formula has no direct equivalent in LTL, but we can construct something similar. The following LTL formula says that in every path, a non-idle thread keeps taking steps until either it completes its operation or another thread takes a step:

$$\mathbf{AG}(t_1.location \neq \mathsf{idle} \rightarrow [lastStepTOne\,\mathbf{U}\,(t_1.location = \mathsf{idle} \vee \neg\, lastStepTOne)]) \quad \text{(OF-2)}$$

These formulas are similar, but not logically equivalent. There are two types of bad (non-obstruction-free) behaviour that can occur when a thread can not complete its operation in isolation:

1. A thread takes an infinite number of steps in isolation but never completes its operation.
2. A thread is only able to take a finite number of steps in isolation, and does not complete its operation.

The first type of behaviour violates both formulas. The second type violates the formula OF-1 (a path is required to exist which does not) but it does not violate the formula OF-2 (if p is false then $p\,\mathbf{U}\,(q \vee \neg\, p)$ is always true). This second type of behaviour is caused by blocking steps, such as "acquire lock", which are to be avoided in the construction of nonblocking algorithms for just this reason. Under the assumption that operations consist only of steps that are always enabled ("read", "write", "CAS" etc.), both OF-1 and OF-2 are equivalent for specifying obstruction-freedom.

5.3.4 Related Work

These nonblocking properties have been independently formalised by Dongol [2006], in a logic [Dongol and Goldson, 2006] not directly applicable to (existing) model checkers. Dongol [2009] has expressed these in first order LTL, which again is not directly applicable to most model checkers. The general structure of these formalisations is similar to the ones I have defined in this chapter, but there are a few differences.

The principal difference is that Dongol's definitions are more general, in particular regarding the concept of "progress". I have followed the common description [e.g. Herlihy, 1991; Colvin and Dongol, 2009] that a thread "makes progress" when it completes an operation. Dongol generalises this by allowing each thread state to have its own definition of what state is required to be reached in order to "make progress"; the motivation for this is that the progress properties can be defined without referring to the programs themselves.

Additional differences exist regarding thread enabledness. For model checking, I have assumed that a thread is always enabled by the system when it has an enabled transition, and that a system-disabled thread is indistinguishable from a very slow one. Furthermore, I do not necessarily assume weak fairness. Dongol [2006] does assume weak fairness in the systems and explicitly represents whether a thread is enabled by the system or not, independent of whether it has an enabled transition. Dongol [2009] does not assume weak fairness, but does assume that systems are "non-blocking", i.e. in every state each thread necessarily has an enabled transition.

The structure of Dongol's wait-freedom formula is the same as WF-1. The structure of the obstruction-freedom formula is logically equivalent to OF-2, but is slightly simplified — it is equivalent to:

$$\mathbf{AG}(t_1.location \neq \mathsf{idle} \rightarrow \mathbf{F}(t_1.location = \mathsf{idle} \vee \neg\, lastStepTOne))$$

The structure of the lock-freedom formula is more detailed than LF-1 because of the generalised presentation, but is relatively equivalent. In contrast, Colvin and Dongol [2009] formalise lock-freedom with an LTL formula directly equivalent to LF-1.

Gao and Hesselink [2007] formalise lock-freedom by counting the number of completed operations and ensuring that the number increases. With an auxiliary variable *completed* that is incremented by the final step of each operation, we could define lock-freedom as follows:

$$\forall\, n \in \mathbb{N} \bullet \mathbf{AG}(completed = n \rightarrow \mathbf{F}(completed > n))$$

This approach is unsuitable for model checking, as the auxiliary variable results an infinite statespace.

Chapter 6

Bounded Verification

In this chapter, we investigate bounded verification of linearisability and nonblocking properties, using the approaches described in Chapter 5. This chapter makes a novel contribution by analytically considering the small scope hypothesis for nonblocking data structures, providing some evidence towards it, and considering the question of how much confidence we can have in an algorithm given a successful bounded verification.

In Section 5.1, we described how to represent concurrent data structure algorithms using models that are parametrised by the number of threads, the maximum number of elements, and the size of the data type. If any one of these parameters is unbounded then the model has an infinite statespace and cannot be analysed directly by a model checker; when all three parameters are finite then the model has a finite statespace. In this chapter we attempt to verify the models that result when all parameters have finite bounds. Given the finite parameters n_t, n_s and n_d, each data structure model has n_t threads, an array[1] of length n_s and uses the integer data type $\{1 \ldots n_d\}$.

Initially, I performed preliminary work with the model checkers Spin (see Section 3.6.1) and SAL (see Section 3.6.2), with the dual aims of comparing the tools and investigating different ways of specifying systems and properties. The principal comparisons between Spin and SAL were for statespace explorations that did not check any properties; these models did not perform any memory reuse, which additionally bounded the statespaces. I found SAL's SAT-based model checker to be faster and use less memory than the BDD-based model checker. However, for the guar-

[1]The array is used directly for array based implementations, and is used to represent the available memory for nodes in linked list based implementations.

antees of verification required in this chapter we would need to know the diameter of the model (see Section 3.3.3) — an easy way to find this would be from the depth of the BDD-based analysis, but that would negate the benefits gained from using the SAT-based approach. In turn, I found SAL's BDD-based model checker to be somewhat more efficient than Spin. However, I found Spin to be much more practical for implementing garbage collection. The procedural language Promela allows all of the garbage collection code to be wrapped in a `d_step` block, so that it is performed by a single transition. The declarative SAL language has no such feature, so garbage collection may take several transitions each time, increasing the size of the stored statespace. Consequently, I chose to use Spin for all of the analyses in this chapter.

In Section 6.1 we explore bounded verification of linearisability, specifying the property as described in Section 5.2. We first discover (known) bugs in some incorrect algorithms, and then verify to the largest possible bounds some correct algorithms. Similarly, in Section 6.2 we explore bounded verification of nonblocking properties, specifying the properties as described in Section 5.3. In both of the these sections we consider how the results obtained could inform our confidence in a bounded verification result. In Section 6.3, we discuss some related work. Finally, Section 6.4 provides a summary of the results.

6.1 Checking Linearisability

Spin models of algorithms can be adapted to check linearisability using the approach presented in Section 5.2. An auxiliary specification data structure is added to each model, which performs its operations atomically at the linearisation points.

6.1.1 Example

Let us consider the stack from Section 2.6.1 as presented in Figure 5.6 (page 105). Recall that the linearisation point for a push operation is at the successful CAS step beginning at line 8; for a non-empty pop operation it is at the successful CAS step beginning at line 37; and for an empty pop operation it is at the read of the *Head* snapshot beginning at line 23. My modified Spin model performs the specification operations atomically at these linearisation points, as well as checking and updating the auxiliary

variable *doneLP*. The variable is initialised at the start of each operation and checked at the end.

The stack algorithm as presented is meant to be deployed in an environment with a garbage collector. If memory is explicitly freed and reused without a garbage collector, and there are at least two threads and two data values, an ABA error can occur (see Section 2.5.3).

Suppose that we modify the algorithm in Figure 5.6 to include the line

$$48': \quad \textbf{free}(ss)$$

in place of the existing lines 48 and 49. Figure 6.1 shows an execution, generated by Spin, of this modified algorithm with $n_t = 2$, $n_s = 1$ and $n_d = 2$, where an ABA error is detected. For clarity and space, the output has been simplified — the CAS steps are shown unified (unlike in Figure 5.6), the data values are given as d_1 and d_2, and the steps modifying *doneLP* are not shown. Additionally, the values of the variables at each step are shown in annotations.

Initially, Thread 2 completes an uninterrupted Push operation of the data value d_1, using the (sole) memory location 0; the specification stack has an entire Push(d_1) operation completed after the successful CAS step. Then Thread 2 continues and starts a Pop operation, where it reads a snapshot of the *Head* node's memory location and its *next* and *val* fields.

Now Thread 1 begins execution and starts a Pop operation, reading a snapshot of the *Head* node's location.

Control then switches back to Thread 2, which completes its Pop operation, with the specification stack having an atomic Pop(d_1) operation completed after the successful CAS step. Note that the value returned by the specification pop (d_1) matches the value returned by the implementation operation. At the end of this Pop operation the *Head* snapshot (memory location 0) is freed to the system to be reused, despite it still being the *Head* snapshot of Thread 1.

Thread 1 then takes a few more steps in its Pop operation, checking that the snapshot is not null and reading its *next* and *val* fields.

Then Thread 2 completes an entire Push operation of the data value d_2, using the free memory location 0. Again the specification stack has an entire Push(d_2) operation completed after the successful CAS step.

Finally, control switched back to the Thread 1, which performs the CAS step of its Pop operation, as the value of the *Head* pointer now matches its snapshot value. However, the value returned by the specification pop op-

Thread 1	Thread 2
	begin PUSH(d_1)
	3: $n :=$ **new**(Node)[0]
	4: $n.val := lv[d_1]$
	6: $ss := Head$[null]
	7: $n.next := ss$[null]
	9: CAS($Head$[null], ss[null], n[0])
	13: SPECPUSH($lv[d_1]$)
	begin POP()
	24: $ss := Head$[0]
	31: $ss[0] \neq$ null
	35: $ssnext := ss.next$[null]
	36: $lv := ss.val[d_1]$
begin POP()	
24: $ss := Head$[0]	
	38: CAS($Head$[0], ss[0], $ssnext$[null])
	42: **assert**($lv[d_1] =$ SPECPOP()[d_1])
	48′: **free**(ss[0])
	51: **return** $lv[d_1]$
25: $ss[0] \neq$ null	
35: $ssnext := ss.next$[null]	
36: $lv := ss.val[d_1]$	
	begin PUSH(d_2)
	3: $n :=$ **new**(Node)[0]
	4: $n.val := lv[d_2]$
	6: $ss := Head$[null]
	7: $n.next := ss$[null]
	9: CAS($Head$[null], ss[null], n[0])
	13: SPECPUSH($lv[d_2]$)
38: CAS($Head$[0], ss[0], $ssnext$[null])	
42: **assert**($lv[d_1] =$ SPECPOP()[d_2])	

Figure 6.1: Simplified execution of stack showing ABA error counter-example

eration (d_2) does not match the value to be returned by the implementation operation (d_1); thus the assertion fails.

This counterexample is not sufficient to disprove linearisability on its own — we must consider the possibility that the execution is correct but we have chosen incorrect linearisation points. Recalling the original definition of linearisability from Section 2.2, the execution has the following concurrent history:

$$\begin{aligned}&\mathrm{inv}_2(\textsc{Push}, d_1), \mathrm{resp}_2(\textsc{Push}, \mathrm{ok}), \mathrm{inv}_2(\textsc{Pop}), \mathrm{inv}_1(\textsc{Pop}),\\ &\quad \mathrm{resp}_2(\textsc{Pop}, d_1), \mathrm{inv}_2(\textsc{Push}, d_2), \mathrm{resp}_2(\textsc{Push}, \mathrm{ok}), \mathrm{resp}_1(\textsc{Pop}, d_1)\end{aligned}$$

There are three corresponding linear histories, where the overlapping operations have been permuted:

$$\begin{aligned}&\mathrm{inv}_2(\textsc{Push}, d_1), \mathrm{resp}_2(\textsc{Push}, \mathrm{ok}), \mathrm{inv}_2(\textsc{Pop}), \mathrm{resp}_2(\textsc{Pop}, d_1),\\ &\quad \mathrm{inv}_2(\textsc{Push}, d_2), \mathrm{resp}_2(\textsc{Push}, \mathrm{ok}), \mathrm{inv}_1(\textsc{Pop}), \mathrm{resp}_1(\textsc{Pop}, d_1)\end{aligned}$$

$$\begin{aligned}&\mathrm{inv}_2(\textsc{Push}, d_1), \mathrm{resp}_2(\textsc{Push}, \mathrm{ok}), \mathrm{inv}_2(\textsc{Pop}), \mathrm{resp}_2(\textsc{Pop}, d_1),\\ &\quad \mathrm{inv}_1(\textsc{Pop}), \mathrm{resp}_1(\textsc{Pop}, d_1), \mathrm{inv}_2(\textsc{Push}, d_2), \mathrm{resp}_2(\textsc{Push}, \mathrm{ok})\end{aligned}$$

$$\begin{aligned}&\mathrm{inv}_2(\textsc{Push}, d_1), \mathrm{resp}_2(\textsc{Push}, \mathrm{ok}), \mathrm{inv}_1(\textsc{Pop}), \mathrm{resp}_1(\textsc{Pop}, d_1),\\ &\quad \mathrm{inv}_2(\textsc{Pop}), \mathrm{resp}_2(\textsc{Pop}, d_1), \mathrm{inv}_2(\textsc{Push}, d_2), \mathrm{resp}_2(\textsc{Push}, \mathrm{ok})\end{aligned}$$

None of these linear histories is a legal history for a stack algorithm. Thus the execution is not a linearisable stack execution and the algorithm is not a linearisable stack.

6.1.2 Minimal Counterexamples

The above example of a stack error occurs in an instantiation of the algorithm with $n_t = 2$, $n_s = 1$ and $n_d = 2$, and occurs also in any instantiation where the parameters are increased, as the set of histories of the larger system is a superset of the smaller system's. However, when any of the parameters are decreased the error does not occur: when $n_t = 1$ the behaviour is sequential and correct (hence linearisable); n_s is already the minimum possible value; when $n_d = 1$ the correct value is returned as there is only one possible value.

Searching for bugs in smaller instantiations is preferable to larger systems, as it is quicker to find bugs (if they exist) and often easier to comprehend the error that results.

Algorithm	Bug	n_t	n_s	n_d	Reference
Stack	ABA^1	2	1	2	[Treiber, 1986]
Stack	ABA^1	2	2	1	[Treiber, 1986]
Queue	ABA^1	2	2	1	[Michael and Scott, 1996, 1998]
Queue	ABA^2	2	3	1	[Michael and Scott, 1996, 1998]
Queue	"skip"	2	2	2	[Shann et al., 2000]
Queue	ABA^1	2	2	1	[Ladan-Mozes and Shavit, 2004, 2008]
Deque	"empty"	2	3	1	[Detlefs et al., 2000]
Deque	ABA^3	3	4	1	[Detlefs et al., 2000]

Table 6.1: Minimum parameters for counterexamples to linearisability

I investigated the minimum instantiations needed to elicit errors in a number of non-linearisable concurrent data structures. The numbers are shown in Table 6.1 and were discovered by model checking instantiations beginning with $n_t = n_s = n_d = 1$ and incrementing parameters until an error was detected. Most of the errors are some form of ABA error.

ABA^1 is an error deliberately introduced for testing by using an implementation that frees and reuses memory without garbage collection. One counterexample is detailed in Section 6.1.1 — a thread performing a Pop operation detects that the *Head* node has remained "unchanged" but does not detect that its value has changed, so returns an incorrect value; this can be detected in the stack algorithm when $n_t = 2$, $n_s = 1$ and $n_d = 2$. An alternative counterexample occurs when a thread performing a Pop operation detects a singleton list (i.e. *Head.next* is null) and later detects that the *Head* has remained "unchanged", despite the list now being non-singleton, so transforms it to an empty list; this can be detected in the stack algorithm when $n_t = 2$, $n_s = 2$ and $n_d = 1$. It can be argued that the two counterexamples are equally small for the stack algorithm, but for the two queues the latter counterexample is smaller (with $n_d = 1$) due to the fact that the list always contains a dummy node.

ABA^2 is the error that occurs when the the `next` field of a dequeued node is set to null, as explained in Section 2.6.2 (page 23).

The bug "skip" occurs symmetrically in the Enqueue and Dequeue operations of an array based queue. It was discovered by Colvin and Groves [2005] and allows extra elements to be added or removed from the array-based list due to insufficient checks when restarting the loop after a conflict is detected.

Both bugs of the deque were discovered by Doherty [2003] (see also Doherty et al. [2004a]), and both appear symmetrically in the popLeft and popRight operations. The "empty" bug allows a pop operation to return empty when the list has never been empty — it insufficiently checks only local variables before returning. ABA^3 allows two pop operations to return the same value.

6.1.3 Confidence in Bounded Verification

When an instantiation of a parametrised model is verified, it provides a guarantee of correctness for instantiations up to those limits. It provides no general guarantee though — there is always the possibility that an error may exist within a larger instantiation. It is a widely-held belief though that most bugs can be exhibited with "small" instantiations — named the small scope hypothesis by Jackson [2012]. If so, then we can claim some level of confidence in an algorithm if the model can be verified with "medium" parameters. What is classed as medium, and the level of confidence that can be drawn (apart from it being less than 100%), can only be a subjective assertion.

The numbers from Table 6.1 conform to the hypothesis of small minimum instantiations for many bugs of nonblocking data structure algorithms. The parameters are all arguably "small", especially compared with practical uses, with $2 \leqslant n_t \leqslant 3$, $1 \leqslant n_s \leqslant 4$, and $1 \leqslant n_d \leqslant 2$.

Given these numbers, I would be mildly confident in the correctness of an algorithm if it was verified with bounds 1–2 greater than these, i.e. $4 \leqslant n_t \leqslant 5$, $5 \leqslant n_s \leqslant 6$, and $3 \leqslant n_d \leqslant 4$. This would verify the absence of the bugs identified in Section 6.1.2, and any others of a similar size. It is tempting to say that most bugs would have been caught at this level, but it does not leave very much margin to detect any potentially larger bugs.

With this amount of confidence, I would feel comfortable beginning a formal verification attempt, knowing that many common bugs are not present.

The corpus of bugs in Section 6.1.2 is relatively small, so more work must be undertaken to strengthen and/or extend these ranges. Given the importance of data structures within programs, it is hard to see how anything short of a full verification could suffice for providing a very high level of confidence in an algorithm.

Parameters			Spin		−DCOLLAPSE		−DMA	
Th	Mem	Data	RAM	Time	RAM	Time	RAM	Time
3	4	3	2,983	59s	2,298	89s	1,471	10.4m
4	3	4	3,228	74s	2,392	115s	1,396	10.2m
2	6	2	3,256	46s	2,933	80s	957	12.6m
7	2	1	4,023	119s	3,214	191s	1,873	14.4m
6	2	4	4,282	122s	3,413	209s	1,547	14.0m
2	5	5	—	—	—	—	2,865	80.5m
6	2	7	—	—	—	—	2,896	39.7m
5	3	2	—	—	—	—	3,153	24.0m
4	3	6	—	—	—	—	3,390	37.2m
3	4	4	—	—	—	—	3,596	38.5m
3	6	1	—	—	—	—	3,755	28.3m

Table 6.2: Treiber Stack Bounded Verification

6.1.4 Verification Limits

Having considered what bounds on verification will give some level of confidence in the correctness of an algorithm, it is natural to consider what can be practically achieved. I investigated this question by verifying linearisability for four algorithms to the largest bounds possible on a machine with a 4.5 GB RAM limit and a 3.33 GHz Intel Xeon processor. The Spin models used mostly default options, with a notable addition being the use of stack cycling (preprocessor macro −DSC), which uses the hard disk instead of RAM for the majority of the depth-first search stack. This option was used for all models, and proved beneficial in many, as the size of the stack (depth × state size) exceeded 2 GB.

With three parameters, it is not always clear when comparing two different instantiations which one is the "biggest". Comparing the number of stored states could be a useful metric, but I chose to compare the physical RAM use as it corresponds approximately with the statespace size and is the principal constraint on the model checker. Selected results ranked by RAM use are presented in the first pair of columns[2] after the parameters of Table 6.2 for the stack from Section 2.6.1 [Treiber, 1986], Table 6.3 for the original queue from Section 2.6.2 [Michael and Scott, 1996, 1998], Table 6.4 for the modified queue from Section 2.6.2 [Doherty et al., 2004b], and Ta-

[2]The remaining columns are discussed in the next section.

Parameters			Spin		−DCOLLAPSE		−DMA	
Th	Mem	Data	RAM	Time	RAM	Time	RAM	Time
2	4	5	3,050	48s	2,506	76s	1,493	19.3m
3	3	4	3,122	60s	2,230	95s	925	9.8m
5	2	1	3,249	75s	2,162	114s	876	9.9m
4	3	1	3,627	80s	2,422	123s	1,110	12.0m
4	2	7	3,880	83s	2,648	134s	650	10.9m
2	7	1	4,482	57s	3,205	95s	3,202	38.8m
3	3	7	—	—	—	—	3,279	63.3m
3	4	2	—	—	—	—	3,332	46.2m
3	5	1	—	—	—	—	3,635	43.0m
5	2	3	—	—	—	—	4,080	91.8m
2	5	3	—	—	—	—	4,164	73.5m
2	4	7	—	—	—	—	4,192	89.3m

Table 6.3: MS Queue Bounded Verification

Parameters			Spin		−DCOLLAPSE		−DMA	
Th	Mem	Data	RAM	Time	RAM	Time	RAM	Time
5	2	2	2,574	58s	1,728	90s	631	7.4m
3	3	5	2,911	61s	2,093	92s	1,086	9.7m
2	7	1	3,505	49s	2,518	70s	3,076	29.2m
2	4	6	4,194	66s	3,373	113s	2,852	34.5m
3	5	1	—	—	3,082	145s	2,047	18.0m
3	3	6	—	—	3,262	155s	1,628	17.8m
5	2	3	—	—	3,335	197s	1,136	17.2m
3	4	2	—	—	3,517	164s	2,202	21.5m
3	3	7	—	—	—	—	2,307	29.7m
6	2	1	—	—	—	—	2,632	47.7m
4	3	2	—	—	—	—	2,806	34.5m
5	2	7	—	—	—	—	3,126	97.0m

Table 6.4: DGLM Queue Bounded Verification

Parameters			Spin		−DCOLLAPSE		−DMA	
Th	Mem	Data	RAM	Time	RAM	Time	RAM	Time
3	3	2	1,872	34s	1,231	50s	276	4.7m
2	4	4	3,279	46s	2,478	78s	1,598	20.0m
2	5	2	3,741	48s	2,652	78s	2,376	24.0m
5	2	3	4,487	102s	2,409	168s	41	14.1m
5	2	4	—	—	4,093	241s	49	19.2m
3	3	3	—	—	4,190	176s	733	17.8m
3	4	1	—	—	4,381	177s	1,482	20.3m
6	2	7	—	—	—	—	728	23.5h
4	3	1	—	—	—	—	2,172	114.3m
7	2	1	—	—	—	—	2,455	130.0h
2	4	5	—	—	—	—	3,006	55.5m
3	3	6	—	—	—	—	3,585	180.0m

Table 6.5: LMS Queue Bounded Verification

ble 6.5 for another linked-list based queue [Ladan-Mozes and Shavit, 2004, 2008]. All RAM results are in MB as reported by Spin, and time results are in seconds, minutes or hours as indicated; analyses that ran out of memory are indicated with dashes.

None of the algorithms are able to be verified with these constraints for the minimum of the parameter ranges mentioned in the previous section ($n_t = 4, n_s = 5, n_d = 3$) and only the stack is able to be verified for the maximum values of each of the parameters in Table 6.1 ($n_t = 3, n_s = 4, n_d = 2$). There are verifiable instantiations that exceed these for one or two parameters, but not for all three. Hence we can conclude that model checking under these restrictions is not sufficient to provide general confidence in the correctness of a nonblocking data structure algorithm.

A first thought is that an increase in the RAM limitation could allow much larger instantiations to be verified. Unfortunately the RAM usage increases exponentially with each parameter increase, due to the state explosion problem. This means that a ten-fold increase in RAM may only raise the maximum verification rates by 1 for each parameter. Thus to make bounded verification practical for giving confidence in an algorithm it is essential that we apply some statespace reduction techniques.

Spin Reduction

Spin implements partial order reduction (see Section 3.4.2), which reduces the number of states that have to be searched. However, this is enabled by default and none of the Spin analyses used in this thesis explicitly disable the feature. Spin also implements two optional techniques for losslessly compressing the RAM used for storing states — collapse compression and minimised automata compression [Holzmann, 2004, pp. 198–206].

Collapse compression is a hierarchical indexing method that stores different components of the state separately; it is invoked using the preprocessor macro `-DCOLLAPSE`. The results of using this technique are given in the next pair of columns in Tables 6.2–6.5. The analyses are more memory efficient, using on average 50–90% of the RAM originally used, with the tradeoff of taking 140–175% of the time originally used. However, the gains against the state explosion problem are minimal — for the first two algorithms no further instantiations are able to be verified with the given limits, and for the second two algorithms only a small number (four and three, respectively) of additional instantiations are able to be verified.

Minimised automata compression uses a minimised finite state recogniser for state descriptors; it is invoked using the preprocessor macro `-DMA`. The results of using this technique are given in the final pair of columns in Tables 6.2–6.5. The analyses are even more memory efficient than for collapse compression, though they are more variable and have a much greater time tradeoff. The RAM usage varies from 1% to 90% of that originally used, and the time taken from 800% to 4,000% of the original. For all algorithms a number of additional instantiations are able to be verified, though again little progress is made against the state explosion problem — for n_t and n_s the maximum increase in verified parameters is 2. Furthermore the extended time penalties of this technique (one result in Table 6.5 took over five days) may make it less practical to use in certain cases.

Neither of these techniques of RAM compression are effective enough to combat the state explosion problem and engender a higher level of confidence in the algorithm. Thus it is necessary to consider techniques that reduce the statespace itself.

Symmetry Reduction

One of the most promising statespace reduction techniques for nonblocking data structures is symmetry reduction, introduced in Section 3.4.4.

Parameters			Spin		−DMA		TopSpin	
Th	Mem	Data	RAM	Time	RAM	Time	RAM	Time
3	4	3	2,983	59s	1,471	10.4m	607	10s
4	3	4	3,228	74s	1,396	10.2m	199	4s
2	6	2	3,256	46s	957	12.6m	1,945	24s
7	2	1	4,023	119s	1,873	14.4m	10	0s
6	2	4	4,282	122s	1,547	14.0m	30	1s
6	2	7	—	—	2,896	39.7m	70	2s
5	3	2	—	—	3,153	24.0m	107	3s
4	3	6	—	—	3,390	37.2m	625	14s
3	4	4	—	—	3,596	38.5m	1,856	30s
3	6	1	—	—	3,755	28.3m	1,282	20s
7	3	3	—	—	—	—	2,611	96s
3	5	2	—	—	—	—	2,672	44s
2	5	4	—	—	1,270	23.8m	3,225	42s
7	4	1	—	—	—	—	3,359	110s
5	3	7	—	—	—	—	3,389	93s
5	4	2	—	—	—	—	3,746	525s
4	4	3	—	—	—	—	3,847	79s
3	4	5	—	—	—	—	3,849	68s
6	3	5	—	—	—	—	3,981	127s

Table 6.6: Treiber Stack Bounded Verification With Symmetry Reduction

The data values of many algorithms can be modelled using scalarsets. Consider the stack algorithm from Section 2.6.1 — the only operations involving data values within the algorithm are assignments, and in an augmented model for checking linearisability there are additional comparisons of equality. Thus the data values can be represented using a scalarset. Similarly, the thread ids are not explicitly used within the algorithm, and in the augmented model for checking linearisability are only compared for equality; thus the thread ids can be represented using a scalarset. Similarly, the index of the array used for representing the linked list is only used for assignments and comparisons on equality; thus it can be represented using a scalarset.[3] Hence any two states that differ only by permutations of the data values, thread ids and node array index can be considered equivalent for model checking linearisability.

I used the extension TopSpin [Donaldson and Miller, 2006] to investigate verifying the stack algorithm with symmetry reduction.[4] TopSpin does not use scalarsets to identify symmetry, and the automatic procedure it uses was able to identify the symmetry of the threads only. Selected results are presented in Table 6.6, compared with figures from Table 6.2. The analyses are much more efficient, both in RAM and time. The reduction increases as the number of threads does — models with two threads used 51-98% of the original RAM usage, whilst models with six or seven threads used less than 2% of the original RAM usage. In contrast with the Spin reduction techniques, the running times of the analyses are reduced. The reductions are approximately the same as for the RAM use, which was expected as the efficiency is gained by reducing the size of the statespace to be explored, rather than the storage of the statespace.

Symmetry reduction does appear to combat the exponential state explosion problem, as the RAM reductions increase as n_t increases. However, for these models TopSpin is only able to address one of the three parameters — as n_s and n_d increase the statespace still increases exponentially. As a result, the increase in instantiations that are able to be verified with the given physical constraints is modest. The parameter increase ranges spanned 0–3 for n_t, 0–1 for n_s and 0–6 for n_d. However, the minimum instantiations mentioned in Section 6.1.3 were unable to be verified, though several nearby instantiations were able to be verified:

[3] A scalarset could not be used for the array index in an array-based data structure, as those algorithms must increment and decrement the values.

[4] The models of the queue algorithms were incompatible with TopSpin's Promela restrictions so would need to be rewritten.

- out of memory: $n_t = 4$, $n_s = 5$, $n_d = 3$
- verified: $n_t = 2$, $n_s = 5$, $n_d = 3$
- verified: $n_t = 4$, $n_s = 4$, $n_d = 3$
- verified: $n_t = 4$, $n_s = 5$, $n_d = 1$

The results indicate that symmetry reduction is an effective approach for combating the state explosion problem for nonblocking data structures. If the models were analysed with a tool capable of addressing the symmetry of all three parameters (perhaps through the use of scalarsets) it appears highly likely that the resulting reductions would be large enough to allow verification of instantiations in the ranges mentioned in Section 6.1.3. The reductions may also be large enough to greatly exceed these ranges. Additional work is needed to investigate this approach further.

6.2 Checking Nonblocking Properties

Spin models of algorithms can also be adapted to check nonblocking properties using the approaches presented in Section 5.3. In this section, we consider a small number of algorithms that all happen to be wait-free or lock-free (in part because there are relatively few published algorithms that are obstruction-free but not lock-free).

For wait-freedom, I defined two formulas: WF-1 (page 111) implicitly assumes weak fairness and WF-2 (page 112) is longer to account for the possibility of weak fairness not being satisfied. Spin has a run-time option for enforcing weak fairness [Holzmann, 2004, p. 538] so it would be simpler to use this with the first formula. Spin transforms the formula to a Büchi automaton described in Promela's "never claim" syntax, as shown in Figure 6.2. The label "`IDLE`" is given to the step (a `do` loop) that chooses which operation to perform.

As well, Spin provides an alternative to using LTL formulas — both WF-1 for wait-freedom and LF-1 for lock-freedom. It has an option to check whether certain labelled "progress steps" occur infinitely often in each execution path. For wait-freedom we label the last step of each of the *first* thread's operations, and utilise the weak fairness option in the model checker. For lock-freedom we label the last step of *every* thread's operations, and do not need to require weak fairness.

```
never {    /* !G((!p) -> F(p)) */
T0_init :      /* init */
    if
    :: (! (thread[1]@IDLE)) -> goto accept_S4
    :: (1) -> goto T0_init
    fi;
accept_S4 :    /* 1 */
    if
    :: (! (thread[1]@IDLE)) -> goto accept_S4
    fi;
}
```

Figure 6.2: Promela never claim for LTL formula WF-1

6.2.1 Example

Let us again consider the stack from Section 2.6.1, as shown in Figure 2.7. We can check this algorithm for wait-freedom using the formula WF-1 — specifically in Spin by including the never claim in Figure 6.2. The never claim attempts to prove the negation of 6.2

$$\mathbf{EF}(\neg\, \texttt{thread[1]@IDLE} \wedge \mathbf{G} \neg\, \texttt{thread[1]@IDLE})$$

i.e. that there is a path where the first thread never returns to the IDLE state.

Figure 6.3 shows a simplified execution displaying such a counterexample, when $n_t = 2$, $n_s = 1$ and $n_d = 1$. Thread 1 is instantiated at IDLE and completes a push operation, returning to IDLE, and then begins a pop operation by reading a snapshot of the *Head*. Control switches to Thread 2, which performs an entire pop operation. Now Thread 1 continues and the CAS step fails because Thread 2 has changed the value of *Head*. Control switches again to Thread 2, which performs an entire push operation.

When Thread 1 retries the loop in its pop operation, reading a snapshot of the *Head*, the same behaviour occurs again. This results in an execution where Thread 1 is never able to complete — between reading the *Head* snapshot and attempting the CAS update the snapshot is obsoleted by Thread 2 performing a pop operation. Since a thread can take an infinite number of steps without completing an operation the algorithm is not wait-free.

Thread 1	Thread 2
begin PUSH(d_1)	
2: $n :=$ **new**(Node)[0]	
3: $n.val := lv[d_1]$	
5: $ss := Head$[null]	
6: $n.next := ss$[null]	
7: CAS($Head$[null], ss[null], n[0])	
begin POP()	
11: $ss := Head$[0]	
—*begin infinite loop*—	
12: $ss[0] \neq$ null	
15: $ssnext := ss.next$[null]	
16: $lv := ss.val[d_1]$	
	begin POP()
	11: $ss := Head$[0]
	12: $ss[0] \neq$ null
	15: $ssnext := ss.next$[null]
	16: $lv := ss.val[d_1]$
	17: CAS($Head$[0], ss[0], $ssnext$[null])
	18: $ss.next :=$ null
	19: $ss.val :=$ null
	20: **return** $lv[d_1]$
17: $\neg$ CAS($Head$[null], ss[0], $ssnext$[null])	
	begin PUSH(d_1)
	2: $n :=$ **new**(Node)[0]
	3: $n.val := lv[d_1]$
	5: $ss := Head$[null]
	6: $n.next := ss$[null]
	7: CAS($Head$[null], ss[null], n[0])
11: $ss := Head$[0]	
—*end infinite loop*—	

Figure 6.3: Simplified execution of stack showing wait-freedom counterexample

Algorithm	Property	Th	Mem	Data	Reference
Stack	WF	2	1	1	[Treiber, 1986]
Queue	WF	2	2	1	[Michael and Scott, 1996, 1998]
Queue	WF	2	3	1	[Ladan-Mozes and Shavit, 2004, 2008]
Deque	WF	2	3	1	[Detlefs et al., 2000]
Queue	WF	1	1	1	[Shann et al., 2000]
Queue	LF	1	1	1	[Shann et al., 2000]
Queue	OF	1	1	1	[Shann et al., 2000]

Table 6.7: Minimum parameters for counterexamples to nonblocking properties

6.2.2 Minimal Counterexamples

As is the case for linearisability (see Section 6.1.2) it is quicker to discover counterexamples to nonblocking properties in smaller instantiations of a model. In the above example, a counterexample to wait-freedom is found when $n_t \geqslant 2$, $n_s \geqslant 1$ and $n_d \geqslant 1$ but no smaller — with only one thread there is no inter-thread conflict to cause an operation to restart.

Table 6.7 shows the minimum instantiations needed to produce counterexamples to nonblocking properties in the algorithms used in Section 6.1. The linked list based algorithms are all lock-free, so we find counterexamples to wait-freedom. All of these algorithms require only two threads and one data value, but different values for n_s depending on the algorithm design.

The original array based queue of Shann et al. [2000] is intended to be lock-free, however the enqueue and dequeue operations wait for the list to change when the queue is full or empty (respectively); thus wait-, lock- and obstruction-freedom can all be disproved when $n_t = n_s = n_d = 1$.

This table appears to indicate that minimal nonblocking counterexample instantiation sizes are similar to or smaller than those for linearisability. However, the selection is small and limited so further work would be needed to draw any firm conclusions.

6.2.3 Confidence in Bounded Verification

The results from Table 6.7 are too few to draw any general conclusions about counterexamples to wait-freedom, and do not address counterexamples to lock- or obstruction-freedom. However, whilst further work

Parameters			Spin	
Th	Mem	Data	RAM	Time
3	4	2	3,210	42s
6	2	1	3,279	51s
2	4	6	3,867	57s
3	3	7	4,083	76s
5	2	5	4,184	97s
5	3	1	4,254	87s
4	4	1	4,294	91s
7	1	6	4,364	116s

Table 6.8: Treiber Stack Bounded Verification

Parameters			Spin	
Th	Mem	Data	RAM	Time
4	3	1	3,164	46s
4	2	7	3,264	52s
2	4	4	3,315	40s
2	5	2	3,347	36s
3	3	4	3,695	64s
5	2	2	4,129	100s

Table 6.9: MS Queue Bounded Verification

will be needed to investigate these questions, the results do appear to indicate that counterexamples to nonblocking properties are of a similar order of magnitude to those for linearisability.

6.2.4 Verification Limits

I investigated the limits of verifying lock-freedom for the four algorithms above, using the same machine and constraints (notably 4.5 GB RAM) as in Section 6.1.4. Tables 6.8–6.11 present selected results ordered by RAM usage. Again, the numbers are of a similar magnitude to those for linearisability.

Further work is required to investigate these questions across all nonblocking properties, but my preliminary results suggest that the answers may be similar to those for linearisability. If so, then symmetry reduction will be needed to combat the state explosion problem and allow verifica-

Parameters			Spin	
Th	Mem	Data	RAM	Time
2	5	2	3,143	29s
3	3	5	3,815	68s
6	2	1	4,237	118s
5	2	4	4,378	112s
2	4	5	4,443	74s

Table 6.10: DGLM Queue Bounded Verification

Parameters			Spin	
Th	Mem	Data	RAM	Time
4	2	7	3,007	63s
2	4	3	3,336	50s
2	6	1	3,565	56s
3	3	2	3,639	77s

Table 6.11: LMS Queue Bounded Verification

tion up to sufficient bounds to supply confidence in an algorithm. TopSpin only supports verifying safety properties, but future work may extend it combine techniques for supporting LTL properties [e.g. Bošnački, 2003].

6.3 Related Work

A number of authors report model checking attempts on single algorithms [e.g. Harris, 2001; Colvin et al., 2006; Lamport, 2006]. Analyses were limited in the number of operations that were performed, and the largest instances able to be checked were all quite small.

Vechev et al. [2009] report checking linearisability for a range of algorithms (though only a lazy list algorithm is mentioned specifically) using Spin. They use a similar approach to mine for identified linearisation points, but can also analyse models without identified linearisation points. In the latter case, they record the history within the state and search for valid interleavings; the number of operations must be bounded or otherwise the statespace would be infinite. For the lazy list algorithm with identified linearisation points and 16 GB of RAM they can only verify linearisability for 2 threads and 2 keys.

A number of algorithms have had linearisability checked [Liu et al., 2009; Zhang et al., 2009; Zhang and Liu, 2010] with the PAT model checker [Sun et al., 2009]. These analyses use FDR-style refinement to show trace inclusion of the implementation and specification, and do not require linearisation points to be identified. For analyses with identified linearisation points, their results appear to be similar to mine, giving no greater degree of confidence. It is hard to compare directly though, as they only report the number of threads and the size of the list, but not the number of distinct data values. For the stack algorithm, they used 2 GB of RAM and report verifying a model with 2 threads and list of size 14, one with 3 threads and list of size 2, and one with 4 threads and list of size 2. For the (original) queue algorithm, they report verifying a model with 2 threads and list of size 8. The analyses without identified linearisation points are naturally larger and slower. For a lazy list algorithm they are able to verify a model with 2 threads and 1 key, but for any larger parameters must limit each thread to only 1 operation, despite using 32 GB of RAM.

6.4 Conclusion

In this chapter I investigated bounded verification of linearisability and nonblocking properties. For linearisability, I determined that bounded verification under current physical constraints, even using Spin's memory compression techniques, is not sufficient to provide much general confidence in an algorithm's correctness.

When symmetry reduction was applied to the threads the results improved. I hypothesise that if this technique is also applied to the linked list memory elements and the data values then the possible bounds on verification could be large enough to engender some level of confidence in an algorithm's correctness from bounded verification alone.

I performed some preliminary investigations, which indicate that the same conclusions may be found for nonblocking properties; however further data is needed.

For both types of properties, my results provide some evidence to support the small scope hypothesis for nonblocking data structures, though they do not (and cannot) prove it.

Part III

Verification

Chapter 7

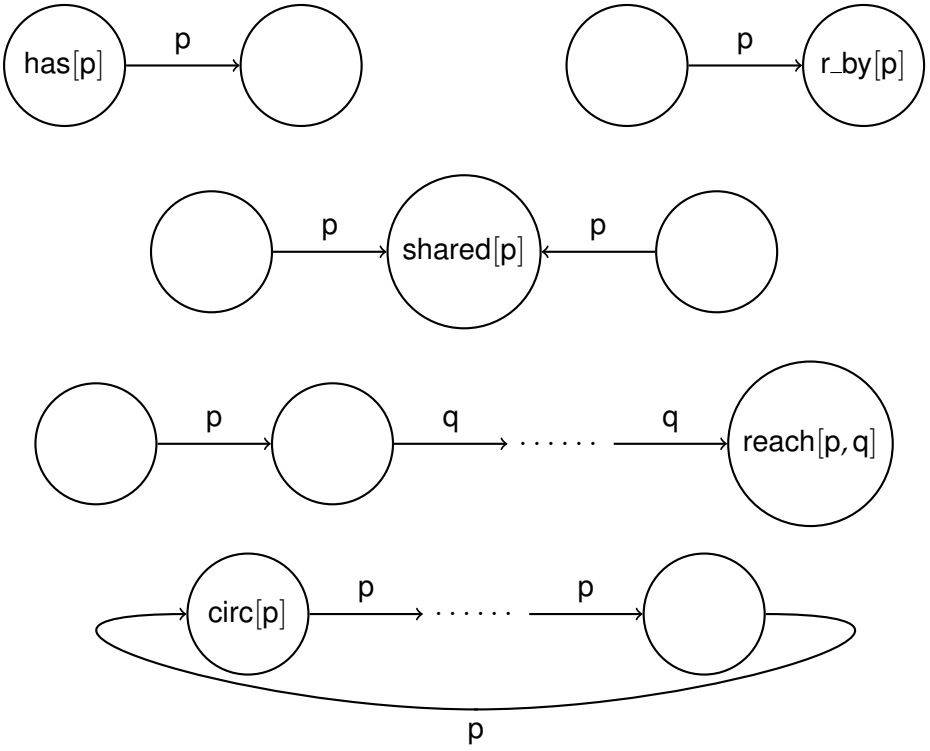
Canonical Abstraction for Linearisability

In this chapter we explore how canonical abstraction can be used to abstract linked list based data structures to finite sized models with enough precision to allow linearisability to be verified, and introduce several novel instrumentation predicates.

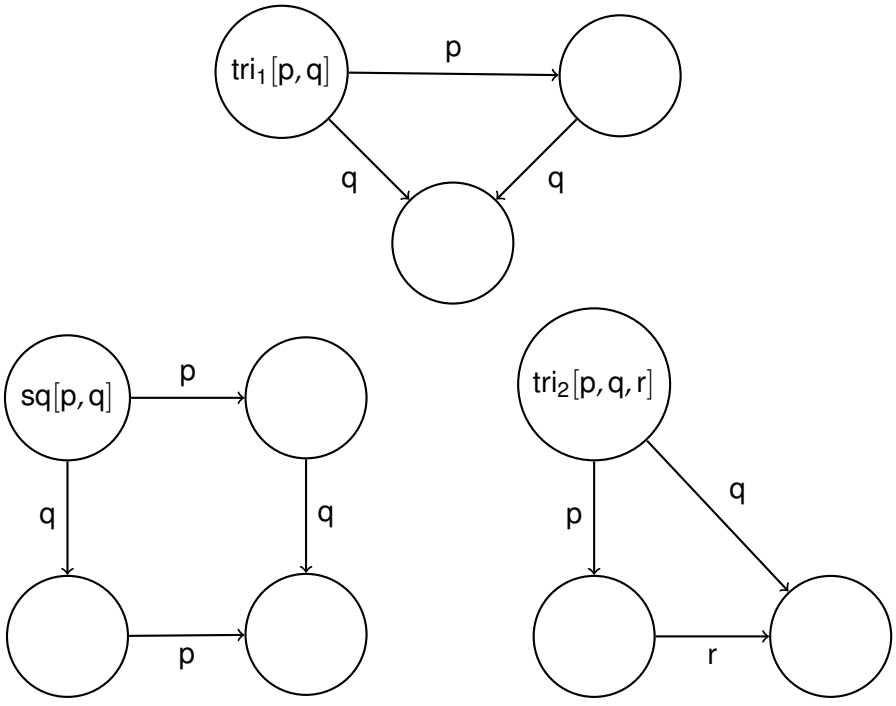
For verifying linearisability, we add a specification data structure to the model as described in Section 5.2. In an abstract state we need to be able to infer properties such as whether the implementation and specification lists are the same length, and whether they contain the same data values in the same order. This information cannot be retained in canonically abstract states using only the instrumentation predicates introduced in Chapter 4; indeed, previous work on using canonical abstraction for verifying linearisability invented another abstraction technique to solve this problem (see Section 7.10).

The instrumentation predicates from Table 4.2 can be classified as *linear*. Figure 7.1a illustrates the linear nature of these predicates — some relate two or three objects joined by one or two binary predicates; some relate a line of objects of unbounded length (circ is a line that begins and ends at the same point). I propose several novel *geometric* predicates, which have a triangle or square shape, as illustrated in Figure 7.1b. Such predicates allow information about important "two-dimensional" relationships between objects to be retained in canonically abstract states.

With the addition of a core binary predicate to relate the nodes of the implementation and specification lists, these geometric predicates can be used to refine canonical abstraction sufficiently to enable linearisability to



(a) Linear predicates



(b) Geometric predicates

Figure 7.1: Instrumentation predicate structures

be verified for unbounded threads.

This chapter uses the stack algorithm from Section 2.6.1 as a running example. Section 7.1 restates the stack model from Section 5.2.6 in a format that more closely resembles the operational semantics used for canonical abstraction. Section 7.2 presents the elements needed for representing states as 3-valued logical structures — core predicates, integrity rules, and instrumentation predicates from Chapter 4. Section 7.3 discusses the properties of the linked lists that need to be retained by an abstraction for verifying linearisability; it introduces a new core predicate and defines two geometric instrumentation predicates. Section 7.4 discusses how to represent more than one thread, and which properties of threads' fields need to be retained by an abstraction for verifying linearisability; it defines four more geometric instrumentation predicates. Section 7.5 describes the initial state that I used for the stack model. Section 7.6 describes some additional compatibility constraints that were added to aid the concretisation of abstract states. Section 7.7 presents the results of verifying linearisability of the stack algorithm. Section 7.8 discusses several less efficient variations of the stack model, including alternative instrumentation predicate definitions and less restrictive ad hoc partial order reductions. Section 7.9 explains how the model of the stack can be adapted to verify linearisability of the unbounded queue algorithms from Section 2.6.2. Finally, Section 7.10 discusses some related work, and Section 7.11 summarises the results.

7.1 Basic Stack Model

We begin by considering the stack algorithm from Section 2.6.1, as described in Section 5.2.6 and Figure 5.6, with the sequential specification merged in. In Figure 7.2 we present a different representation of the algorithm that more closely resembles the operational semantics used for canonical abstraction (see Section 4.4.2).

Each transition contains a "from" thread location, where it is enabled, an update action, and a "to" thread location; the thread locations are named to match the line numbers from Figure 5.6. The transitions are wrapped in atomic blocks, which prevent interleaving with transitions from other threads. These include the atomic blocks present in Figure 5.6, and some are expanded to include transitions on local variables in a form of ad hoc

atomic {

idle	beginPush()	$push_3$
$push_3$	newNode(n)	$push_4$
$push_4$	assign($n.val, lv$)	$push_6$
$push_6$	assign($ss, Head$)	$push_7$
$push_7$	assign($n.next, ss$)	$push_9$

}
atomic {

$push_9$	CASfail($Head, ss$)	$push_6$
$push_9$	CASsucc($Head, ss, n$)	$push_{11}$
$push_{11}$	specPush()	$push_{18}$
$push_{18}$	endPush()	idle

}

atomic {

idle	beginPop()	pop_{24}
pop_{24}	assign($ss, Head$)	pop_{25}
pop_{25}	isNotNull(ss)	pop_{35}
pop_{25}	isNull(ss)	pop_{26}
pop_{26}	specPopEmpty()	pop_{48}

}
atomic {

pop_{35}	assign($ssnext, ss.next$)	pop_{36}
pop_{36}	assign($lv, ss.val$)	pop_{38}

}
atomic {

pop_{38}	CASfail($Head, ss$)	pop_{24}
pop_{38}	CASsucc($Head, ss, ssnext$)	pop_{40}
pop_{40}	specPop()	pop_{48}
pop_{48}	endPop()	idle

}

Figure 7.2: Transitions of stack model

partial order reduction, as discussed in Section 5.1.3.[1]

The update actions are detailed in Figure 7.3. Each action has the form

$$guard \longrightarrow updates$$

where *guard* is a boolean formula that must be true for the action to be applied, and *updates* is a sequence of assignments. The most notable actions are for the CAS tests — we have separated them into disjoint success and failure actions.

In order to discover linearisation errors, some alarm must be raised if the response (endPush, endPop) and specification (specPush, specPop, specPopEmpty) action guards are not met. One option would be to define a complementary action for each action, using the negation of the original guard and add alternative transitions that execute these actions and move to an error state. For example, to complement the response of a push operation, we could define action

endPushError() $\neg\, doneLP \longrightarrow$

and transition

push_{18} endPushError() lin_fail

However, it is easier in TVLA to halt the analysis during the original transition, by using the "message" mechanism [see Lev-Ami, 2000, Section C.3]. We remove the guard from the transition's precondition, and use the negation of the guard as the trigger for a message that will halt the analysis before the update is performed. For example, we redefine the action endPush in Figure 7.3 as

endPush() true $\longrightarrow$
$lv, ss, n := \text{null}$

and have it trigger a message $\neg\, doneLP$ before the update is performed.

7.2 Three-Valued Model

In order to represent the states of this model using logical structures, we define the core predicates necessary for 2-valued structures (7.2.1), and some integrity rules (7.2.2) necessary for concretising from 3-valued structures. We then define a selection of instrumentation predicates from those introduced in Chapter 4, for refining abstractions from 2- to 3-valued structures (7.2.3).

[1] Section 7.8 contains a discussion of models with different partial order reduction.

newNode(x) true $\longrightarrow$
$x := \text{newNode}(\text{null}, \text{null})$

assign(x,y) true $\longrightarrow$
$x := y$

isNull(x) $x = \text{null} \longrightarrow$

isNotNull(x) $x \neq \text{null} \longrightarrow$

CASfail(loc,old) $loc \neq old \longrightarrow$

CASsucc(loc,old,new) $loc = old \longrightarrow$
$loc := new$

beginPush() true $\longrightarrow$
$doneLP := \text{false};\ lv \in \textbf{DATA}$

specPush() $\neg\, doneLP \longrightarrow$
$doneLP := \text{true};\ spec.Head := \text{newNode}(lv, spec.Head)$

endPush() $doneLP \longrightarrow$
$lv, ss, n := \text{null}$

beginPop() true $\longrightarrow$
$doneLP := \text{false}$

specPopEmpty() $\neg\, doneLP \wedge spec.Head = \text{null} \longrightarrow$
$doneLP := \text{true}$

specPop() $\neg\, doneLP \wedge spec.Head \neq \text{null} \wedge spec.Head.val = lv \longrightarrow$
$doneLP := \text{true};\ spec.Head := spec.Head.next$

endPop() $doneLP \longrightarrow$
$lv, ss, ssnext := \text{null}$

Figure 7.3: Update actions used in stack model

7.2.1 Core Predicates

I use core predicates to describe the types of objects — threads, nodes and data values — so we use the following three unary predicates:

$$\mathsf{is\_thread}, \mathsf{is\_node}, \mathsf{is\_data}$$

To reduce clutter and make diagrams of structures clearer, we will not represent these predicates textually. Instead we use different shapes for each of the types — circles for nodes, hexagons for threads and squares for data values.

One could choose to represent the implementation and specification stacks explicitly as objects in the universe, with the head fields as binary predicates. However, since they are the only data structures in the model it is simpler to just define the heads as global variables. Thus we use the following unary predicates, using subscripts to denote the Implementation and Specification heads:

$$\mathsf{Head_I}, \mathsf{Head_S}$$

For the threads we define a unary predicate for each location:

$$\begin{array}{l}\mathsf{at}[\mathsf{idle}], \mathsf{at}[\mathsf{push}_3], \mathsf{at}[\mathsf{push}_4], \mathsf{at}[\mathsf{push}_6], \\ \mathsf{at}[\mathsf{push}_7], \mathsf{at}[\mathsf{push}_9], \mathsf{at}[\mathsf{push}_{11}], \mathsf{at}[\mathsf{push}_{18}], \\ \mathsf{at}[\mathsf{pop}_{24}], \mathsf{at}[\mathsf{pop}_{25}], \mathsf{at}[\mathsf{pop}_{26}], \mathsf{at}[\mathsf{pop}_{35}], \\ \mathsf{at}[\mathsf{pop}_{38}], \mathsf{at}[\mathsf{pop}_{40}], \mathsf{at}[\mathsf{pop}_{48}]\end{array}$$

Finally, the fields of the threads and nodes are represented by binary predicates and the boolean variables by unary predicates:

$$\mathsf{next}, \mathsf{val}, \mathsf{n}, \mathsf{lv}, \mathsf{ss}, \mathsf{ssnext}, \mathsf{doneLP}$$

7.2.2 Integrity Rules

The integrity rules that are needed for refining the abstraction come from the properties we expect from pointer fields and global variables. The two stack head variables are unique, so we include the following rules:

$$\begin{array}{l}\forall\, v_1, v_2 \bullet \mathsf{Head_I}(v_1) \wedge \mathsf{Head_I}(v_2) \rightarrow \mathsf{eq}(v_1, v_2) \\ \forall\, v_1, v_2 \bullet \mathsf{Head_S}(v_1) \wedge \mathsf{Head_S}(v_2) \rightarrow \mathsf{eq}(v_1, v_2)\end{array}$$

Additionally, the node and thread fields are functional, so we include the following rules:

$$\forall v_1, v_2, v_3 \bullet \mathsf{next}(v_1, v_2) \wedge \mathsf{next}(v_1, v_3) \rightarrow \mathsf{eq}(v_2, v_3)$$
$$\forall v_1, v_2, v_3 \bullet \mathsf{val}(v_1, v_2) \wedge \mathsf{val}(v_1, v_3) \rightarrow \mathsf{eq}(v_2, v_3)$$
$$\forall v_1, v_2, v_3 \bullet \mathsf{n}(v_1, v_2) \wedge \mathsf{n}(v_1, v_3) \rightarrow \mathsf{eq}(v_2, v_3)$$
$$\forall v_1, v_2, v_3 \bullet \mathsf{lv}(v_1, v_2) \wedge \mathsf{lv}(v_1, v_3) \rightarrow \mathsf{eq}(v_2, v_3)$$
$$\forall v_1, v_2, v_3 \bullet \mathsf{ss}(v_1, v_2) \wedge \mathsf{ss}(v_1, v_3) \rightarrow \mathsf{eq}(v_2, v_3)$$
$$\forall v_1, v_2, v_3 \bullet \mathsf{ssnext}(v_1, v_2) \wedge \mathsf{ssnext}(v_1, v_3) \rightarrow \mathsf{eq}(v_2, v_3)$$

7.2.3 Instrumentation Predicates

As expected, several instrumentation predicates are needed to record information that gets lost by the abstraction on core predicates, notably about the values of fields and the connectedness of the list. However, it is possible to define additional instrumentation predicates that are not necessary for the property being verified and serve only to increase the statespace. Here I describe not only which instrumentation predicates I chose to define, but also why I chose not to define certain others for verifying linearisability.

These decisions were arrived at through a process of experimentation with different inputs to TVLA. When the abstraction was too coarse then a (spurious) error was encountered; examining the counterexample I was able to determine (first that it was actually spurious, then) what instrumentation predicate(s) could be used to refine the abstraction enough to prevent the spurious execution from occurring.

It was not so easy, however, to determine when the abstraction was too fine — the algorithm could still be verified, just not as efficiently. I initially used too many instrumentation predicates, and was puzzled that small models (e.g. with just one or two threads) were much more expensive to verify than I expected. When I realised that the abstraction was too fine I looked at each instrumentation predicate and considered whether it was *necessary* for verification.

Reachability and circularity

This model contains two distinct acyclic lists — the implementation stack and the specification stack. As discussed in Section 4.2.2, we can preserve

these facts using one circularity and two reachability predicates:[2]

$$\mathsf{circ}(v) \Leftrightarrow \mathsf{next}^+(v, v)$$
$$\mathsf{reach_I}(v) \Leftrightarrow \exists\, u \bullet \mathsf{Head_I}(u) \wedge \mathsf{next}^*(u, v)$$
$$\mathsf{reach_S}(v) \Leftrightarrow \exists\, u \bullet \mathsf{Head_S}(u) \wedge \mathsf{next}^*(u, v)$$

Has-a-field

For every field it is important at some point in the algorithm to distinguish whether it is null or not, e.g.:

- val is always non-null, as we only push non-null values into the list.
- At location push_9, the next field of a thread's n node is null if and only if the thread's ssnext field is null.
- Push operations only have non-null values for n and lv after they are assigned.
- In a non-empty pop operation, ss is always non-null after it is assigned.

When the value of a field predicate is unknown, it may be focussed to either false (null) or true (non-null). For this reason we use instrumentation predicates of the following form to record when a field is non-null:

$$\mathsf{has}[\mathsf{field}](v) \Leftrightarrow \exists\, u \bullet \mathsf{field}(v, u)$$

Referenced-by-field

It is possible to distinguish the objects that are pointed to by the various field predicates using the r_by[field] predicate defined in Section 4.2.2. For the field predicates next, ss, ssnext and lv, the additional information retained in a canonical abstraction by such a predicate is not necessary in order to verify linearisability, so only serves to inflate the abstract state-space. (This assertion is quantified in Section 7.8.)

This instrumentation predicate would be beneficial for the n field predicate, as it is necessary to distinguish the nodes that have yet to be added

[2]For brevity, we remove the "parameters" of these predicates, as used in Chapter 4, and use subscripts to identify reachability from the Implementation and Specification Heads.

to the lists from the nodes that have been removed (the nodes currently in the lists are distinguished by the reachability predicates). However, r_by[n] would also distinguish between nodes in the implementation list that are pointed to by an n field and those that aren't — this is not needed for verifying linearisability so unnecessarily inflates the statespace.

Since a thread's n node is appended to the stack at the linearisation point, we can uniquely distinguish the "unpushed" nodes with the following instrumentation predicate:

$$\mathsf{waiting}(v) \Leftrightarrow \exists\, u \bullet \mathsf{n}(u, v) \wedge \neg\, \mathsf{doneLP}(u)$$

Shared

An important property of a push operation is that each thread's n node is unique and not pointed to by another thread's n field. This property can be lost by canonical abstraction, but can be retained by using the "shared" predicate:

$$\mathsf{shared}[\mathsf{n}](v) \Leftrightarrow \exists\, u_1, u_2 \bullet \mathsf{n}(u_1, v) \wedge \mathsf{n}(u_2, v) \wedge \neg\, \mathsf{eq}(u_1, u_2)$$

None of the other fields have a similar property, so defining this instrumentation predicate for them would again only serve to inflate the abstract statespace.

7.3 Preserving Linearisability Information

The predicates defined so far are not sufficient to preserve enough information for verifying linearisability. Consider Figure 7.4, which shows the abstraction of (the lists of) two states where the implementation and specification stacks have length 3 — in $S_1^\natural$ the lists have the same values in the same order, and in $S_2^\natural$ the lists' head values differ. Both states have the same canonical abstraction and the information about the values is lost.

A consequence of this is that if the algorithm is linearisable, a spurious counterexample will be generated during a Pop operation. The states $S_1^\natural$ and $S_2^\natural$ are in the same equivalence class, even though $S_1^\natural$ is reachable and $S_2^\natural$ is not; thus $S_2^\natural$'s unreachable (and erroneous) successors are treated as being reachable from $S_1^\natural$. Specifically, abstract state S_1, representing $S_1^\natural$,

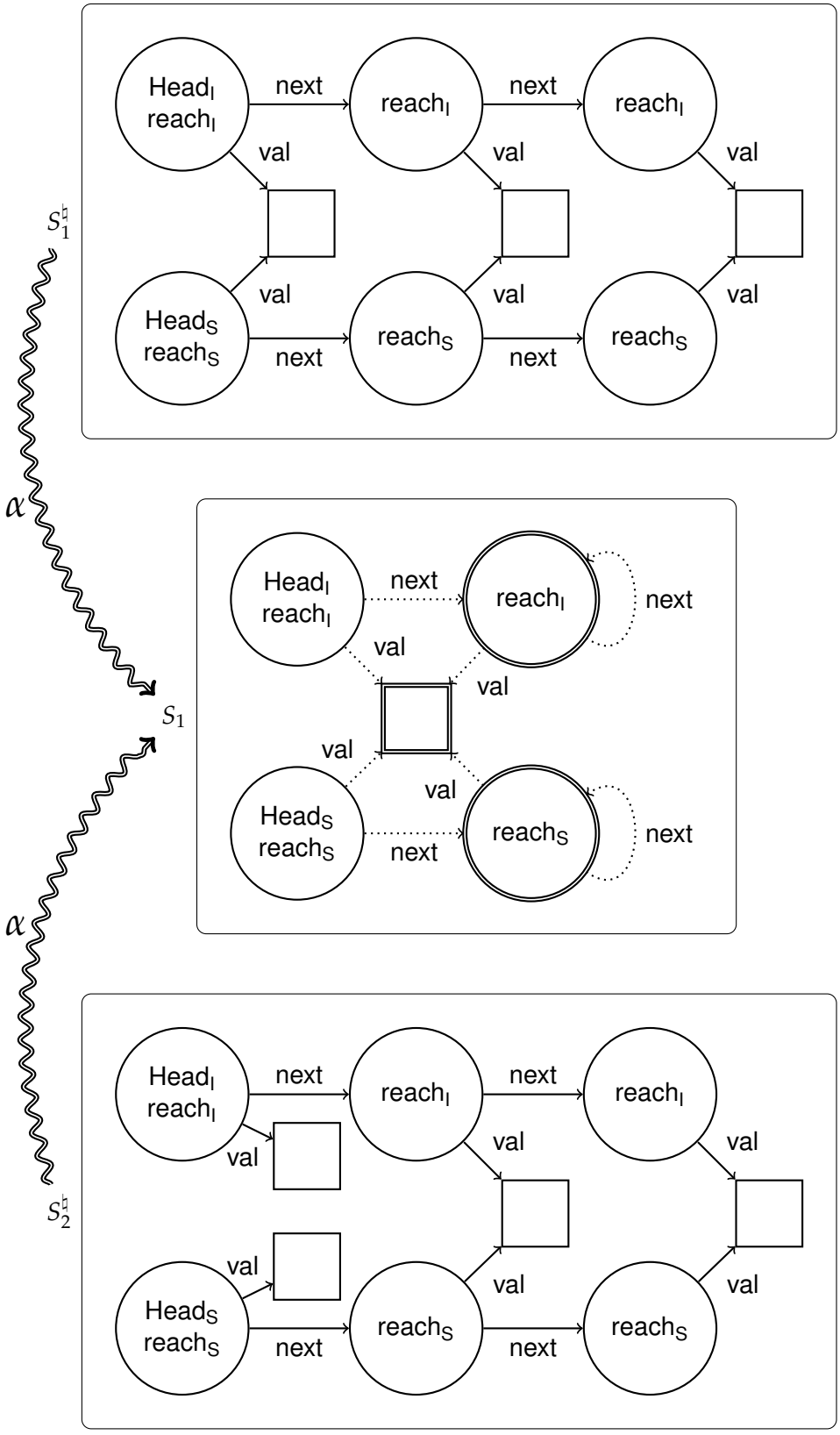


Figure 7.4: Canonical abstraction of two lists: the property of matching values is lost

specPush(x) $\neg\, doneLP \longrightarrow$
$doneLP := \text{true};\ spec.Head := \text{newNode}(x, spec.Head);$
$spec(Head, spec.Head) := \text{true}$

specPop(x) $\neg\, doneLP \wedge spec.Head \neq \text{null} \wedge spec.Head.val = x \longrightarrow$
$doneLP := \text{true};\ spec(\_, spec.Head) := \text{false};$
$spec.Head := spec.Head.next$

Figure 7.5: Stack specification update operations, incorporating the *spec* relation

can be concretised by Focus into $S_2^\natural$, which will lead to a failure of linearisability when the values returned by the implementation and specification Pop operations do not match. Thus it is necessary to preserve information about the relationship between the two lists' values — namely whether they have the same values in the same order.

In order to define instrumentation predicates that capture that information, we need to be able to relate the i-th pair of nodes in the two lists. This is easy for the head nodes, but for the remaining nodes it is not possible to do in first order logic with transitive closure We would need to extend the logic in order to define an instrumentation predicate of the form:

$$\begin{array}{l}\mathsf{ipair}(n_1, n_2) \Leftrightarrow \exists\, n_3, n_4 \bullet \\ \quad \mathsf{Head_I}(n_3) \wedge \mathsf{Head_S}(n_4) \wedge \\ \quad\quad \exists\, i \bullet \mathsf{next}^i(n_3, n_1) \wedge \mathsf{next}^i(n_4, n_2)\end{array}$$

Instead, we introduce an additional core binary predicate that relates nodes in the two lists. We call it spec to indicate that the target (a specification node) is the "specification" of the source (an implementation node). This relation changes the semantics of the specification operations: it is set in the **specPush** step and unset in the **specPop** step as shown in Figure 7.5.

The new core predicate spec records a one-to-one correspondence between nodes, so we include integrity rules for functionality and inverse functionality:

$$\forall\, v_1, v_2, v_3 \bullet \mathsf{spec}(v_1, v_2) \wedge \mathsf{spec}(v_1, v_3) \rightarrow \mathsf{eq}(v_2, v_3)$$
$$\forall\, v_1, v_2, v_3 \bullet \mathsf{spec}(v_1, v_3) \wedge \mathsf{spec}(v_2, v_3) \rightarrow \mathsf{eq}(v_1, v_2)$$

Additionally, it is important to record in an abstraction whether a particular node is the source or target of a spec relation. This can be achieved with the instrumentation predicates has[spec] and r_by[spec]. The former is implied by the predicate matching, defined below, so we only define:

$$\mathsf{r\_by}[\mathsf{spec}](v) \Leftrightarrow \exists u \bullet \mathsf{spec}(u, v)$$

7.3.1 Matching Values

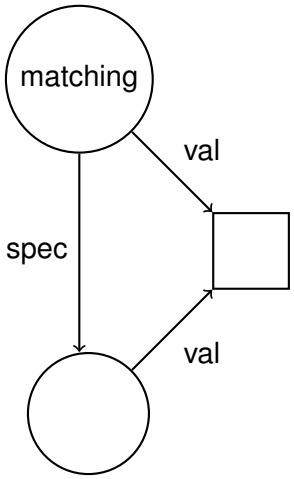
The first property that is lost by the abstraction, but which we wish to preserve, is whether each pair of related nodes in the two lists have the same data value. This property can be recorded by defining an instrumentation predicate for the implementation list nodes that expresses:

> "This node has a value that is shared by its specification counterpart."

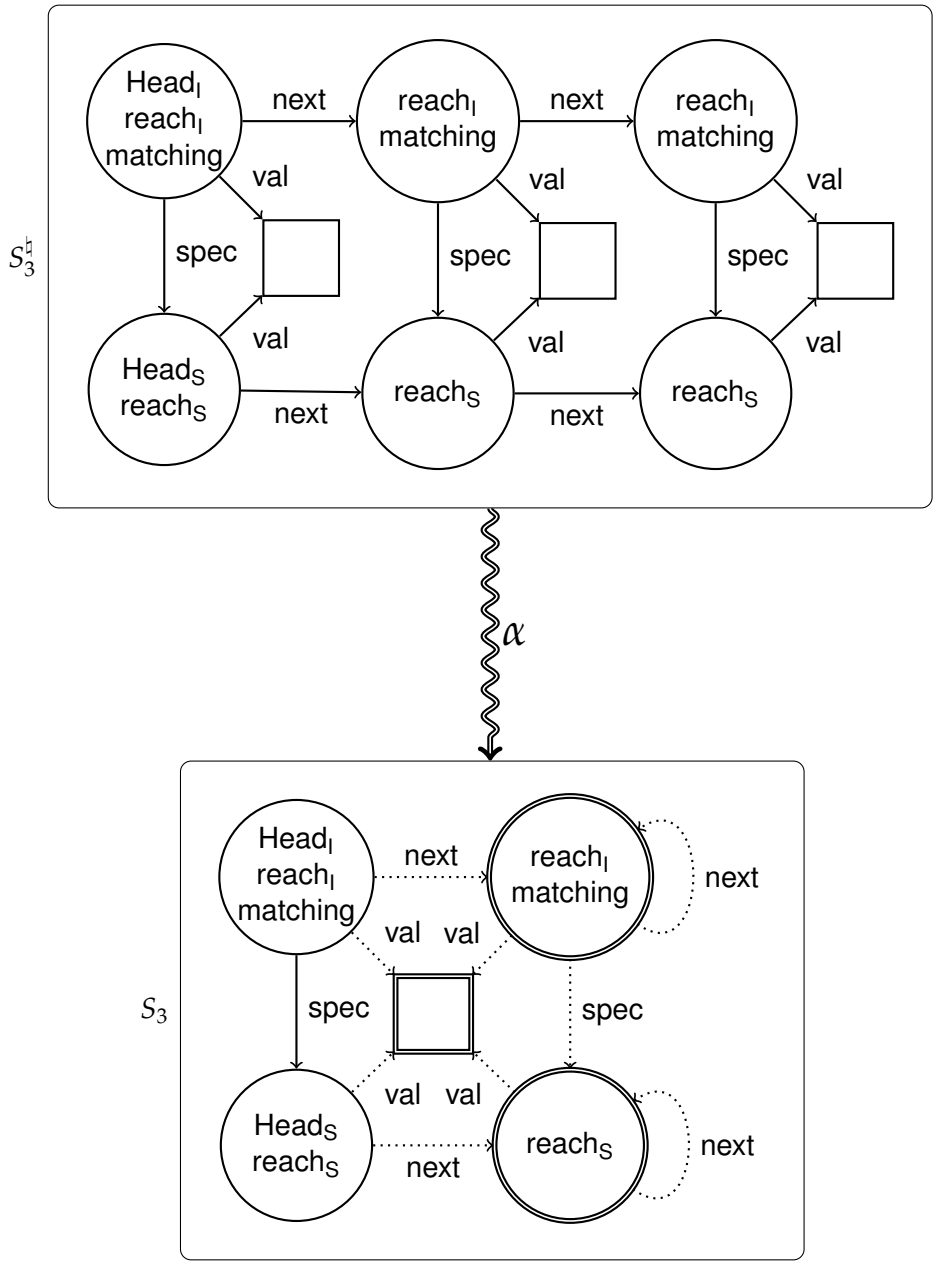
I defined such a predicate and called it matching:

$$\mathsf{matching}(n_1) \Leftrightarrow \exists n_2, d_1 \bullet \mathsf{spec}(n_1, n_2) \wedge \mathsf{val}(n_1, d_1) \wedge \mathsf{val}(n_2, d_1)$$

The predicate records a "triangular" relationship between nodes and data values, as shown in the following diagram:

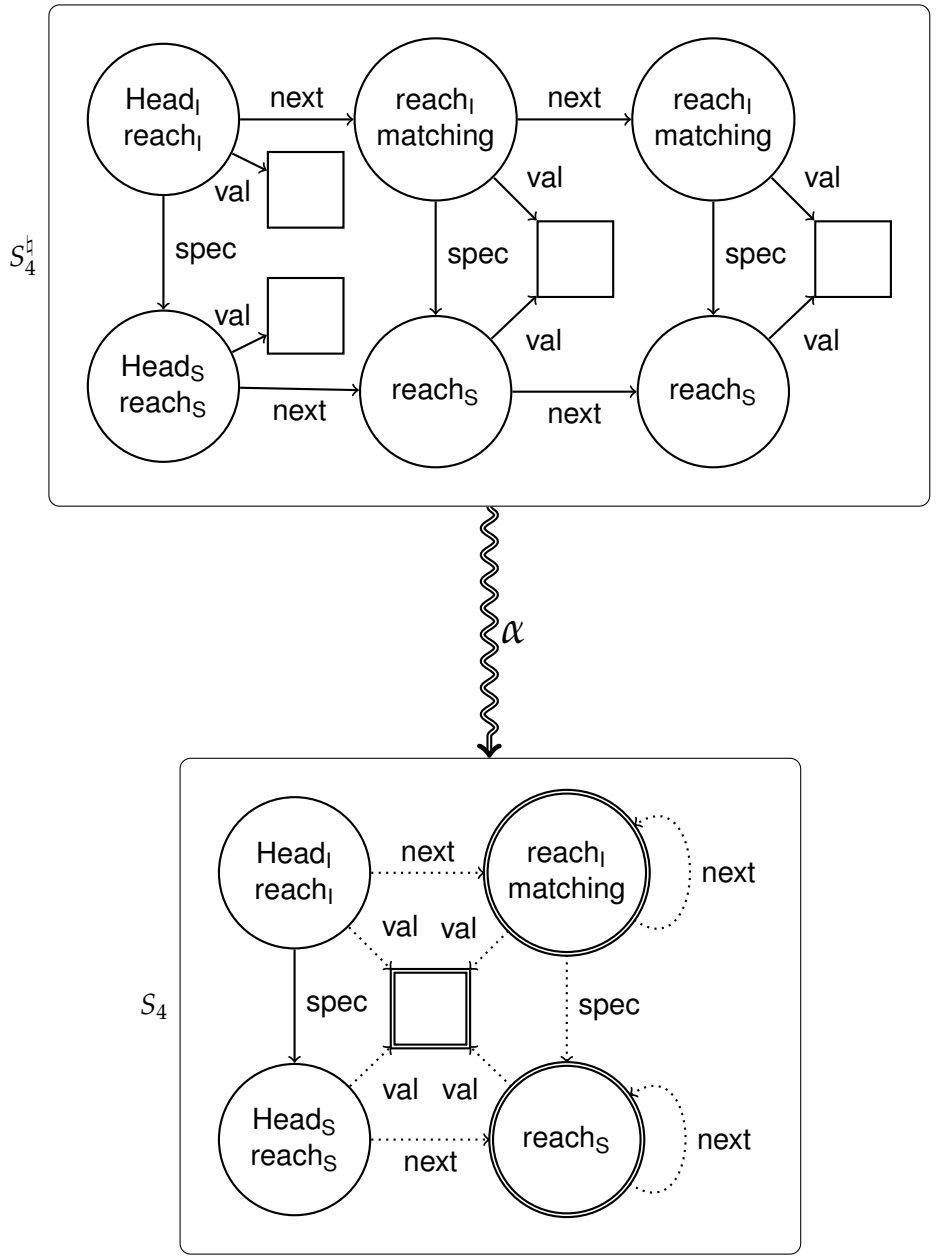


In Figure 7.6 we add matching (and spec) to the concrete states from Figure 7.4, and see that they now have different canonical abstractions. In Figure 7.6a, all of the implementation nodes have matching true, so even



(a) All values match

Figure 7.6: Canonical abstraction of stack lists using the predicate matching: the property of matching values is retained



(b) Head values are different

Figure 7.6: Canonical abstraction of stack lists using the predicate matching: the property of matching values is retained

in S_3 the two lists necessarily contain the same values. In Figure 7.6b, S_4 differs from S_3 because the head node has matching false; the bodies of two lists necessarily contain the same values and the two head nodes necessarily have different values.

The predicate matching alone is not a sufficient addition to verify linearisability however. Consider the two concrete states in Figure 7.7 — they both have three elements, and matching is true for all the implementation list nodes. In $S_5^\natural$, the spec relations have "crossed", so after a Pop operation the head nodes will have different values — another Pop from both lists will trigger a linearisability error. We see that these two states have the same canonical abstraction, so analysis of a linearisable stack can still provide spurious errors.

7.3.2 Ordered Values

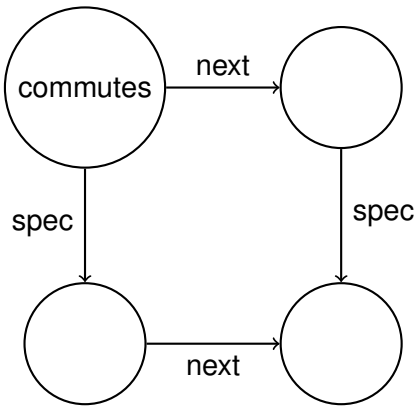
The second property that is lost by abstraction, but which we wish to preserve, is whether the nodes in both lists have the same ordering with respect to the spec predicate. This property can be recorded using an instrumentation predicate on the implementation list nodes that expresses:

> "This node's successor's specification is the same as its specification's successor."

I defined such a predicate and called it commutes:

$$\mathsf{commutes}(n_1) \Leftrightarrow \exists\, n_2, n_3, n_4 \bullet \\ \mathsf{next}(n_1, n_2) \wedge \mathsf{spec}(n_1, n_3) \wedge \mathsf{next}(n_3, n_4) \wedge \mathsf{spec}(n_2, n_4)$$

The predicate records a "square" relationship between nodes, as shown in the following diagram:



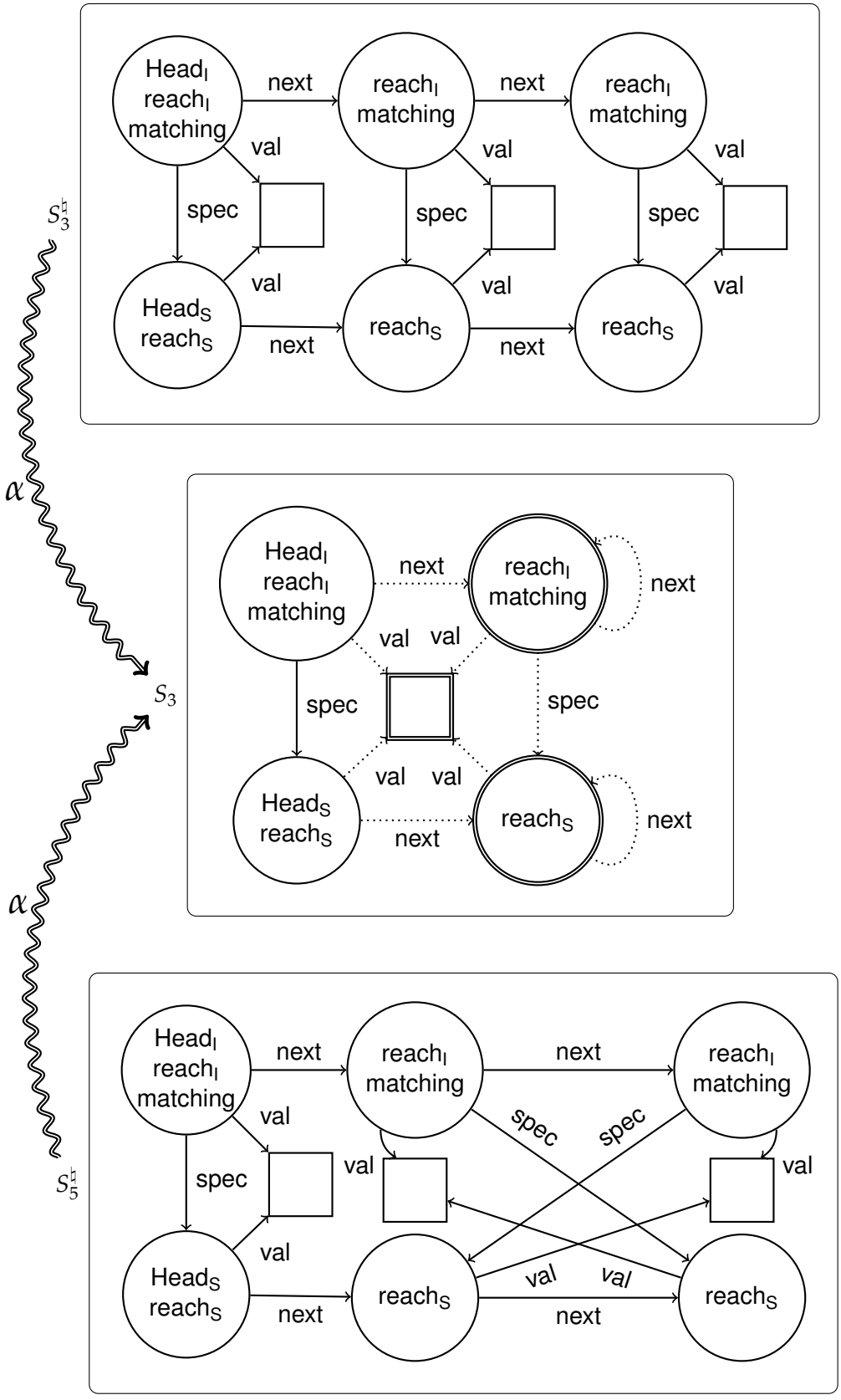


Figure 7.7: Canonical abstraction with "crossed" spec predicates: the property of ordered values is lost

In Figure 7.8 we add commutes to the concrete states from Figure 7.7, and see that they now have different canonical abstractions, in which the orderedness or otherwise of the spec predicates is recorded. In Figure 7.8a, commutes is true for the first two implementation nodes, so the specification list in S_6 has the same values in the same order as the implementation list. In Figure 7.8b, commutes is false for all nodes, so the two lists contain the same values but in different orders. Note that $S_7^\natural$ has the same diagram as $S_5^\natural$, but it is a different and more precise state because commutes is explicitly false for all objects.

7.3.3 Hanging Head

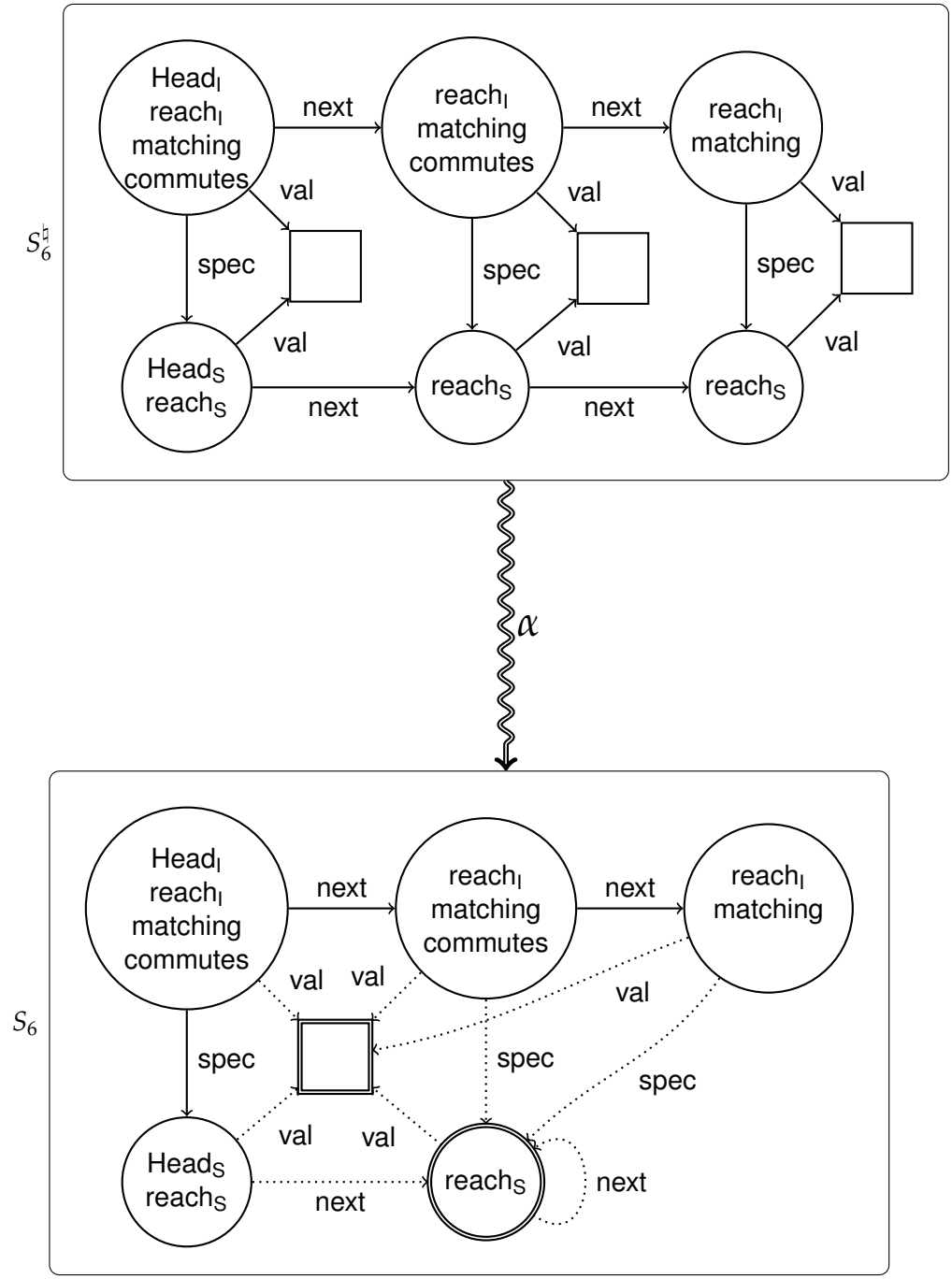
Finally, it is necessary to distinguish the old implementation head node in between the transitions that perform the implementation Push update (CASsucc at location push_9) and the specification Push update (specPush at location push_{11}). Figure 7.9 shows a potential configuration of the lists in between these two transitions, where the old implementation head node (the second node in the list) has been abstracted with the body.[3] Logically, because of the values of the matching and commutes, this abstract state only represents concrete states where the second implementation node is paired with the specification head node. Unfortunately, this fact cannot be established using Focus and Coerce without concretising the whole list, which would focus to an infinite number of states. Hence we must use an instrumentation predicate to distinguish the node object in the abstraction.

I defined a new instrumentation predicate to record the node that is paired with the specification head node:

$$\mathsf{has}_S[\mathsf{spec}](v_1) \Leftrightarrow \exists\, v_2 \bullet \mathsf{Head}_S(v_2) \wedge \mathsf{spec}(v_1, v_2)$$

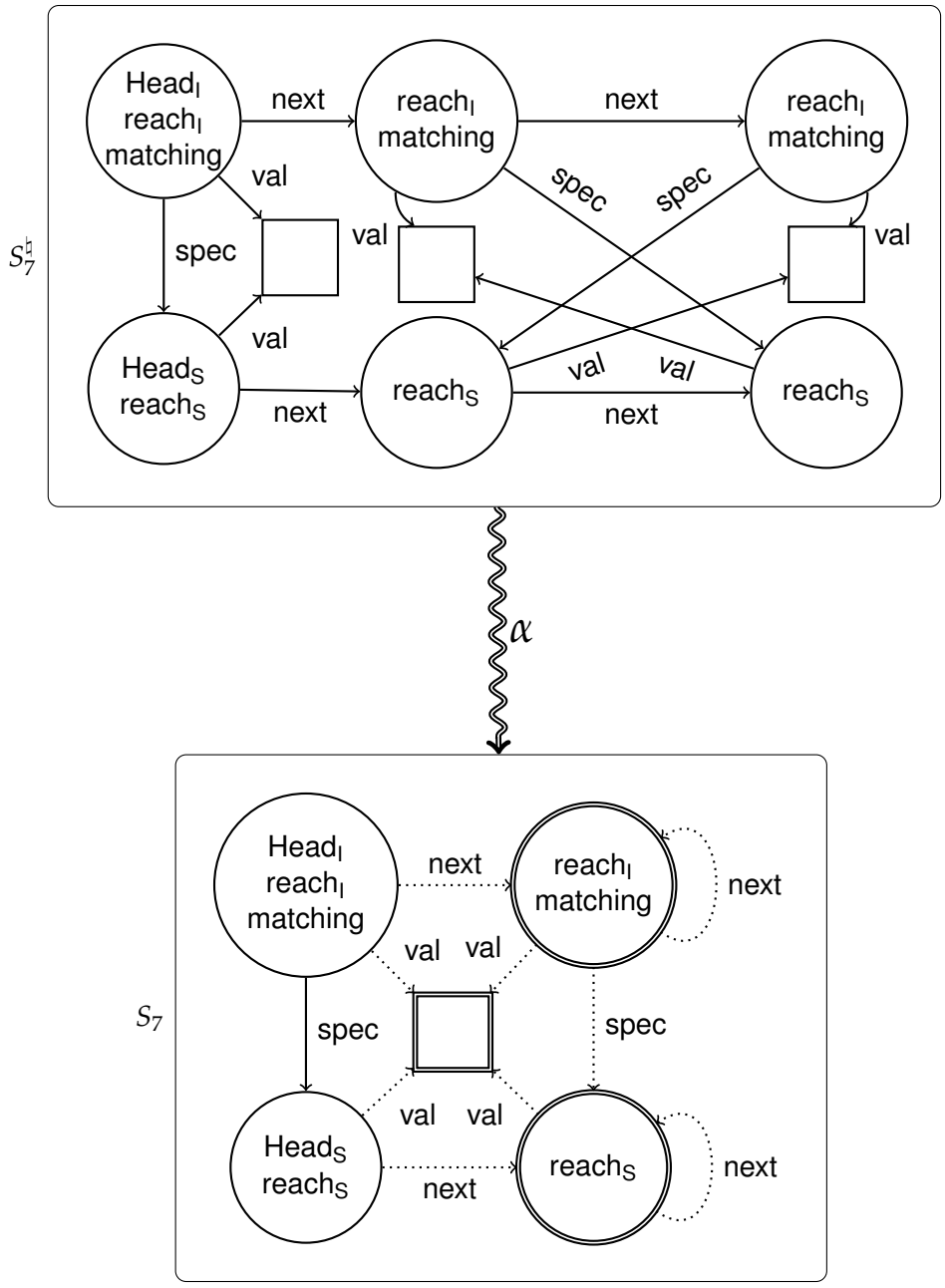
This predicate distinguishes the old implementation head node for the specification update transition, as shown in Figure 7.10. In every other situation it is true only for the implementation head node, so does not otherwise affect the abstraction.

[3] The data values are not shown, in order to make the diagram clearer.



(a) Ordered values

Figure 7.8: Canonical abstraction using the commutes predicate: the property of ordered values is retained



(b) Unordered values

Figure 7.8: Canonical abstraction using the commutes predicate: the property of ordered values is retained

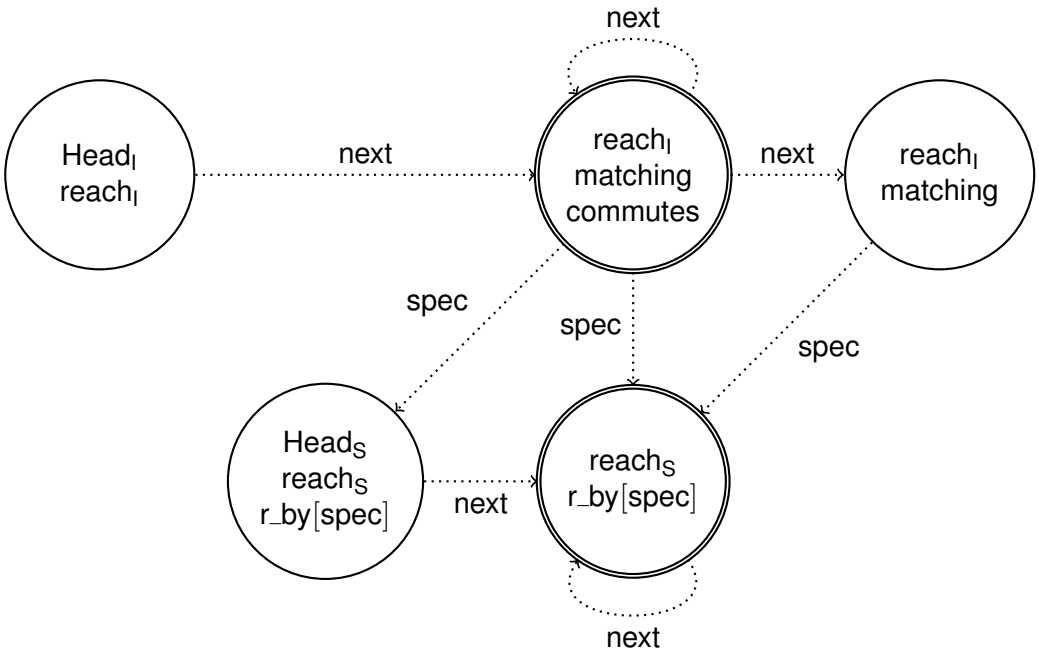


Figure 7.9: In between the implementation and specification push updates the old implementation head node is abstracted with the list body.

7.4 (Un)bounded Threads

As we have constructed the model so far, the abstraction is fine enough to preserve sufficient information about the structure of the two lists for verifying linearisability, but we have not yet considered how to handle multiple threads.

The simplest approach is to introduce a unique unary predicate for each thread, and to define instrumentation predicates to distinguish the objects pointed to by the fields of each thread. Effectively, the threads and other objects are prevented from being abstracted. This is the approach taken by Amit et al. [2007] — e.g. for two threads they define unique unary predicates threadA and threadB, and for each thread field *fld* the instrumentation predicates:

$$\begin{aligned} \mathsf{r\_by}[\mathsf{fld}, \mathsf{threadA}](v) &\Leftrightarrow \exists t \bullet \mathsf{threadA}(v) \wedge \mathsf{fld}(t, v) \\ \mathsf{r\_by}[\mathsf{fld}, \mathsf{threadB}](v) &\Leftrightarrow \exists t \bullet \mathsf{threadB}(v) \wedge \mathsf{fld}(t, v) \end{aligned}$$

Whilst this approach works for models with a fixed number of threads,

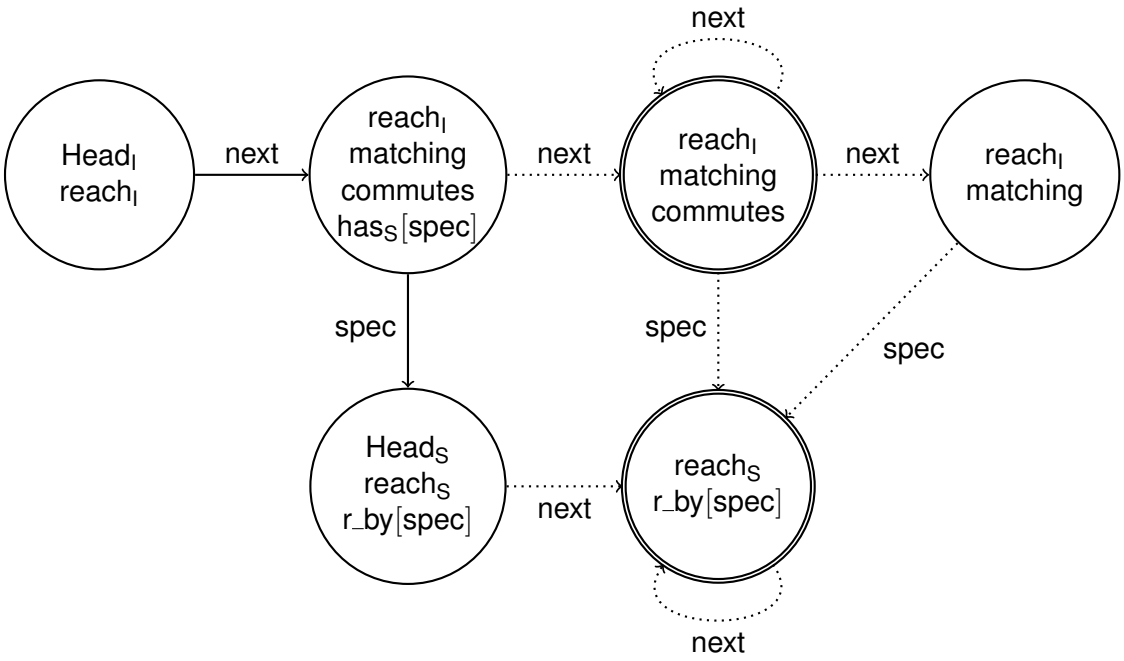


Figure 7.10: With the $\mathsf{has}_S[\mathsf{spec}]$ predicate the old implementation head node is distinguished.

it cannot handle models with unbounded threads. Additionally, the fact that each thread is uniquely distinguished means that the symmetry of the threads is not exploited, resulting in a much larger statespace than necessary due to the exponential number of interleavings.

Better results would be achieved by allowing threads to be abstracted normally (i.e. without the unique naming predicates) but we must first solve two problems:

- how to bound the number of threads without affecting the implicit symmetry reduction of canonical abstraction; and

- what instrumentation predicates to define in order to retain properties of the threads' fields that would otherwise be lost in canonical abstraction.

7.4.1 Bounded Threads

By default, the canonical abstraction of threads gives the option of either one or unbounded threads, depending on whether the thread object in the initial state is a non-summary or summary object, respectively.[4] It is useful to consider models with a thread bound greater than one, and to do so without losing the implicit symmetry reduction that canonical abstraction offers due to having unnamed threads (and other objects).

This can be achieved by including an additional compatibility constraint (recall from Section 4.3.2) that ensures that there are not $bound + 1$ thread objects. E.g. for a bound of 2:

$$\exists\, t_1, t_2, t_3 \bullet \mathsf{is\_thread}(t_1) \wedge \mathsf{is\_thread}(t_2) \wedge \mathsf{is\_thread}(t_3) \\ \wedge \neg\, \mathsf{eq}(t_1, t_2) \wedge \neg\, \mathsf{eq}(t_2, t_3) \wedge \neg\, \mathsf{eq}(t_1, t_3) \rhd 0$$

and for a bound of 3:

$$\exists\, t_1, t_2, t_3, t_4 \bullet \\ \mathsf{is\_thread}(t_1) \wedge \mathsf{is\_thread}(t_2) \wedge \mathsf{is\_thread}(t_3) \wedge \mathsf{is\_thread}(t_4) \\ \wedge \neg\, \mathsf{eq}(t_1, t_2) \wedge \neg\, \mathsf{eq}(t_2, t_3) \wedge \neg\, \mathsf{eq}(t_3, t_4) \\ \wedge \neg\, \mathsf{eq}(t_1, t_3) \wedge \neg\, \mathsf{eq}(t_2, t_4) \wedge \neg\, \mathsf{eq}(t_1, t_4) \rhd 0$$

These constraints can be optimised further. Instead of expressing:

> "Any state with $bound + 1$ threads is invalid",

we can express that

> "in any state with *bound* distinct thread objects, each thread object must be non-summary".

E.g. for a bound of 2:

$$\exists\, t_1, t_2, t_3 \bullet \mathsf{is\_thread}(t_1) \wedge \mathsf{is\_thread}(t_2) \wedge \mathsf{is\_thread}(t_3) \\ \wedge \neg\, \mathsf{eq}(t_1, t_2) \wedge \neg\, \mathsf{eq}(t_2, t_3) \rhd \mathsf{eq}(t_1, t_3)$$

and for a bound of 3:

$$\exists\, t_1, t_2, t_3, t_4 \bullet \\ \mathsf{is\_thread}(t_1) \wedge \mathsf{is\_thread}(t_2) \wedge \mathsf{is\_thread}(t_3) \wedge \mathsf{is\_thread}(t_4) \\ \wedge \neg\, \mathsf{eq}(t_1, t_2) \wedge \neg\, \mathsf{eq}(t_2, t_3) \wedge \neg\, \mathsf{eq}(t_3, t_4) \\ \wedge \neg\, \mathsf{eq}(t_1, t_3) \wedge \neg\, \mathsf{eq}(t_2, t_4) \rhd \mathsf{eq}(t_1, t_4)$$

[4] Or, more generally in the latter case, if the initial state has more than one thread object.

The latter forms ensure that any state with *bound* thread objects has no summary thread objects. In contrast, the previous forms allow such states to contain summary thread objects, which will result in unnecessary computation as Focus creates states with more than *bound* thread objects that are then discarded by Coerce.

Note, however, that the latter form implicitly assumes that the number of thread objects can only increase by one during the Focus function, which is the case for the models constructed in this chapter. If the focus formulas are such that $bound - 1$ thread objects could be focussed to $bound + 1$ (or more) objects, then this constraint will coerce them all to be non-summary but not discard the state, effectively increasing the bound.

7.4.2 Thread Field Properties

When the threads and their fields are abstracted, the relationships between some of the fields' values can be lost. The relationships can be retained through the used of geometric instrumentation predicates.

Snapshots

In the stack's Pop operation, the ssnext field is the next-successor of the ss field when it is read. This property is assumed to persist, so it is not checked before the CAS step at location pop_{38} that attempts to set Head_l to ssnext.

Figure 7.11 shows that this property is not retained in canonical abstraction using the predicates defined so far. Both states (shown without the data values and specification lists) have two threads performing a Pop operation — in $S_8^\natural$, both ssnext predicates are the next-successors of the respective ss predicates, but this is not the case in $S_9^\natural$; nevertheless, both states have the same canonical abstraction (S_8). As a consequence, the CAS transition can remove an arbitrary prefix of the list because ssnext can be concretised at any point.

This information can be preserved using a geometric instrumentation predicate on threads that expresses:

> "This thread's fld_1 and fld_2 fields are non-null and fld_2 is the next-successor of fld_1".

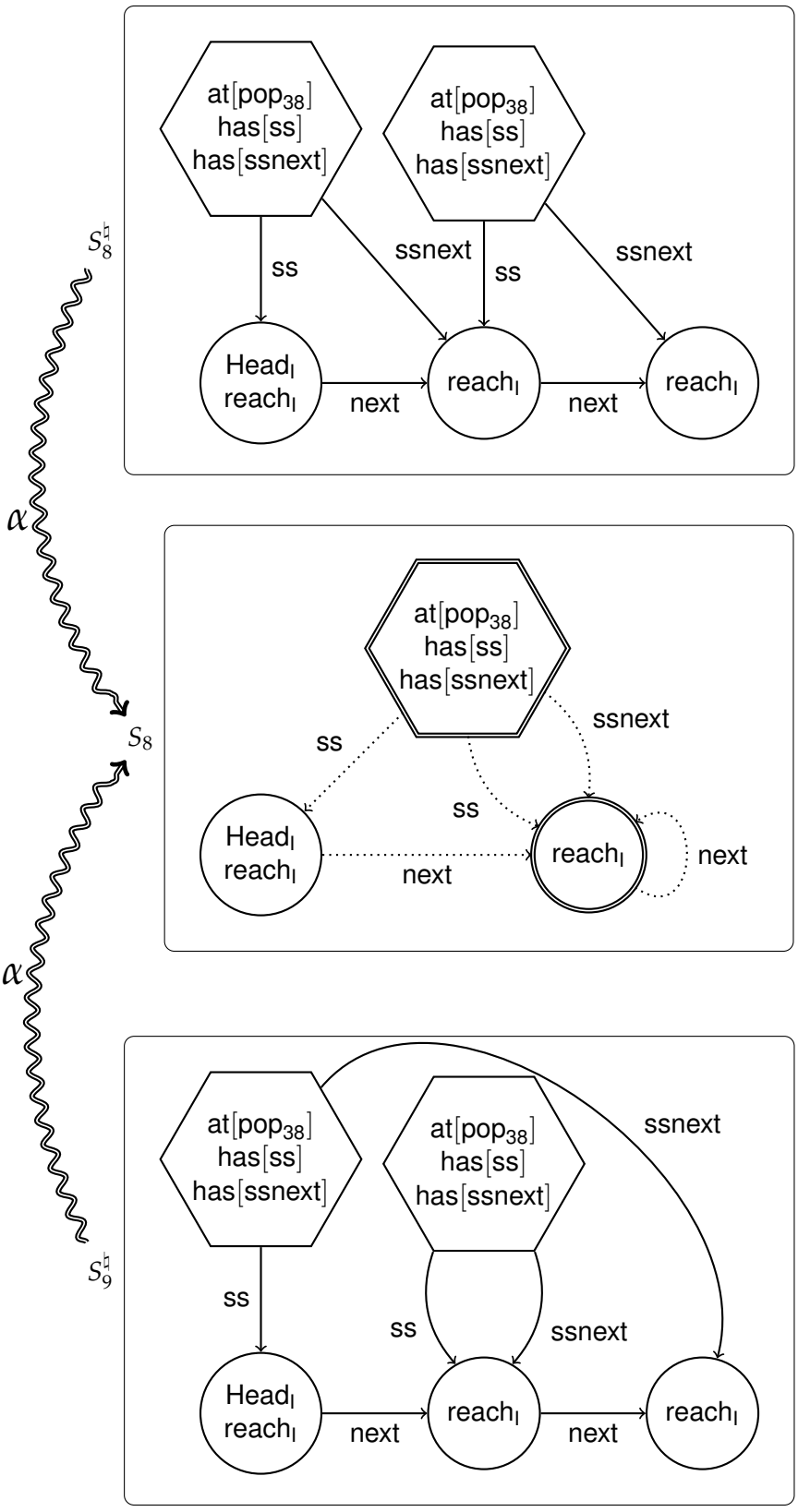
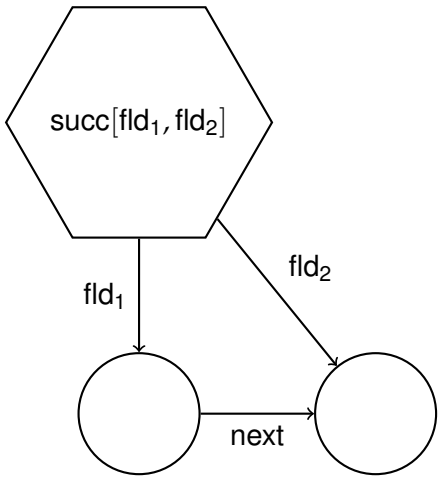


Figure 7.11: Canonical abstraction of threads: the relationships between fields' values are lost

I defined such a predicate:

$$\mathsf{succ}[\mathsf{fld}_1, \mathsf{fld}_2](t_1) \Leftrightarrow \exists\, n_1, n_2 \bullet \mathsf{fld}_1(t_1, n_1) \wedge \mathsf{fld}_2(t_1, n_2) \wedge \mathsf{next}(n_1, n_2)$$

This predicate records a "triangular" relationship between threads and nodes, as shown in the following diagram:



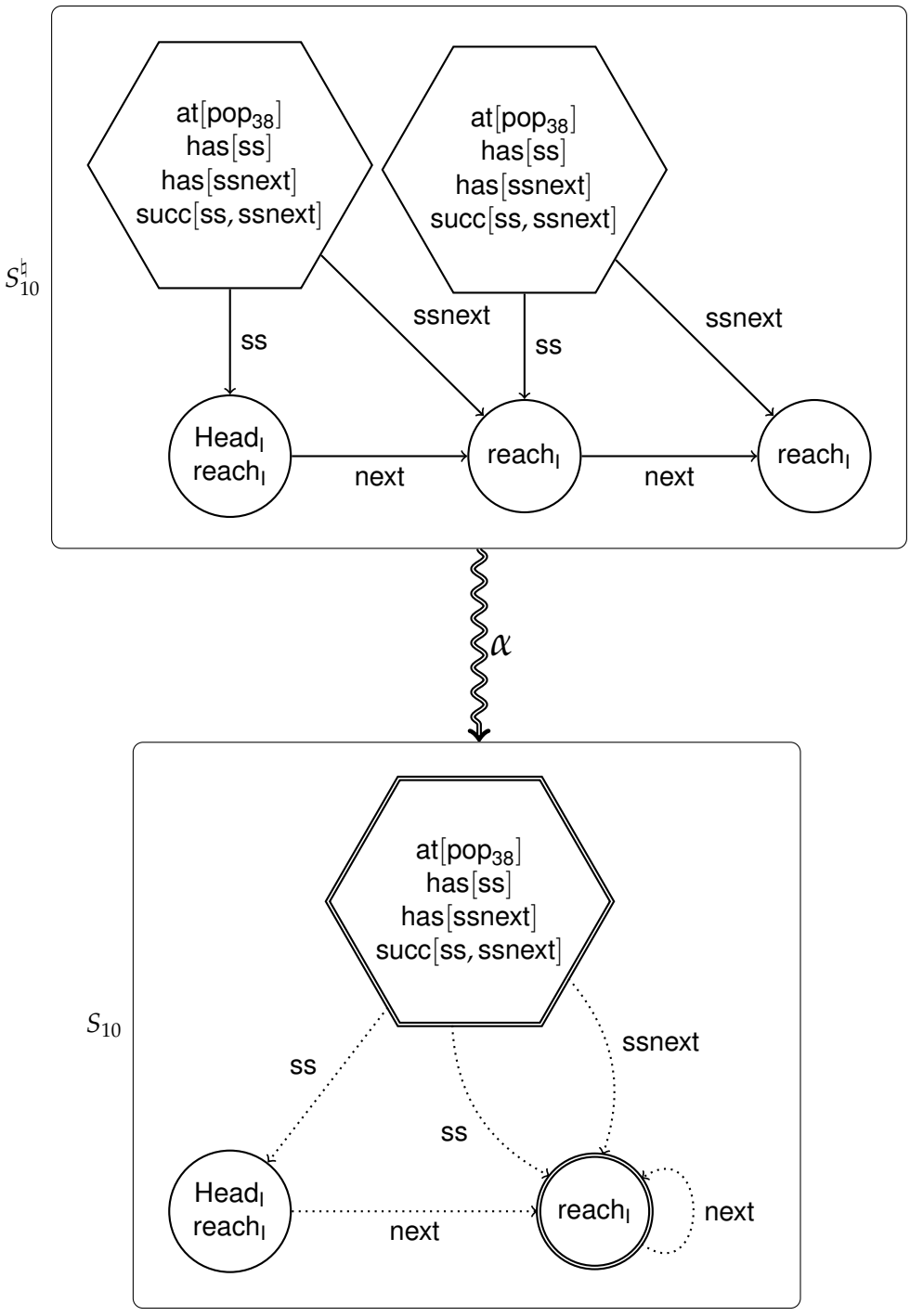
In Figure 7.12, we add succ[ss, ssnext] to the states from Figure 7.11 and can see that they now have different canonical abstractions. Note that S_{11} has the same diagram as S_8, but is a different and more precise state, as succ[ss, ssnext] is explicitly false for all threads (and other objects). Now the CAS step at location pop_{38} will only remove the first node from the stack, as ssnext will always be concretised to the next-successor of ss.

A similar issue arises in the Push operation, as it is implicit at the CAS step at location push_9 that the next field of the n node still points to the same node as ss; this can be resolved by including the predicate succ[n, ss].

Data Values

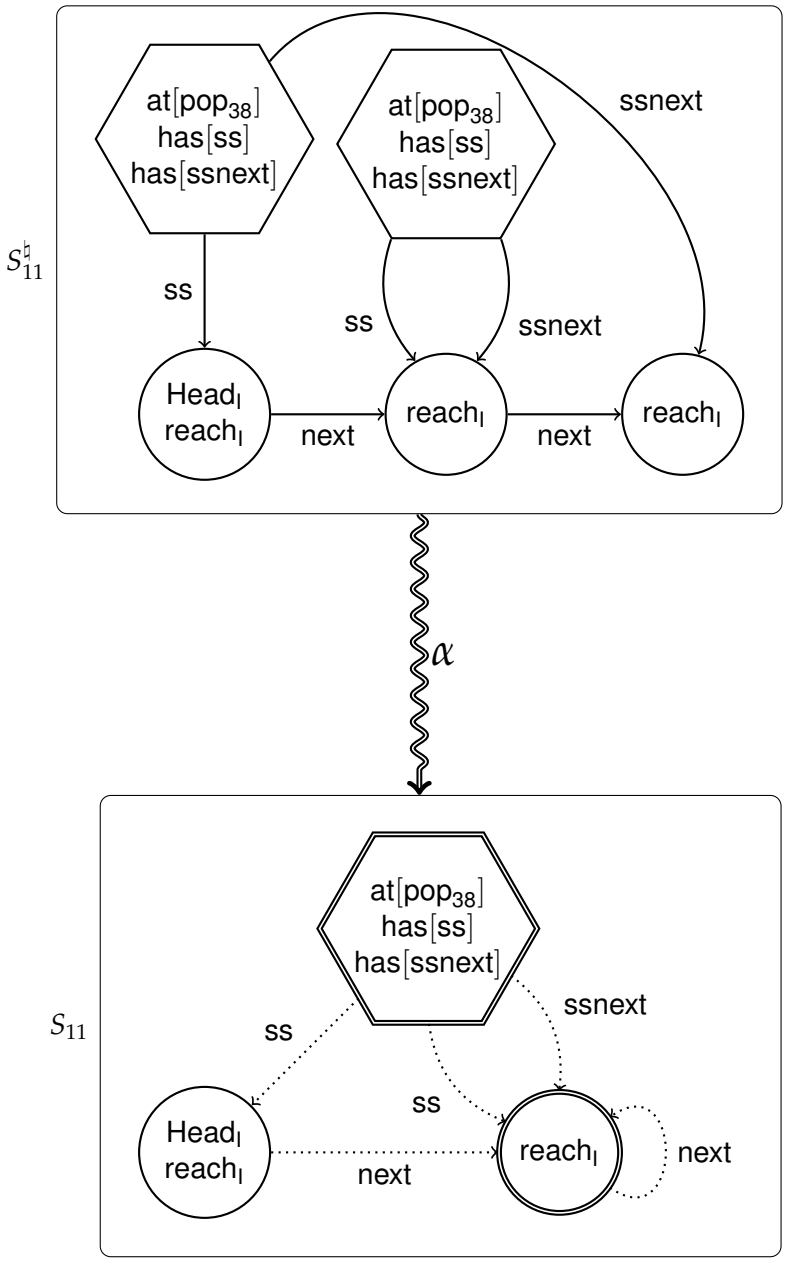
In the stack's push operation, the lv data value that is being pushed is assigned to n.val in the third transition, and is not used again until the specification push operation is performed. At this point, lv is implicitly assumed to still be the same value as n.val, but as with the snapshots, this property is not retained in canonical abstraction.

This property can be retained by introducing an instrumentation predicate that expresses:



(a) All next-successors

Figure 7.12: Canonical abstraction using the succ[ss, ssnext] predicate: the relationships between fields' values are retained



(b) None next-successors

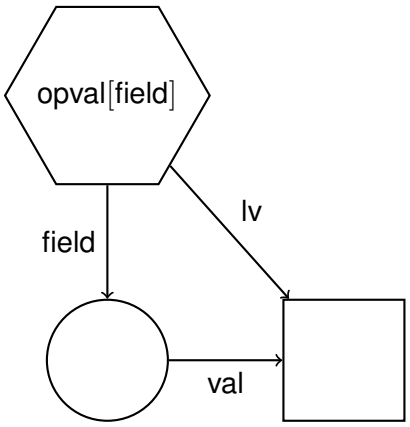
Figure 7.12: Canonical abstraction using the succ[ss, ssnext] predicate: the relationships between fields' values are retained

> "this thread's field node has the same data value (val) as the thread does (lv)".

I defined such a predicate:

$$\mathsf{opval}[\mathsf{field}](v) \Leftrightarrow \exists\, u_1, u_2 \bullet \mathsf{field}(v, u_1) \wedge \mathsf{val}(u_1, u_2) \wedge \mathsf{lv}(v, u_2)$$

This predicate records a "triangular" relationship between threads, nodes and data values, as shown in the following diagram:



I included the predicate opval[n], which records the relationship throughout a push operation between the n and lv fields.

A similar need arises in the pop operation, when lv is assigned to be the ss node's val field before the linearisation point. This relationship is implicitly assumed to be unchanged at the linearisation point, and the value popped from the specification stack is expected to be the same as lv. To ensure that the relationship between ss and lv is retained in the abstraction, I included the predicate opval[ss].

7.5 Initial State

The initial states of the models in Chapter 6 simply contain an empty data structure and a number of idle threads. Such a canonically abstract state for the stack, with all possible data values explicitly represented, is shown in Figure 7.13.

Since no garbage collection is performed in the model, the abstract states record the history of whether a pop operation's linearisation point

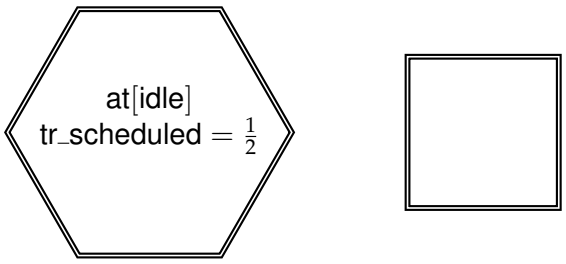


Figure 7.13: Initial state of the stack model (plain)

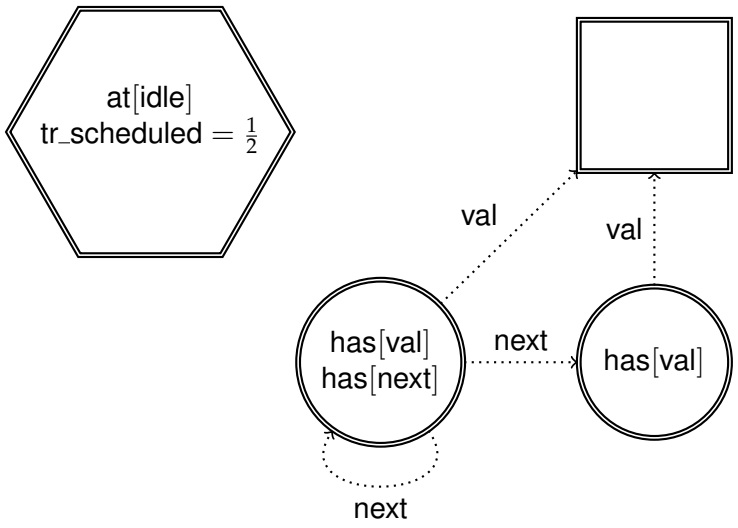


Figure 7.14: Initial state of the stack model (with "garbage" nodes)

has ever been performed or not, indicated by the presence or absence of "garbage" nodes.

To reduce the abstract statespace further, I included garbage nodes in the initial state. As shown in Figure 7.14, there are two summary nodes, distinguished by the has[next] instrumentation predicate. The efficiency of this approach is quantified in Section 7.8.

7.6 Additional Compatibility Constraints

The predicates we have defined so far are theoretically sufficient for performing a canonically abstract verification of the stack algorithm. However, in the Focus/Coerce/Blur approach of TVLA, which only approximates the best transformer (see Section 4.3), the compatibility constraints automatically generated from the instrumentation predicates and integrity rules (see Section 4.3.2) are not always comprehensive enough to sharpen structures or eliminate inconsistent structures.

I have added additional compatibility constraints for the reachability predicates (7.6.1) and the geometric predicates (7.6.2).

7.6.1 Reachability Predicates

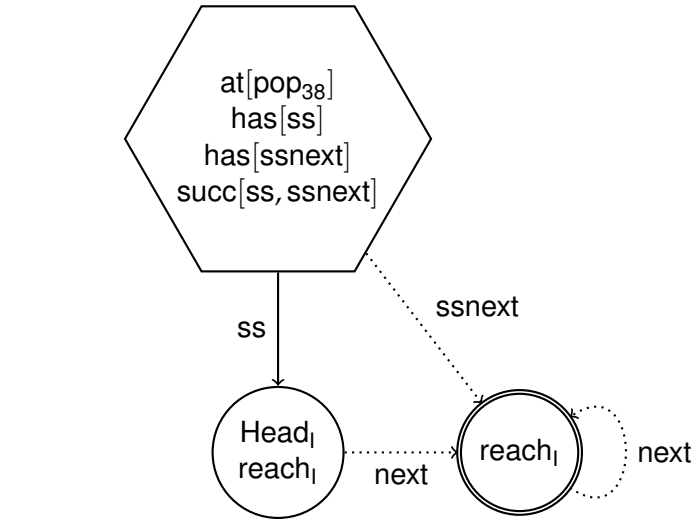
Example Consider the steps of a Pop update shown in Figure 7.15 (with the data values, other threads and the specification list removed for simplicity). The initial state (7.15a) has a single thread and a list with a non-summary head and summary body. Using the focus formula

$$\mathsf{Head}_{\mathsf{I}}(n_1) \wedge \mathsf{next}(n_1, n_2) \wedge \mathsf{ss}(t_1, n_3) \wedge \mathsf{ssnext}(t_1, n_4)$$

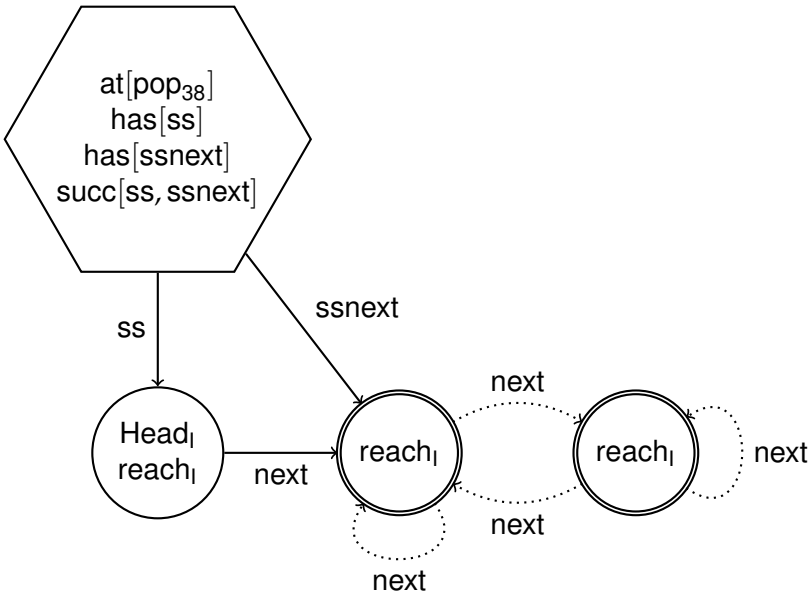
the Focus operation produces a state where the summary node has been split in two (7.15b).[5]

When Coerce is applied, the result is the same, except that the second node in the list is non-summary due to the functionality integrity rules (7.15c). However, the state is not as precise as possible: circ is false for the second node of the list, so the two unknown next predicates that point to it can only be false. Coerce is not able to sharpen these — the next values

[5]Only one of several focussed states is shown, as the unknown next and ssnext predicates can be concretised in several ways.

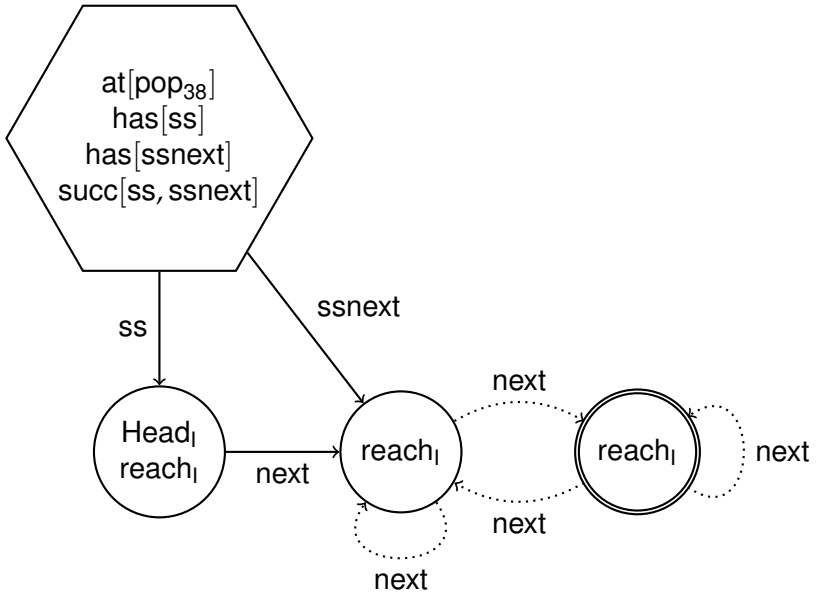


(a) Initial state

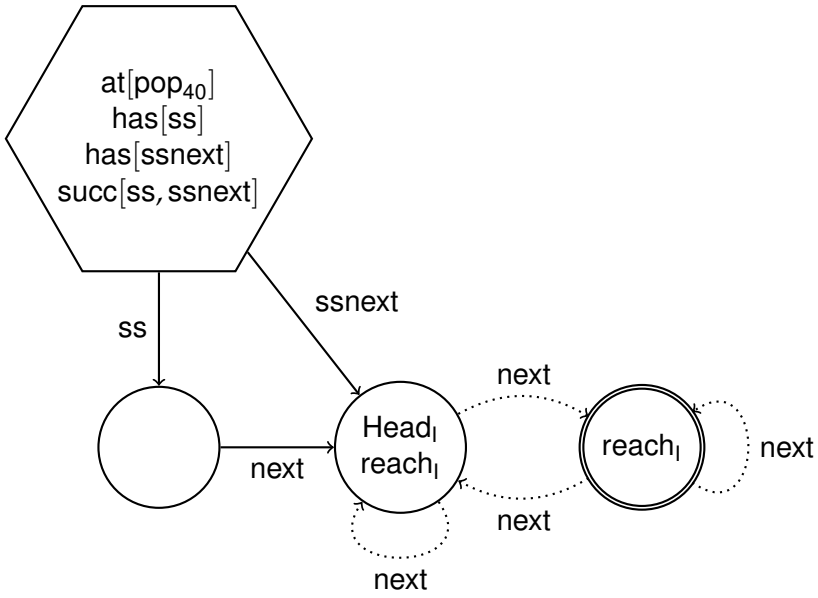


(b) After Focus (one of several states)

Figure 7.15: Steps of a Pop update transition



(c) After Coerce



(d) Final state after Update, Coerce and Blur

Figure 7.15: Steps of a Pop update transition

are unknown, so next^+, and hence the entire bodies of circ's compatibility constraints (recall from Section 4.3.2), evaluate to unknown:

$$\exists\, u \bullet v = u \wedge \mathsf{next}^+(v,u) \rhd \mathsf{circ}(v)$$
$$\forall\, u \bullet v \neq u \vee \neg\, \mathsf{next}^+(v,u) \rhd \neg\, \mathsf{circ}(v)$$

The Head_I predicate is advanced along the list by the Update step (7.15d), which is also the final state as Coerce is still unable to sharpen the unknown next predicates, and the state is already canonically abstract. ■

This example may seen fairly benign — a subsequent transition that focusses the next predicate from Head_I will Coerce the two redundant ones to false — but these unnecessary imprecisions can adversely affect the analysis. In the worst case they can allow the discovery of otherwise unreachable spurious counterexamples. More generally they induce greater computation effort, both by increasing the number of structures that Focus creates and Coerce examines, but also in allowing inconsistent states to be stored and explored.

Example Figure 7.16 shows an example of how a completely inconsistent state can be reached, because of the weakness of the circ constraints. The transition is the assignment of ssnext from location pop_{35} to pop_{38}; for simplicity, we leave out the data values, the specification list and other threads from the diagram.

In the initial state (7.16a) the list has redundant next predicates pointing at the head node, as created in the previous example; the thread's ss field points to somewhere in the body of the list.

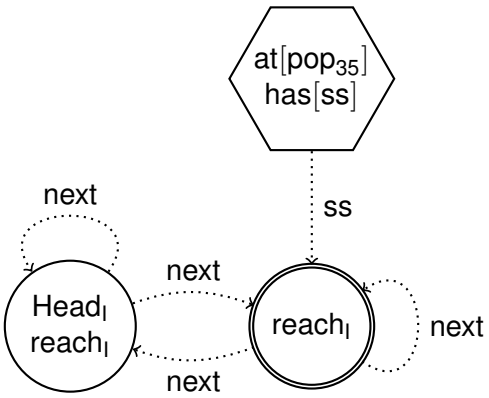
Using the focus formula

$$\mathsf{ss}(t_1,n_1) \wedge \mathsf{next}(n_1,n_2)$$

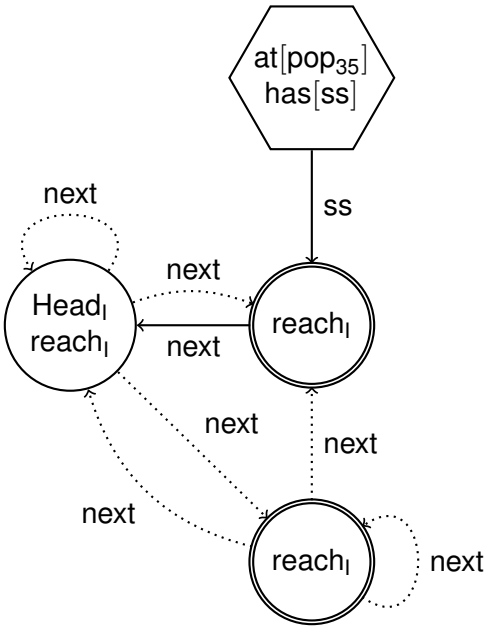
the Focus operation produces a state where the summary node has been split, and the next-successor of ss is the head node (7.16b).[6] This state is inconsistent because circ is false for the head node, so ideally it would be discarded by Coerce.

However, Coerce is not able to discard this state, due to the weakness of the circ compatibility constraints, as before. Coerce instead just sharpens the ss node to be non-summary (7.16c).

[6] Again, this is only one of several states produced by Focus, as the unknown ss and next predicates can be concretised in several ways.

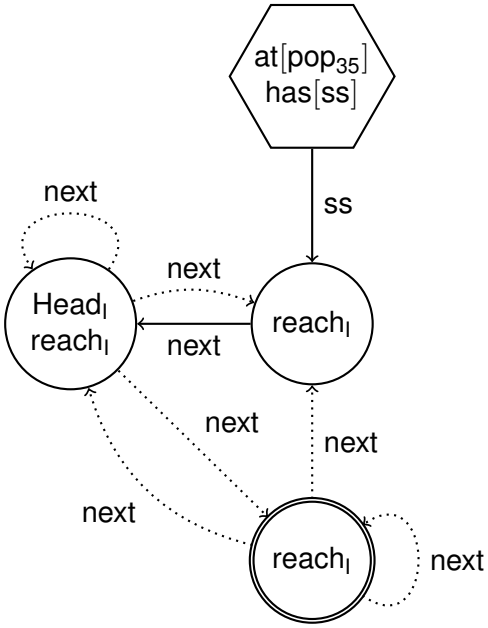


(a) Initial state

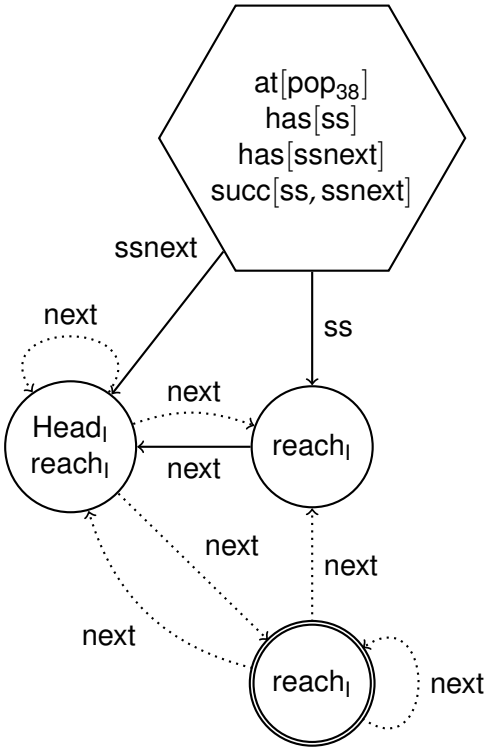


(b) After Focus (one of several states)

Figure 7.16: Transition to an inconsistent state

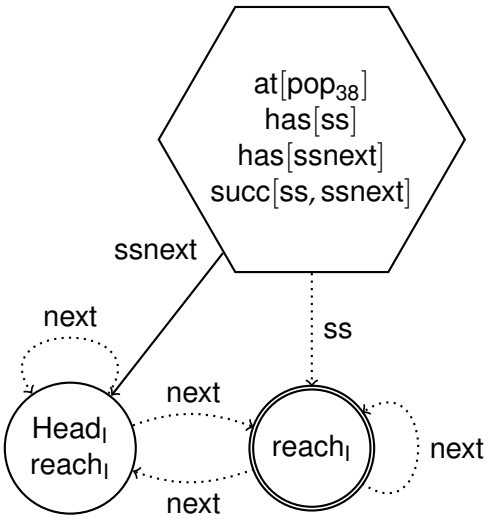


(c) After Coerce



(d) After Update and Coerce

Figure 7.16: Transition to an inconsistent state



(e) Final state, after Blur

Figure 7.16: Transition to an inconsistent state

The Update step sets ssnext as expected (7.16d), and Coerce is unable to discard the state or sharpen any values. Blur merges the two non-head nodes (7.16e).

Thus we end up with a final state (7.16e) that is inconsistent — the predicate succ[ss, ssnext] implies that there is a node reachable from Head_l whose next field points to Head_l, but circ implies that Head_l is not part of a next cycle. ■

There are three possible means for addressing these problems:

- adding further focus formulas to certain transitions, or
- adding further instrumentation predicates, or
- adding further compatibility constraints.

The first option, adding focus formulas, is the least attractive as it is an ad hoc approach that is not readily transferable to other models, and the performance of Focus can be exponential in the size of the focus formulas. Adding further instrumentation predicates has the advantage of being fully automatic, but adds more explicit information in order to discern information that is already logically available. Adding compatibility constraints is often the best approach, as they deal systematically with the

logical information stored in the structure. Constructing constraints is in some sense ad hoc, and can be a source of human error if an invalid rule is chosen, but they can easily be reused in other models that use the same instrumentation predicates.

I chose to define additional compatibility constraints, as I encountered similar or identical issues in the different models, and the constraints could be reused in all of them. When I encountered situations like those in Figure 7.15 and Figure 7.16, I defined a constraint to allow Coerce to sharpen the structure further. I have included the following compatibility constraints in the stack model:

- next is not true reflexively on non-circular nodes:[7]

$$\neg\, \mathsf{circ}(n_1) \wedge n_1 = n_2 \rhd \neg\, \mathsf{next}(n_1, n_2)$$

- For non-circular nodes, next is not symmetric:

$$\neg\, \mathsf{circ}(n_1) \wedge \mathsf{next}(n_1, n_2) \rhd \neg\, \mathsf{next}(n_2, n_1)$$

- A node in a non-circular list cannot point back to the head:

$$\mathsf{Head}_\mathsf{l}(n_1) \wedge \neg\, \mathsf{circ}(n_1) \wedge \mathsf{reach}_\mathsf{l}(n_2) \rhd \neg\, \mathsf{next}(n_2, n_1)$$
$$\mathsf{Head}_\mathsf{S}(n_1) \wedge \neg\, \mathsf{circ}(n_1) \wedge \mathsf{reach}_\mathsf{S}(n_2) \rhd \neg\, \mathsf{next}(n_2, n_1)$$

- A node in a list cannot point to a node that is not in the list:

$$\mathsf{reach}_\mathsf{l}(n_1) \wedge \neg\, \mathsf{reach}_\mathsf{l}(n_2) \rhd \neg\, \mathsf{next}(n_1, n_2)$$
$$\mathsf{reach}_\mathsf{S}(n_1) \wedge \neg\, \mathsf{reach}_\mathsf{S}(n_2) \rhd \neg\, \mathsf{next}(n_1, n_2)$$

The additional compatibility constraints I have defined (here, and later in the thesis) are all relatively small, and appeared to be straightforward consequences of the instrumentation predicate definitions and the predicate properties such as functionality. However, to avoid any risk of mistakenly defining an invalid constraint, I proved their 2-valued validity by hand — see Appendix A.

[7] Note that formulating the rule as $\neg\, \mathsf{circ}(v) \rhd \neg\, \mathsf{next}(v, v)$ would remove unknown loops on summary nodes.

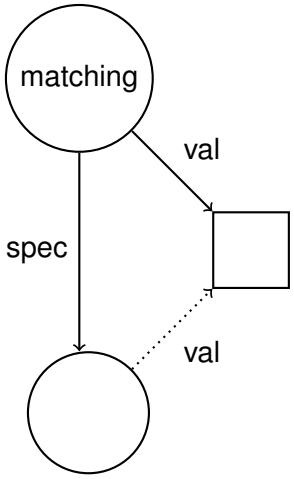


Figure 7.17: Incomplete matching triangle

7.6.2 Geometric Predicates

The automatically generated compatibility constraints (see Section 4.3.2) for the geometric predicates are also not comprehensive enough.

Example Consider the diagram in Figure 7.17 showing a "matching triangle" with one val predicate undefined, and the compatibility constraints generated for matching:

$$\begin{aligned}
\forall\, n_2, d_1 \bullet \neg\, \mathsf{spec}(n_1, n_2) \vee \neg\, \mathsf{val}(n_1, d_1) \vee \neg\, \mathsf{val}(n_2, d_1) &\rhd \neg\, \mathsf{matching}(n_1) \\
\exists\, n_2, d_1 \bullet \mathsf{spec}(n_1, n_2) \wedge \mathsf{val}(n_1, d_1) \wedge \mathsf{val}(n_2, d_1) &\rhd \mathsf{matching}(n_1) \\
\exists\, d_1 \bullet \neg\, \mathsf{matching}(n_1) \wedge \mathsf{val}(n_1, d_1) \wedge \mathsf{val}(n_2, d_1) &\rhd \neg\, \mathsf{spec}(n_1, n_2) \\
\exists\, n_2 \bullet \neg\, \mathsf{matching}(n_1) \wedge \mathsf{spec}(n_1, n_2) \wedge \mathsf{val}(n_2, d_1) &\rhd \neg\, \mathsf{val}(n_1, d_1) \\
\exists\, n_1 \bullet \neg\, \mathsf{matching}(n_1) \wedge \mathsf{spec}(n_1, n_2) \wedge \mathsf{val}(n_1, d_1) &\rhd \neg\, \mathsf{val}(n_2, d_1)
\end{aligned}$$

If Coerce could sharpen the indefinite val to true it would make the analysis more efficient by removing redundant information. However, none of the compatibility constraints are sufficient to allow this, as their bodies all evaluate to $\frac{1}{2}$ or 0.

I defined the following two compatibility constraints, the first of which can be used by Coerce to sharpen the diagram in Figure 7.17, and the second of which can be used to sharpen the corresponding diagram where it

is the other val predicate that is indefinite:

$$\exists\, n_1 \bullet \mathsf{matching}(n_1) \wedge \mathsf{spec}(n_1, n_2) \wedge \mathsf{val}(n_1, d_1) \rhd \mathsf{val}(n_2, d_1)$$
$$\exists\, n_2 \bullet \mathsf{matching}(n_1) \wedge \mathsf{spec}(n_1, n_2) \wedge \mathsf{val}(n_2, d_1) \rhd \mathsf{val}(n_1, d_1)$$

Note that these compatibility constraints rely on the fact that spec and val are functional predicates. However, val is not inversely functional, so the following constraint rule — to sharpen when it is spec that is indefinite — is incorrect, and if added to the analysis could cause a state to be altered or discarded incorrectly, thus preventing part of the statespace from being analysed.

$$\exists\, d_1 \bullet \mathsf{matching}(n_1) \wedge \mathsf{val}(n_1, d_1) \wedge \mathsf{val}(n_2, d_1) \rhd \mathsf{spec}(n_1, n_2)$$

■

The same problem occurs for the other geometric predicates, so I defined the following compatibility constraints:

$$\exists\, t_1 \bullet \mathsf{succ}[\mathsf{ss}, \mathsf{ssnext}](t_1) \wedge \mathsf{ss}(t_1, n_1) \wedge \mathsf{ssnext}(t_1, n_2) \rhd \mathsf{next}(n_1, n_2)$$
$$\exists\, n_1 \bullet \mathsf{succ}[\mathsf{ss}, \mathsf{ssnext}](t_1) \wedge \mathsf{ss}(t_1, n_1) \wedge \mathsf{next}(n_1, n_2) \rhd \mathsf{ssnext}(t_1, n_2)$$
$$\exists\, t_1 \bullet \mathsf{succ}[\mathsf{n}, \mathsf{ss}](t_1) \wedge \mathsf{n}(t_1, n_1) \wedge \mathsf{ss}(t_1, n_2) \rhd \mathsf{next}(n_1, n_2)$$
$$\exists\, n_1 \bullet \mathsf{succ}[\mathsf{n}, \mathsf{ss}](t_1) \wedge \mathsf{n}(t_1, n_1) \wedge \mathsf{next}(n_1, n_2) \rhd \mathsf{ss}(t_1, n_2)$$
$$\exists\, t_1 \bullet \mathsf{opval}[\mathsf{n}](t_1) \wedge \mathsf{n}(t_1, n_1) \wedge \mathsf{lv}(t_1, d_1) \rhd \mathsf{val}(n_1, d_1)$$
$$\exists\, n_1 \bullet \mathsf{opval}[\mathsf{n}](t_1) \wedge \mathsf{n}(t_1, n_1) \wedge \mathsf{val}(n_1, d_1) \rhd \mathsf{lv}(t_1, d_1)$$
$$\exists\, t_1 \bullet \mathsf{opval}[\mathsf{ss}](t_1) \wedge \mathsf{ss}(t_1, n_1) \wedge \mathsf{lv}(t_1, d_1) \rhd \mathsf{val}(n_1, d_1)$$
$$\exists\, n_1 \bullet \mathsf{opval}[\mathsf{ss}](t_1) \wedge \mathsf{ss}(t_1, n_1) \wedge \mathsf{val}(n_1, d_1) \rhd \mathsf{lv}(t_1, d_1)$$
$$\exists\, n_1, n_2 \bullet \mathsf{commutes}(n_1) \wedge \mathsf{next}(n_1, n_2) \wedge \mathsf{spec}(n_1, n_3) \wedge \mathsf{spec}(n_2, n_4) \rhd \mathsf{next}(n_3, n_4)$$
$$\exists\, n_1, n_3 \bullet \mathsf{commutes}(n_1) \wedge \mathsf{next}(n_1, n_2) \wedge \mathsf{spec}(n_1, n_3) \wedge \mathsf{next}(n_3, n_4) \rhd \mathsf{spec}(n_2, n_4)$$
$$\exists\, n_3, n_4 \bullet \mathsf{commutes}(n_1) \wedge \mathsf{spec}(n_1, n_3) \wedge \mathsf{next}(n_3, n_4) \wedge \mathsf{spec}(n_2, n_4) \rhd \mathsf{next}(n_1, n_2)$$

These also rely on the functionality of spec, ss, ssnext, next, nand lv, and the inverse functionality of spec. As with matching, each has one potential constraint that would be incorrect to define, because next and val are not inversely functional.

Th.	Heap Limit (MB)	Time (s)	Ave RAM (MB)	Max RAM (MB)	Uns. States	Stored States	Total States	Breaches
1	800	1	4	6	14	86	209	431
2	800	18	94	252	129	1,312	4,271	20,903
3	800	102	173	336	440	6,493	21,676	121,166
4	800	535	255	479	915	18,564	67,264	394,264
5	2,048	6,409	502	1,184	1,398	36,749	134,991	826,696
6	2,048	143,302	480	1,052	1,727	55,069	203,029	1,264,353
7	4,096	2,625,113	1,182	2,633	1,870	67,334	249,034	1,552,497
∞	1,024	6,524	647	1,057	1,910	74,056	273,610	1,680,866
∞	2,048	1,934	849	1,603	1,910	74,056	273,742	1,680,866

Table 7.1: Stack verification results

7.7 Stack Results

I analysed thread-bounded and unbounded models of the stack algorithm using TVLA 3.0α on a machine with an Intel Core 2 3.0 GHz processor and 4 GB of RAM, running Java 1.6.0 on a 32-bit GNU/Linux operating system. The results are contained in Table 7.1, showing that each model is verified to be a linearisable stack, with an unbounded length and unbounded data values.

The first two columns are the inputs to TVLA — the number of threads and the maximum size of the Java memory allocation pool; the remainder are numbers reported by TVLA's output.

The figures for time and RAM use are largely indicative in general. The Spin model checker uses the same amount of RAM every time a particular analysis is performed; the time varies only based on the processor speed and parallel system activity. In contrast, the time and memory use in TVLA can vary by additional factors, such as how much RAM is available (demonstrated in Table 7.1) and whether a 32-bit or 64-bit platform is used.

Three columns record the numbers of states. The first column records the number of states (outside atomic blocks) where no thread is explicitly scheduled; this is analogous to the states that Spin stores and is perhaps ideally the number of states that would need to be stored in a canonical abstraction analysis. The second column records the number of states that are actually stored by TVLA, including the states during atomic blocks, af-

ter the "schedule" transition and before the "unschedule" transition, when one thread is explicitly scheduled. The third column records the total number of states that TVLA reports encountering, including states merged due to the partially disjunctive analysis; this value can vary slightly between different runs of each analysis. The final column in Table 7.1 records the number of constraint breaches, i.e. the number of times a state was altered or discarded by Coerce.

The principal result of Table 7.1 is that the linearisability of the stack model is able to be verified for unbounded numbers of threads and data values, and for lists of unbounded length. This is the first verification of a nonblocking data structure algorithm using only canonical abstraction.

Each of the thread-bounded models shown is larger than the previous one, but the relative increases in the numbers of states are smaller. However, the time taken for each analysis increases exponentially, indicating that the compatibility constraints used to bound the threads become increasingly inefficient.

I expect that the unbounded model is identical in its statespace to the model bounded to 11 thread objects, though it was impractical for me to verify the models with a thread bound of eight or more, due to the time requirements.[8] Examining the statespace of the unbounded model, the maximum number of thread objects that can occur together in a state is 10 — there can be up to nine thread objects in an unscheduled state, and states within atomic blocks will have an extra thread object because of the different location predicates. However, in a model bounded to 10 threads the bounding compatibility constraint will ensure that whenever 10 thread objects occur together in a state they are non-summary, which will differ from the unbounded model.

Table 7.2 shows the nine canonical thread objects that can occur in an unscheduled state, distinguished by a location predicate and the values of the other abstraction predicates. As might be expected, there are idle states, threads at push_9 with null and non-null head snapshots (ss), and threads at pop_{38} with null and non-null next snapshots (ssnext). For the location pop_{35} there are threads executing the loop for the first time — these have non-null head snapshots (ss) only. There are also threads executing the loop a subsequent time without the fields being reset — they additionally have non-null lv values, and null or non-null next snapshots

[8] Six threads took 40 minutes to complete; seven threads took 30 days. The model with eight threads was manually killed after 10 months.

	doneLP	has[lv]	has[n]	has[ss]	has[ssnext]	opval[n]	opval[ss]	succ[n, ss]	succ[ss, ssnext]
at[idle]	0	0	0	0	0	0	0	0	0
at[push_9]	0	1	1	0	0	1	0	0	0
at[push_9]	0	1	1	1	0	1	$\frac{1}{2}$	1	0
at[pop_{35}]	0	0	0	1	0	0	0	0	0
at[pop_{35}]	0	1	0	1	0	0	$\frac{1}{2}$	0	0
at[pop_{35}]	0	1	0	1	1	0	$\frac{1}{2}$	0	0
at[pop_{35}]	0	1	0	1	1	0	$\frac{1}{2}$	0	$\frac{1}{2}$
at[pop_{38}]	0	1	0	1	0	0	1	0	0
at[pop_{38}]	0	1	0	1	1	0	1	0	1

Table 7.2: Canonical thread objects in unscheduled abstract stack states

(ssnext); the predicates opval[ss] and succ[ss, ssnext] are either false or unknown because the transition that reads the snapshot value does not focus the state sufficiently to determine when these predicates are true.

7.8 Stack Variations

In addition to the model and analyses described so far in this chapter, I constructed and analysed several differing models of the stack algorithm. Some were constructed experimentally, in order to evaluate two or more alternatives; others were constructed deliberately in order to quantify the differences with an approach that was known to be less efficient. They cover variations to TVLA (7.8.1), variations to logical definitions (7.8.2) and variations to the thread interleaving restrictions (7.8.3).

The analyses in this section used the same hardware and software as in Section 7.7 unless specified.

7.8.1 TVLA Changes

Isomorphic State Comparison

The analyses in Section 7.7 used a partially isomorphic comparison between states (see Section 4.5.2). Manevich et al. [2004] report this approach providing reductions between 0 and 99.6% in the number of stored states.

Th.	States (uns.)	%	States (stored)	%	States (total)	%
1	7,037	50,264	38,550	44,826	38,550	18,445
2	>32,166	>24,935	>261,000	>19,893	>261,500	>6,123

Table 7.3: Stack analyses using isomorphic state comparisons

Th.	Heap Limit (MB)	Time (s)	%	Ave RAM (MB)	%	Max RAM (MB)	%
1	800	6	406	4	88	7	131
2	800	518	2,839	38	40	104	41
3	800	4,221	4,138	59	34	183	54
4	800	26,380	4,931	118	46	278	58
∞	2,048	195,733	10,123	421	50	956	60

Table 7.4: Comparison with TVLA 2

For comparison, I attempted to verify the stack models using an isomorphic comparison between states (called a "relational join" in TVLA). Table 7.3 contains the results — the numbers of states along with a percentage comparison with the figures in Table 7.1. For the single-thread model, the partially isomorphic approach provides a 99.8% reduction in the number of stored states. For two threads, TVLA runs out of memory, but even with this limit the reduction provided by the partially isomorphic approach is more than 99.5%.

Clearly, the partially isomorphic approach is necessary to make canonical abstraction a practical technique for verifying nonblocking data structure algorithms.

TVLA 2

Section 4.5.3 discusses some of the enhancements introduced in TVLA 3 for improving Coerce. Bogudlov et al. [2007a] report reductions in time between 35.6% and 98.0% using TVLA 3 compared to TVLA 2, and report nothing about the effects on memory use.

I verified some of the models from Section 7.7 in TVLA 2α. Table 7.4 shows the time and memory results along with percentage comparisons with Table 7.1. Note that the tables display the figures rounded to the

Th.	States (uns.)	%	States (stored)	%	States (total)	%
1	56	400	326	379	904	433
2	>20,240	15,690	>160,168	12,208	>374,000	8,757

Table 7.5: Stack analyses with unnecessary referenced-by instrumentation predicates

nearest second or megabyte, but the percentage comparisons use the unrounded figures reported by TVLA. The time reduction of TVLA 3 over TVLA 2 is 75.3% for a single thread, and then increases with each bound increase, from 96.5% to 99.0%. Conversely, TVLA 3 *increases* memory use, with the non-single thread models using 1.7–2.9 times as much as TVLA 2.

These results indicate that TVLA 3 is vastly more practical for verifying nonblocking data structure algorithms. The analyses are an order of magnitude or more faster, and though they use more memory the increase is a manageable tradeoff.

7.8.2 Definitions

Unnecessary Instrumentation Predicates

In Section 7.2.3, I discussed my selection of the instrumentation predicates used, and how unneeded instrumentation predicates can make the abstraction too fine. To quantify this I analysed a model that further included the instrumentation predicates:

$$\mathsf{r\_by}[\mathsf{ss}], \mathsf{r\_by}[\mathsf{ssnext}], \mathsf{r\_by}[\mathsf{n}], \mathsf{r\_by}[\mathsf{next}], \mathsf{r\_by}[\mathsf{val}]$$

Additionally, I defined the following compatibility constraint, which was needed for the implementation list:

$$\begin{aligned}&\exists\, n_1 \bullet \mathsf{reach}_{\mathsf{I}}(n_1) \wedge \mathsf{reach}_{\mathsf{I}}(n_2) \wedge \neg\, \mathsf{circ}(n_2) \wedge \\ &\qquad \mathsf{next}(n_1, n_2) \wedge \mathsf{reach}_{\mathsf{I}}(n_3) \wedge \neg\, \mathsf{eq}(n_1, n_3) \\ &\rhd \neg\, \mathsf{next}(n_3, n_2)\end{aligned}$$

By recording, in particular, which fields are pointing to each node, the number of canonical node objects the list can contain is increased, and the permutations of these result in an increase in the statespace.

Table 7.5 shows that for one thread the extra instrumentation predicates quadruple the statespace. For two threads TVLA runs out of memory

Th.	States (uns.)	%	States (stored)	%	States (total)	%
1	54	386	320	372	602	288
2	407	316	4,037	308	9,936	233
3	1,271	289	18,347	283	49,825	230
4	2,494	273	49,358	266	137,571	205
∞	4,847	254	179,246	242	532,515	195

Table 7.6: Stack analyses with a pure initial state

after examining an 87-fold increase in states, and storing over 120 times more.

This example shows that instrumentation predicates that refine the abstraction more than necessary for verification can cause an exponential increase in the statespace. Thus the choice of instrumentation predicates can be critical to the success of verification.

Pure Initial State

As described in Section 7.5, I included "garbage" nodes in the initial state to reduce the statespace. For comparison I also verified models with an initial state that contained no node objects. As a result, the presence or absence of garbage nodes (further distinguished by the has[next] predicate) effectively records some of the history of each state, namely whether

- no non-empty pop operations have been performed,
- only non-empty pop operations on singleton stacks have been performed,
- only non-empty pop operations on non-singleton stacks have been performed, or
- non-empty pop operations have been performed on both singleton and non-singleton stacks.

Table 7.6 shows that this produces a statespace roughly two to four times larger.

Th.	States (uns.)	%	States (stored)	%	States (total)	%
2	234	181	2,364	180	6,769	158
3	2,472	562	34,305	528	101,470	468
4	>10,206	>1,115	>196,410	>1,058	>366,000	>544

Table 7.7: Stack analyses with named threads

Named Threads

In Section 7.4, I described two approaches for bounding the number of threads — using a compatibility constraint (which I used in my models) and assigning each thread a unique unary predicate to distinguish them in the abstraction (as used by Amit et al. [2007]). To quantitatively compare the two, I analysed variations of my stack model that had n threads, each with a unique unary predicate. Note that I did not uniquely distinguish the objects pointed to by each of the threads' fields, as Amit et al. [2007] also do.

Table 7.7 shows the results and comparisons. As expected, preventing the threads from being abstracted, and naming the threads to prevent object symmetry from being exploited, produces an exponential increase in the statespaces. For two threads, the statespace is nearly doubled; for three threads it is increased by more than five times; for four threads it is increased by more than 10 times, but TVLA runs out of memory.

Thread Bounding Constraint

In Section 7.4.1, I defined two versions of the compatibility constraint used to bound the number of threads. Table 7.8 shows the results of using the original version ("a state with $bound + 1$ thread objects is invalid") compared to the results from Table 7.1, which used the optimised version ("if there are *bound* thread objects they must be non-summary"). The results show that the optimised version does in general make the models slightly more efficient, though the difference is very minor. There is generally a slight increase in time and memory use, along with a larger number of states examined and constraints breached; in contrast, the numbers of states stored are virtually identical, with no more than a 0.05% increase.

Th.	Heap Limit (MB)	Time (s)	%	Ave RAM (MB)	%	Max RAM (MB)	%
2	800	18	96	108	114	282	112
3	800	108	106	172	100	342	102
4	800	568	106	256	100	463	98
5	2,048	6,532	102	522	104	1,095	93

(a) Resources

Th.	Unsch. States (uns.)	±	Stored States (stored)	±	Total States (total)	%	Breaches	%
2	129	+0	1,312	+0	4,569	107	23,302	111
3	440	+0	6,498	+5	24,669	114	140,510	116
4	915	+0	18,573	+9	72,822	108	443,955	113
5	1,398	+0	36,756	+7	140,677	104	881,964	107

(b) Statespace

Table 7.8: Stack verification results for bounded threads

7.8.3 Full Interleaving

The stack model as presented in Section 7.1 and Figure 7.2 uses manually specified ad hoc partial order reduction to reduce the interleavings between the threads, and thus to reduce the size of the statespace. To quantify the effect that this has, I analysed a model of the stack with the original full interleaving, i.e. the only atomic groups are to ensure that the specification transitions are performed immediately after the linearisation point of the implementation, as shown in Figure 7.18.

Table 7.9 shows that the model with full interleaving is exponentially larger than the model with restricted interleaving, and runs out of memory with a bound of four threads. Restricting the interleaving provided a statespace reduction of 83.2% for two threads and 96.2% for three threads, indicating that this practice is in general likely to be necessary for verification of unbounded threads unless other approaches are used in conjunction.

idle	beginPush()	$push_3$
$push_3$	newNode(n)	$push_4$
$push_4$	assign($n.val, lv$)	$push_6$
$push_6$	assign($ss, Head$)	$push_7$
$push_7$	assign($n.next, ss$)	$push_9$
$push_9$	CASfail($Head, ss$)	$push_6$
atomic {		
$push_9$	CASsucc($Head, ss, n$)	$push_{11}$
$push_{11}$	specPush()	$push_{18}$
}		
$push_{18}$	endPush()	idle
idle	beginPop()	pop_{24}
atomic {		
pop_{24}	assign($ss, Head$)	pop_{25}
pop_{25}	isNotNull(ss)	pop_{35}
pop_{25}	isNull(ss)	pop_{26}
pop_{26}	specPopEmpty()	pop_{48}
}		
pop_{35}	assign($ssnext, ss.next$)	pop_{36}
pop_{36}	assign($lv, ss.val$)	pop_{38}
pop_{38}	CASfail($Head, ss$)	pop_{24}
atomic {		
pop_{38}	CASsucc($Head, ss, ssnext$)	pop_{40}
pop_{40}	specPop()	pop_{48}
}		
pop_{48}	endPop()	idle

Figure 7.18: Transitions of stack model

Th.	States (uns.)	%	States (stored)	%	States (total)	%
1	45	321	148	172	338	162
2	1,511	1,171	7,731	589	25,498	597
3	21,107	4,797	148,191	2,282	567,500	2,618
4	>28,849	>3,153	>282,441	>1,521	>460,800	>685

Table 7.9: Stack verification results with no partial order reduction

7.9 Queue Models

The queue algorithms from Section 2.6.2 are very similar in structure to the stack algorithm, so the model constructed previously in this chapter can be adapted straightforwardly. We describe briefly the three-valued models used for the queues (7.9.1) and the verification results obtained (7.9.2).

7.9.1 Three-Valued Models

We consider the queue algorithms as described in Section 5.2.6 and Figure 5.7, with the sequential specification merged in. Figure 7.19 shows an alternative representation to more closely match the operational semantics used for canonical abstraction; the locations are named to match the line numbers in Figure 5.7. As with the stack model in Section 7.1, some of the atomic blocks have been expanded to effect ad hoc partial order reduction. Figure 7.20 shows the additional update actions used for the invocation, response and specification transitions.

Core Predicates

The queue algorithms are very similar to the stack algorithm, so we use the same, or equivalent, predicates for the list objects and thread fields:

$$\mathsf{is\_data},$$
$$\mathsf{is\_node}, \mathsf{next}, \mathsf{val}, \mathsf{Head}_\mathsf{I}, \mathsf{Tail}_\mathsf{I}, \mathsf{Head}_\mathsf{S}, \mathsf{Tail}_\mathsf{S},$$
$$\mathsf{is\_thread}, \mathsf{doneLP}, \mathsf{doneELP}, \mathsf{lv}, \mathsf{n}, \mathsf{sshead}, \mathsf{sstail}, \mathsf{ssnext}$$

For both algorithms we define the following location predicates:

$$\mathsf{at}[\mathsf{idle}], \mathsf{at}[\mathsf{enq}_3], \mathsf{at}[\mathsf{enq}_4], \mathsf{at}[\mathsf{enq}_6], \mathsf{at}[\mathsf{enq}_7], \mathsf{at}[\mathsf{enq}_8],$$
$$\mathsf{at}[\mathsf{enq}_9], \mathsf{at}[\mathsf{enq}_{11}], \mathsf{at}[\mathsf{enq}_{13}], \mathsf{at}[\mathsf{enq}_{20}], \mathsf{at}[\mathsf{enq}_{24}], \mathsf{at}[\mathsf{enq}_{25}]$$

For the original algorithm only we define the following location predicates:

$$\mathsf{at}[\mathsf{deq}_{30}], \mathsf{at}[\mathsf{deq}_{31}], \mathsf{at}[\mathsf{deq}_{33}], \mathsf{at}[\mathsf{deq}_{34}], \mathsf{at}[\mathsf{deq}_{35}],$$
$$\mathsf{at}[\mathsf{deq}_{40}], \mathsf{at}[\mathsf{deq}_{41}], \mathsf{at}[\mathsf{deq}_{42}], \mathsf{at}[\mathsf{deq}_{43}], \mathsf{at}[\mathsf{deq}_{46}],$$
$$\mathsf{at}[\mathsf{deq}_{48}], \mathsf{at}[\mathsf{deq}_{50}], \mathsf{at}[\mathsf{deq}_{52}], \mathsf{at}[\mathsf{deq}_{61}]$$

atomic{		
idle	beginEnqueue()	enq_3
enq_3	newNode(n)	enq_4
enq_4	assign($n.val, lv$)	enq_6
enq_6	assign($sstail, Tail$)	enq_7
}		
atomic{		
enq_7	assign($ssnext, sstail.next$)	enq_8
}		
atomic{		
enq_8	isEqual($sstail, Tail$)	enq_9
enq_8	isNotEqual($sstail, Tail$)	enq_6
enq_9	isNull($ssnext$)	enq_{11}
enq_9	isNotNull($ssnext$)	enq_{20}
}		
atomic{		
enq_{20}	CASfail($Tail, sstail$)	enq_6
enq_{20}	CASsucc($Tail, sstail, ssnext$)	enq_6
}		
atomic{		
enq_{11}	CASfail($sstail.next, ssnext$)	enq_6
enq_{11}	CASsucc($sstail.next, ssnext, n$)	enq_{13}
enq_{13}	specEnqueue()	enq_{24}
}		
atomic{		
enq_{24}	CASfail($Tail, sstail$)	enq_{25}
enq_{24}	CASsucc($Tail, sstail, n$)	enq_{25}
enq_{25}	endEnqueue()	idle
}		

(a) Enqueue operation

Figure 7.19: Transitions of queue models

atomic{		
idle	beginDequeue()	deq_{30}
deq_{30}	assign($sshead, Head$)	deq
}		
atomic{		
deq_{31}	assign($sstail, Tail$)	deq_{33}
}		
atomic{		
deq_{33}	assign($ssnext, sshead.next$)	deq_{34}
deq_{34}	isNull($ssnext$)	deq_{35}
deq_{34}	isNotNull($ssnext$)	deq_{40}
deq_{35}	specDequeueEmpty()	deq_{40}
}		
atomic{		
deq_{40}	isEqual($sshead, Head$)	deq_{41}
deq_{40}	isNotEqual($sshead, Head$)	deq_{30}
deq_{41}	isEqual($sshead, sstail$)	deq_{42}
deq_{41}	isNotEqual($sshead, sstail$)	deq_{48}
deq_{42}	isNull($ssnext$)	deq_{43}
deq_{42}	isNotNull($ssnext$)	deq_{46}
deq_{43}	endDequeueEmpty()	idle
}		
atomic{		
deq_{46}	CASfail($Tail, sstail$)	deq_{30}
deq_{46}	CASsucc($Tail, sstail, ssnext$)	deq_{30}
}		
atomic{		
deq_{48}	assign($lv, ssnext.val$)	deq_{50}
deq_{50}	CASfail($Head, sshead$)	deq_{30}
deq_{50}	CASsucc($Head, sshead, ssnext$)	deq_{52}
deq_{52}	specDequeue()	deq_{61}
deq_{61}	endDequeue()	idle
}		

(b) Original dequeue operation

Figure 7.19: Transitions of queue models

atomic{

idle	beginDequeue()	deq_{67}
deq_{67}	assign($sshead, Head$)	deq_{69}

}
atomic{

deq_{69}	assign($ssnext, sshead.next$)	deq_{70}
deq_{70}	isNull($ssnext$)	deq_{71}
deq_{70}	isNotNull($ssnext$)	deq_{76}
deq_{71}	specDequeueEmpty()	deq_{76}

}
atomic{

deq_{76}	isEqual($sshead, Head$)	deq_{77}
deq_{76}	isNotEqual($sshead, Head$)	deq_{67}
deq_{77}	isNull($ssnext$)	deq_{78}
deq_{77}	isNotNull($ssnext$)	deq_{81}
deq_{78}	endDequeueEmpty()	idle

}
atomic{

deq_{81}	assign($lv, ssnext.val$)	deq_{83}
deq_{83}	CASfail($Head, sshead$)	deq_{67}
deq_{83}	CASsucc($Head, sshead, ssnext$)	deq_{85}
deq_{85}	specDequeue()	deq_{94}

}
atomic{

deq_{94}	assign($sstail, Tail$)	deq_{95}
deq_{95}	isEqual($sshead, sstail$)	deq_{96}
deq_{95}	isNotEqual($sshead, sstail$)	deq_{98}

}
atomic{

deq_{96}	CASfail($Tail, sstail$)	deq_{98}
deq_{96}	CASsucc($Tail, sstail, ssnext$)	deq_{98}
deq_{98}	endDequeue()	idle

}

(c) Simplified dequeue operation

Figure 7.19: Transitions of queue models

beginEnqueue() true $\longrightarrow$
$doneLP := \text{false};\ lv \in \textbf{DATA}$

specEnqueue() $\neg\, doneLP \longrightarrow$
$doneLP := \text{true};$
$spec.Tail.next := \text{newNode}(lv, \text{null});$
$spec.Tail := spec.Tail.next$

endEnqueue() $doneLP \longrightarrow$
$lv, sstail, ssnext, n := \text{null}$

beginDequeue() true $\longrightarrow$
$doneLP, doneELP := \text{false};\ lv := \text{null}$

specDequeueEmpty() $\neg\, doneLP \wedge spec.Head = spec.Tail \longrightarrow$
$doneELP := \text{true}$

specDequeue() $\neg\, doneLP \wedge spec.Head \neq spec.Tail \wedge spec.Head.next.val = lv \longrightarrow$
$doneLP := \text{true};\ spec.Head := spec.Head.next$

endDequeueEmpty() $doneELP \wedge \neg\, doneLP \longrightarrow$
$lv, sshead, sstail, ssnext := \text{null}$

endDequeue() $doneLP \longrightarrow$
$lv, sshead, sstail, ssnext := \text{null}$

Figure 7.20: Additional update operations used in queue models

For the simplified algorithm only we define the following location predicates:

$$\begin{array}{l}\mathsf{at}[\mathsf{deq}_{67}], \mathsf{at}[\mathsf{deq}_{69}], \mathsf{at}[\mathsf{deq}_{70}], \mathsf{at}[\mathsf{deq}_{71}], \mathsf{at}[\mathsf{deq}_{76}],\\ \mathsf{at}[\mathsf{deq}_{77}], \mathsf{at}[\mathsf{deq}_{78}], \mathsf{at}[\mathsf{deq}_{81}], \mathsf{at}[\mathsf{deq}_{83}], \mathsf{at}[\mathsf{deq}_{85}],\\ \mathsf{at}[\mathsf{deq}_{94}], \mathsf{at}[\mathsf{deq}_{95}], \mathsf{at}[\mathsf{deq}_{96}], \mathsf{at}[\mathsf{deq}_{98}]\end{array}$$

We also define the spec predicate for relating nodes in the specification and implementation lists.

Additionally, I defined the same, or equivalent, integrity rules and compatibility constraints as in Sections 7.2.2 and 7.6.

Instrumentation Predicates

The same properties of the two lists are required to verify the queue algorithm as for the stack, so I defined the same instrumentation predicates for these:[9]

$$\mathsf{has}[\mathsf{val}], \mathsf{has}[\mathsf{next}], \mathsf{reach}_\mathsf{I}, \mathsf{reach}_\mathsf{S}, \mathsf{circ},$$
$$\mathsf{r\_by}[\mathsf{spec}], \mathsf{matching}, \mathsf{commutes}, \mathsf{has}_\mathsf{S}[\mathsf{spec}]$$

For the thread fields, I defined the same or equivalent linear predicates. For each field, it is necessary to know at some point whether it is null or not. The other properties of n in Enqueue are the same as in Push. Thus:

$$\mathsf{has}[\mathsf{lv}], \mathsf{has}[\mathsf{n}], \mathsf{waiting}, \mathsf{shared}[\mathsf{n}],$$
$$\mathsf{has}[\mathsf{sshead}], \mathsf{has}[\mathsf{sstail}], \mathsf{has}[\mathsf{ssnext}]$$

The queues have similar properties that require geometric predicates to preserve sufficient information in the abstract states. In the CAS steps at locations enq_{20}, enq_{24}, deq_{46}, deq_{50}, deq_{83} and deq_{96}, the old and new values are implicitly assumed to be next-successors. Thus I defined the predicates:

$$\mathsf{succ}[\mathsf{sshead}, \mathsf{ssnext}], \mathsf{succ}[\mathsf{sstail}, \mathsf{ssnext}], \mathsf{succ}[\mathsf{sstail}, \mathsf{n}]$$

In the specification actions for enqueues and non-empty dequeues, the value of lv is assumed to remain the same as the val of the node pointed to by n or ssnext, respectively. Thus, I defined the predicates:

$$\mathsf{opval}[\mathsf{n}], \mathsf{opval}[\mathsf{ssnext}]$$

One property requires a novel predicate not defined so far. After the comparison at locations $\mathsf{deq}_{41}/\mathsf{deq}_{95}$ the equality or otherwise of sshead and sstail is lost by the abstraction. I defined the following instrumentation predicate, which allows the property to be retained in the canonically abstract states.

$$\mathsf{same}[\mathsf{sshead}, \mathsf{sstail}](t) \Leftrightarrow \exists\, n \bullet \mathsf{sshead}(t, n) \wedge \mathsf{sstail}(t, n)$$

[9] The reachability predicates are, as before, recording reachability from the head nodes. Since the tail nodes are always the final or penultimate nodes in the list it was not necessary to define additional reachability predicates from these.

Deq	Th.	Heap Limit (MB)	Time (s)	Ave RAM (MB)	Max RAM (MB)	Uns. States	Stored States	Total States	Breaches
O	1	800	1	13	30	26	115	227	345
O	2	800	393	260	476	3,770	24,271	73,874	849,174
O	3	2,048				>25k	>235k	>564k	
S	1	800	1	14	33	26	117	228	361
S	2	800	83	189	354	1,506	10,746	28,856	147,080
S	3	2,048				>22k	>230k	>632k	

Table 7.10: Queue verification results

As for most of the other instrumentation predicates (see Section 7.6) the automatically generated compatibility constraints for this predicate are not comprehensive enough. I defined the following two constraint formulas:

$$\mathsf{same}[\mathsf{sshead}, \mathsf{sstail}](t) \wedge \mathsf{sshead}(t, n) \rhd \mathsf{sstail}(t, n)$$
$$\mathsf{same}[\mathsf{sshead}, \mathsf{sstail}](t) \wedge \mathsf{sstail}(t, n) \rhd \mathsf{sshead}(t, n)$$

7.9.2 Results

I analysed the queue algorithms using the same software and hardware as for the stack in Section 7.7. The results for the queues with the original (O) and simplified (S) dequeue operations are contained in Table 7.10, showing that linearisability has been verified for unbounded lists and data values, and one or two threads. For three or more threads, TVLA runs out of RAM. Clearly, the queue models are much larger than the stack model; this is due to the greater number of list configurations, thread locations and thread properties.

One interesting, and probably unexpected, observation is that the two-thread model with the original dequeue operation is much larger than the model with the simplified dequeue operation. The single-thread models are very similar in size, and the simplified dequeue allows Head_l and Tail_l to "cross over" (recall from Section 2.6.2), resulting in a larger number of list configurations.

However, because the simplified dequeue only reads Tail_l at the end, there are fewer ways to distinguish its thread objects — the instrumentation predicates defined using sstail are all false for the 11 locations deq_{67}

to deq_{94}. In contrast, the model with the original dequeue may have has[sstail] either true or false at locations deq_{30} and deq_{31} (depending whether or not the loop is being repeated) and may have succ[sstail, ssnext] and same[sshead, sstail] either true or false in the six locations deq_{30} to deq_{34} and deq_{40} to deq_{41}. This results in the model having a larger number of canonical thread objects. An examination of the statespaces of the two models reveals that there are 104 distinct canonical thread objects among the 3,770 unscheduled states of the original dequeue model, but only 38 among the 1,506 unscheduled states of the simplified dequeue model.

The disparity between the models' numbers of canonical thread objects indicates that not only will the original dequeue model be larger than the simplified model for higher thread bounds, but that the difference will increase exponentially with each thread bound increase.

7.10 Related Work

The first use of canonical abstraction for analysing nonblocking data structures was by Yahav and Sagiv [2003], who attempted to verify the five safety properties that Michael and Scott [1996] state in their informal argument for correctness of their queue algorithm. These properties do not prove linearisability; furthermore, the formalisations of the properties are incorrect and mutually inconsistent.[10]

Amit et al. [2007] analysed the same nonblocking data structures as I have (plus two lock-based data structures), verifying linearisability for the stack algorithm with three threads and the queue algorithms with two threads (limiting to 1.5 GB of RAM). They combine canonical abstraction with an additional approach called "delta heap abstraction": the relationship between each pair of implementation and specification nodes and their identical value is represented in the state graph by a single object. Delta heap abstraction requires each push/enqueue etc. to be for a unique value, whereas my approach can represent data values being entered into the list multiple times. Their analyses use unique predicates to distinguish each thread and its field values (see Section 7.4); as shown in Section 7.8.2, this is exponentially more expensive than using the geometric instrumentation predicates I have defined.

[10] For example, one property asserts that the Head node is never pointed to by a next field, and another asserts that at the end of a Dequeue operation the next field of the thread's head snapshot points to the new Head node.

Manevich et al. [2008] combine canonical abstraction with graph decomposition (see Section 4.5.5). Adapting the approach of Amit et al. [2007], they are able to verify linearisability for the stack algorithm with 20 threads (limiting to 2 GB of RAM), and for the (simplified) queue algorithm with 15 threads (limiting to 16 GB of RAM). Graph decomposition is an orthogonal approach, and could be applied to my models to make the analyses more efficient; this is left for future work.

Berdine et al. [2008] combine the above approaches with an additional approach called "quantified canonical abstraction" to verify linearisability for unbounded threads. Like graph decomposition, the approach splits the state into (overlapping) subgraphs, each containing the data structure and one non-summary thread. Unlike graph decomposition, each subgraph can represent an unbounded number of identical subgraphs, thus the bounded number of subgraphs together can represent states with unbounded numbers of threads. By partitioning each thread into a separate subgraph, they do not need to define instrumentation predicates, such as the ones I introduced in Section 7.4.2, in order to preserve information lost when threads are abstracted together. Extending the models of Amit et al. [2007], and limiting to 2 GB of RAM, Berdine et al. [2008] were able to verify linearisability for the stack algorithm, but ran out of memory for the queue. Extending the models of Manevich et al. [2008], using graph decomposition to create smaller subgraphs, they were able to verify linearisability for both the stack algorithm (with an 80% reduction in statespace) and queue algorithm. This is the first published work to verify linearisability for unbounded threads using canonical abstraction, though it uses two additional approaches to do so.

7.11 Conclusion

In this chapter I have, for the first time, verified linearisability for a concurrent data structure using only canonical abstraction. That this is possible was not previously known, and in order to achieve this, I defined several novel instrumentation predicates to preserve information about "geometric" relationships between objects in each state. These types of instrumentation predicates are not unique to concurrent data structures so could well be useful for analysing other types of systems with canonical abstraction.

I applied the approach to analysing a stack and two queue algorithms, but it will be adaptable to other linked list based data structure algo-

rithms. Some additional instrumentation predicates that would be needed for different types of data structures have been investigated by other authors. For example, Sagiv et al. [2002, Section 5] provide some instrumentation predicates for preserving the structure of doubly linked lists (used by Detlefs et al. [2000] and Ladan-Mozes and Shavit [2004, 2008]), and Lev-Ami et al. [2000] discuss some predicates for preserving properties of ordered data values (used by Heller et al. [2005, 2007]).

The models produced by this approach do appear to be relatively large — the stack model could be verified comfortably with 2 GB of RAM, but the queues required more RAM for three threads or more. The greatest factor in the size of a model's statespace is the number of canonical thread objects required, due to the exponential number of permutations of these that are stored. One option to address these permutations would be to employ graph decomposition [Manevich et al., 2008]; with this approach, I think that the analyses of my models would be competitive with those of Berdine et al. [2008]. In Chapter 8, I introduce an alternative approach to reducing these permutations, by abstracting all the thread objects into a single summary object.

Chapter 8

Collapsing Threads Safely with Soft Invariants

In Chapter 7, we saw that canonical abstraction can be used to verify linearisability of nonblocking linked-list based data structures. The statespaces of the models, whilst finite, are still relatively large, and the queue models were too large to be verified with 2 GB of RAM. One of the largest factors in the size of the statespace is the number of canonical thread objects — the various permutations of these that can be present in a state contribute exponentially to the total number of states stored. This statespace explosion can be countered by removing all of the unary predicates on threads from the set of abstraction predicates $\mathcal{A}$ — all of the thread objects will be abstracted (or "collapsed") to a single summary object. Of course, doing this naively would make the abstraction too coarse, as the act of distinguishing the thread objects preserves information that is lost when the threads are abstracted together. This information can still be preserved in the collapsed thread object however, by defining appropriate instrumentation predicates.

I have defined instrumentation predicates with the intent of recording invariant properties of the threads at each location. I have called them "soft invariants" as they do not impose the invariant properties on the model like integrity rules (thus requiring independent verification that the properties are actually invariant), but merely record, through statespace exploration, whether each property is invariant in the system or not.

This chapter begins in Section 8.1 with an overview of the soft invariant instrumentation predicates. In Section 8.2, I apply the approach to the stack models from Chapter 7 (Sections 7.1–7.7 and 7.8.3). In Section 8.3,

I apply the approach to the queue models from Section 7.9. Finally, in Section 8.4, I conclude and discuss some related work.

8.1 Overview

The canonical objects in an abstract state can implicitly record a variety of properties, thus preserving the properties during abstraction. Recall Table 7.2 on page 181, which records the canonical thread objects present in the unscheduled states of the stack model with restricted interleaving. From this we can infer a number of properties, e.g. at location push_9:

- $\mathsf{has}[\mathsf{n}]$ is always true
- $\mathsf{has}[\mathsf{ssnext}]$ is always false
- $\mathsf{succ}[\mathsf{n}, \mathsf{ss}]$ is true if $\mathsf{has}[\mathsf{ss}]$ is true

In this chapter, we reduce the statespace of each model by removing all of the thread predicates from the set of abstraction predicates $\mathcal{A}$ (except for is_thread and tr_scheduled). When we do this though, the above three properties are not preserved by the abstraction. Figure 8.1a shows part of a state with three formerly canonical abstract thread objects. With the new abstraction they are merged into a single canonical thread object where each predicate is unknown (8.1b). From this object it is possible to concretise a thread at push_9 that contradicts each of the three properties observed above (8.1c).

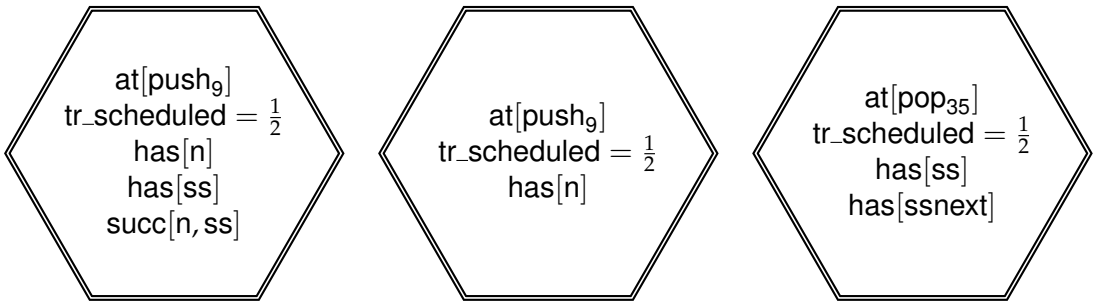
8.1.1 Instrumentation Predicates

The properties noted above may be explicitly recorded within an abstract state by defining specific instrumentation predicates. We can record, for each thread:

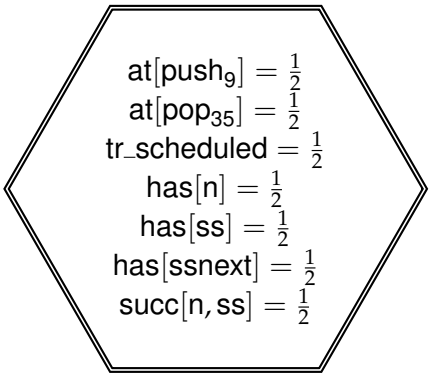
> "if this thread is at loc_X, then property φ holds".

Thus, I defined instrumentation predicates, called soft invariants, of the form:

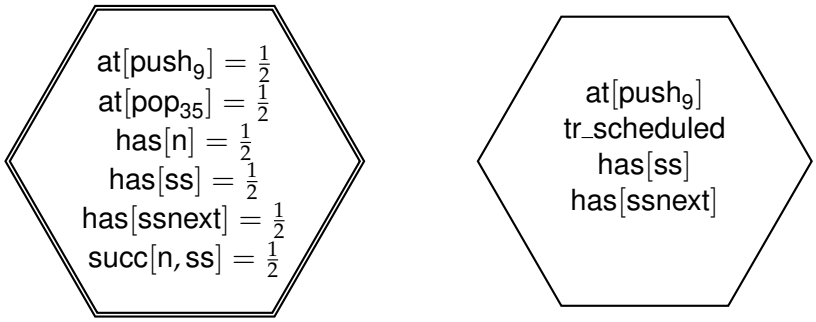
$$\mathsf{if}[\mathsf{loc_X}, \mathsf{p}](v) \Leftrightarrow \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{loc_X}](v) \rightarrow \varphi_{\mathsf{p}}(v))$$



(a) Three previously canonical thread objects



(b) Collapsed thread object



(c) Thread with different properties is concretised

Figure 8.1: Collapsing threads: loss of precision

To record the above three properties, I defined the following three soft invariant instrumentation predicates:

$$\begin{aligned}
&\mathsf{if}[\mathsf{push_9}, \mathsf{has}[\mathsf{n}]](v) \Leftrightarrow \\
&\quad \mathsf{is_thread}(v) \wedge (\mathsf{at}[\mathsf{push_9}] \rightarrow \mathsf{has}[\mathsf{n}](v)) \\
&\mathsf{if}[\mathsf{push_9}, \mathsf{has}[\mathsf{ssnext}]](v) \Leftrightarrow \\
&\quad \mathsf{is_thread}(v) \wedge (\mathsf{at}[\mathsf{push_9}] \rightarrow \neg\, \mathsf{has}[\mathsf{ssnext}](v)) \\
&\mathsf{if}[\mathsf{push_9}, \mathsf{succ}[\mathsf{n}, \mathsf{ss}]](v) \Leftrightarrow \\
&\quad \mathsf{is_thread}(v) \wedge (\mathsf{at}[\mathsf{push_9}] \rightarrow (\mathsf{has}[\mathsf{ss}](v) \rightarrow \mathsf{succ}[\mathsf{n}, \mathsf{ss}](v)))
\end{aligned}$$

The conditional form of these predicates means that they are true for all threads at other locations. Thus the predicate will record the invariant property in all thread objects. Figure 8.2a shows the three thread objects from Figure 8.1a with the new instrumentation predicates all true. Now when the threads are collapsed (8.2b), the new instrumentation predicates are still all true, which prevents a thread being concretised that contradicts any of the properties.

Figure 8.3a shows the state of Figure 8.1a with the following instrumentation predicate defined:

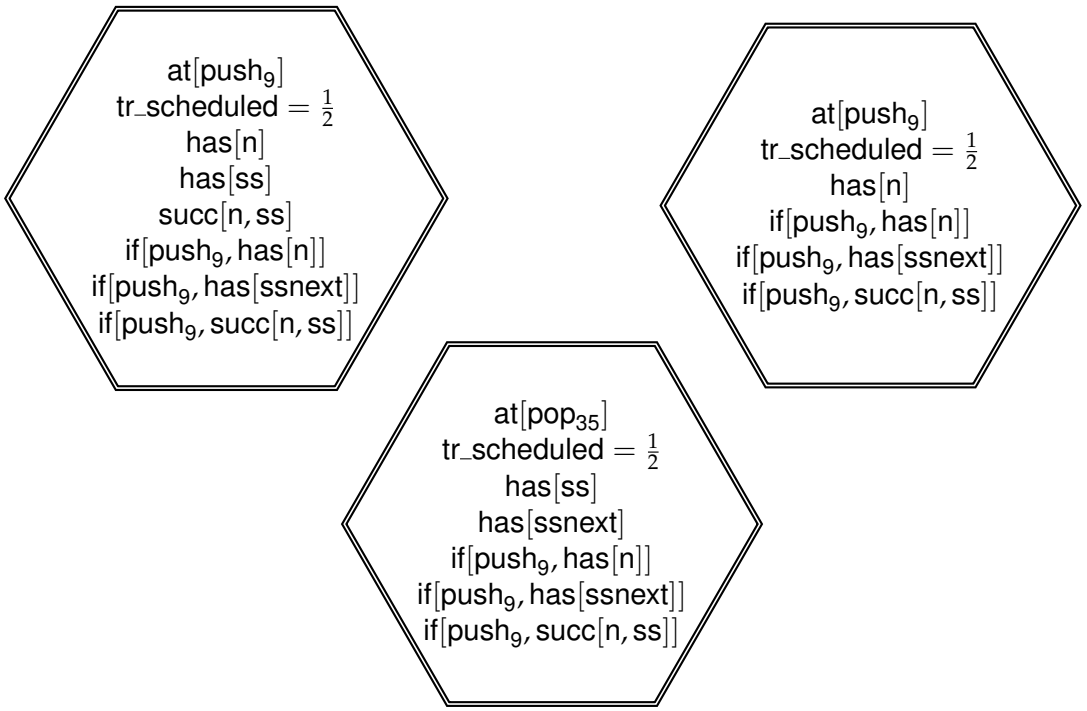
$$\mathsf{if}[\mathsf{push_9}, \mathsf{has}[\mathsf{ss}]](v) \Leftrightarrow \mathsf{is\_thread}(v) \rightarrow (\mathsf{at}[\mathsf{push_9}](v) \rightarrow \mathsf{has}[\mathsf{ss}](v))$$

I do not include the soft invariant predicates in $\mathcal{A}$, so when the threads are collapsed (8.3b), $\mathsf{if}[\mathsf{push_9}, \mathsf{has}[\mathsf{ss}]]$ becomes unknown, because the property it is trying to preserve is not actually invariant. Consequently, when a thread at $\mathsf{push_9}$ is concretised, $\mathsf{has}[\mathsf{ss}]$ can be either true or false.

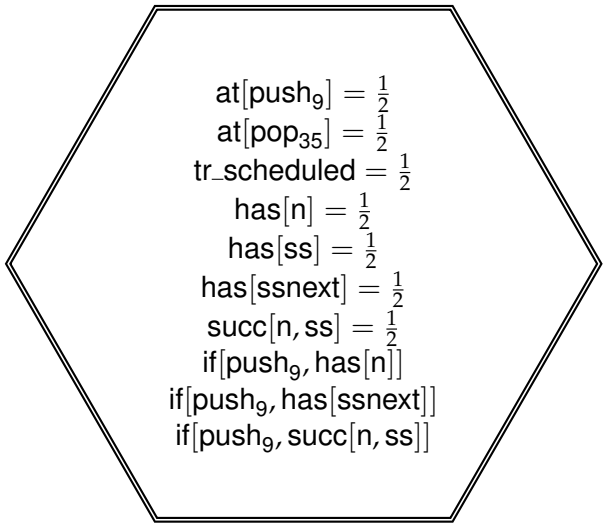
8.1.2 Selection of Predicates

I defined soft invariant predicates for a property consisting of a unary predicate, the negation of a unary predicate, or the implication between two of the same. They could also be defined for more complicated formulas, but it is almost certainly simpler and easier to use the formula as the definition of a new instrumentation predicate, which in turn is used for the definition of the soft invariant.

Identifying the values that every predicate can take at each location, to determine which soft invariant predicates are required to refine the abstraction, can be achieved by careful examination of the model's transitions, coupled with some trial and error. Defining soft invariants is a safe practice — as shown above, if the property is not actually invariant then

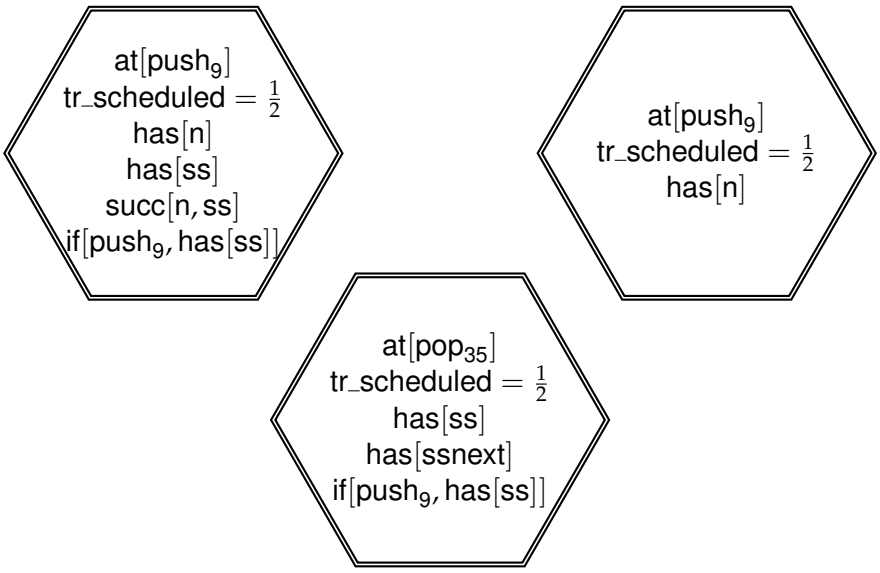


(a) Three previously canonical thread objects

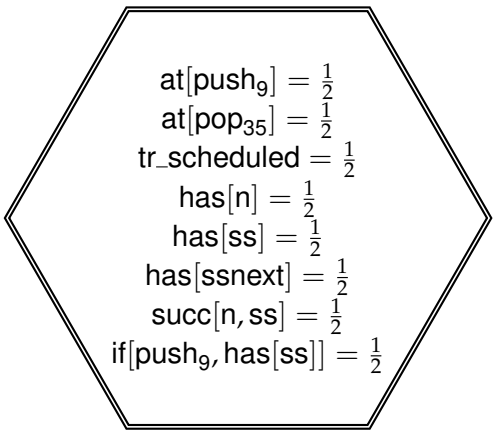


(b) Collapsed thread object

Figure 8.2: Collapsing threads: properties preserved with soft invariants



(a) Three previously canonical thread objects



(b) Collapsed thread object

Figure 8.3: Collapsing threads: soft invariant for non-invariant property

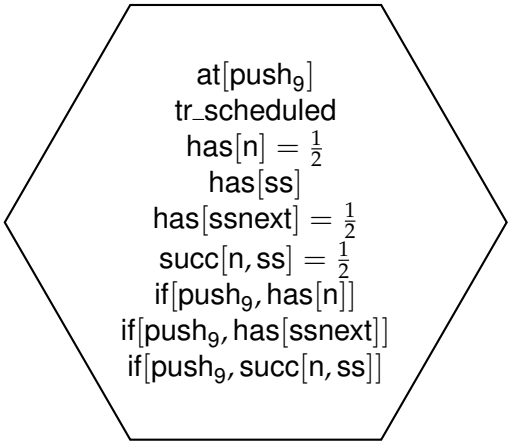


Figure 8.4: Concretised thread object to be sharpened

the predicate will become unknown during the analysis and not affect the correctness of the model. However, having an unnecessary instrumentation predicate will make the model less efficient, both by increasing the storage cost of each state and by adding extra constraints for Coerce to evaluate. Other reasons for soft invariant predicates being unnecessary and inefficient are when they record a property that is already being preserved in the abstraction by other predicates, and when the property being preserved is not needed for verification. These issues will be discussed further in later sections.

8.1.3 Compatibility Constraints

As was the case for the instrumentation predicates defined in Chapter 7 (see Section 7.6), the compatibility constraints generated for the soft invariant instrumentation predicates are not comprehensive enough for TVLA to sharpen (or discard) states sufficiently. Consider the thread object in Figure 8.4. We can logically deduce that all of the unknown predicates can have only a specific definite value, but the compatibility constraints generated by TVLA cannot deduce this, as their bodies all evaluate to unknown:

$$\begin{aligned}
&\mathsf{is_thread}(v) \wedge (\mathsf{at}[\mathsf{push}_9](v) \rightarrow \mathsf{has}[\mathsf{n}](v)) \\
&\quad \rhd \mathsf{if}[\mathsf{push}_9, \mathsf{has}[\mathsf{n}]](v) \\
&\neg\, (\mathsf{is_thread}(v) \wedge (\mathsf{at}[\mathsf{push}_9](v) \rightarrow \mathsf{has}[\mathsf{n}](v))) \\
&\quad \rhd \neg\, \mathsf{if}[\mathsf{push}_9, \mathsf{has}[\mathsf{n}]](v)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{is_thread}(v) \wedge (\mathsf{at}[\mathsf{push}_9](v) \rightarrow \neg\, \mathsf{has}[\mathsf{ssnext}](v)) \\
&\quad \rhd\ \mathsf{if}[\mathsf{push}_9, \mathsf{has}[\mathsf{ssnext}]](v) \\
&\neg\, (\mathsf{is_thread}(v) \wedge (\mathsf{at}[\mathsf{push}_9](v) \rightarrow \neg\, \mathsf{has}[\mathsf{ssnext}](v))) \\
&\quad \rhd\ \neg\, \mathsf{if}[\mathsf{push}_9, \mathsf{has}[\mathsf{ssnext}]](v) \\
&\mathsf{is_thread}(v) \wedge (\mathsf{at}[\mathsf{push}_9](v) \rightarrow (\mathsf{has}[\mathsf{ss}](v) \rightarrow \mathsf{succ}[\mathsf{n}, \mathsf{ss}](v))) \\
&\quad \rhd\ \mathsf{if}[\mathsf{push}_9, \mathsf{succ}[\mathsf{n}, \mathsf{ss}]](v) \\
&\neg\, (\mathsf{is_thread}(v) \wedge (\mathsf{at}[\mathsf{push}_9](v) \rightarrow (\mathsf{has}[\mathsf{ss}](v) \rightarrow \mathsf{succ}[\mathsf{n}, \mathsf{ss}](v)))) \\
&\quad \rhd\ \neg\, \mathsf{if}[\mathsf{push}_9, \mathsf{succ}[\mathsf{n}, \mathsf{ss}]](v)
\end{aligned}$$

I defined compatibility constraints of the following forms:

$$\begin{aligned}
&\mathsf{if}[\mathsf{push}_9, \mathsf{has}[\mathsf{n}]](v) \wedge \mathsf{at}[\mathsf{push}_9](v) \rhd \mathsf{has}[\mathsf{n}](v) \\
&\mathsf{if}[\mathsf{push}_9, \mathsf{has}[\mathsf{ssnext}]](v) \wedge \mathsf{at}[\mathsf{push}_9](v) \rhd \neg\, \mathsf{has}[\mathsf{ssnext}](v) \\
&\mathsf{if}[\mathsf{push}_9, \mathsf{succ}[\mathsf{n}, \mathsf{ss}]](v) \wedge \mathsf{at}[\mathsf{push}_9](v) \wedge \mathsf{has}[\mathsf{ss}](v) \rhd \mathsf{succ}[\mathsf{n}, \mathsf{ss}](v)
\end{aligned}$$

For each of these, the body evaluates to true, which allows the unknown predicates to be coerced to true, false and true, respectively.

8.2 Stack Models

In this section, I apply the approach of collapsing thread objects and defining soft invariants to the stack models from Chapter 7. Section 8.2.1 discusses the changes that need to be made to the model before soft invariants are introduced. Section 8.2.2 outlines the soft invariants I chose to define for the model with restricted interleaving. Section 8.2.3 presents the results of analysing the model. Section 8.2.4 presents the soft invariants and results for the model with full interleaving. Section 8.2.5 adds additional soft invariants to the latter model to investigate the tradeoff between manual precision and automatic efficiency.

8.2.1 Changes to the Model

Collapse

The first change to the stack model constructed in Sections 7.1–7.7 is to remove all thread predicates — including every location predicate — from the set of abstraction predicates $\mathcal{A}$, except is_thread and tr_scheduled. As a consequence of removing the location predicates from $\mathcal{A}$, we need to

define integrity rules to ensure that they are mutually exclusive, i.e. for all locations loc_1 and loc_2, where $loc_1 \neq loc_2$:

$$\mathsf{at}[\mathsf{loc}_1](v) \rightarrow \neg\, \mathsf{at}[\mathsf{loc}_2](v)$$

These constraints could have been defined in the original models in Chapter 7, but I chose not to as they were not necessary for verification — the model did not contain any states that violated them — and it would have made the model less efficient.

Null Equivalence

Two extra instrumentation predicates are required to record properties not preserved by the abstraction after the threads have been collapsed. Implicitly at the CAS update steps at push_9 and pop_{38}, the algorithm assumes that one field (ss and ssnext, respectively) is the next-successor of another (n and ss, respectively). When both fields are non-null, this is explicitly recorded by the instrumentation predicates $\mathsf{succ}[\mathsf{n}, \mathsf{ss}]$ and $\mathsf{succ}[\mathsf{ss}, \mathsf{ssnext}]$. But when one of the fields is null the property is recorded in the state implicitly.

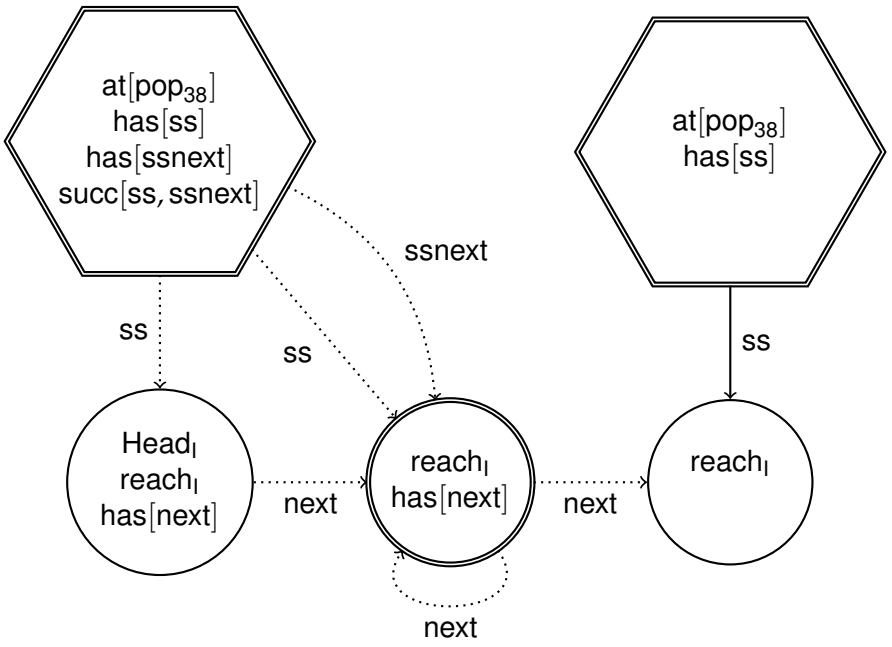
Consider Figure 8.5a, which shows two summary thread objects at location pop_{38}. In the left-hand thread object, with $\mathsf{has}[\mathsf{ssnext}]$ true, the predicate $\mathsf{succ}[\mathsf{ss}, \mathsf{ssnext}]$ records that $\mathsf{ssnext} = \mathsf{ss.next}$. In the right-hand thread object, with $\mathsf{has}[\mathsf{ssnext}]$ false, the distinction of the thread objects and the values of the ss predicate record that $\mathsf{ssnext} = \mathsf{ss.next}$ (i.e. null). Indeed, in every state of the model, a thread at pop_{38} with $\mathsf{has}[\mathsf{ssnext}]$ false has an ss predicate that points to a node with $\mathsf{has}[\mathsf{next}]$ false.

In Figure 8.5b, the thread objects are collapsed, and both of these properties are lost. As we have seen, the properties can be preserved by defining soft invariants. The properties we wish to use are:

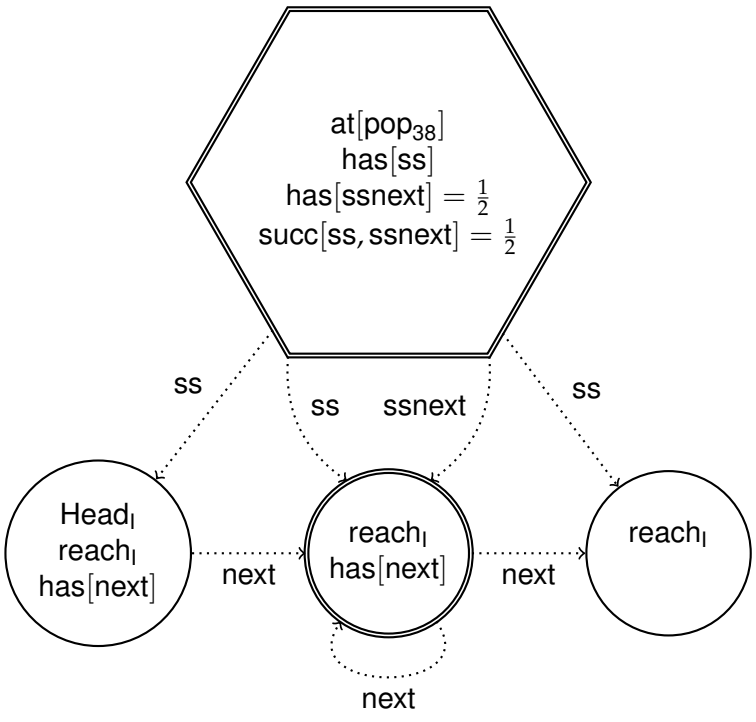
$$\mathsf{has}[\mathsf{ssnext}](t) \rightarrow \mathsf{succ}[\mathsf{ss}, \mathsf{ssnext}](t)$$
$$\neg\, \mathsf{has}[\mathsf{ssnext}](t) \rightarrow \exists\, n \bullet \mathsf{ss}(t, n) \wedge \neg\, \mathsf{has}[\mathsf{next}](n)$$

To make the soft invariant instrumentation predicates simpler and more efficient, I used the latter property to define a new instrumentation predicate for the model:

$$\mathsf{tonull}[\mathsf{ss}](v) \Leftrightarrow \exists\, u \bullet \mathsf{ss}(v, u) \wedge \neg\, \mathsf{has}[\mathsf{next}](u)$$



(a) Before collapse



(b) After collapse

Figure 8.5: Property of null equivalence is lost when threads are collapsed

For the same reasons at push_9, I also defined an equivalent instrumentation predicate:

$$\mathsf{tonull}[\mathsf{n}](v) \Leftrightarrow \exists u \bullet \mathsf{n}(v,u) \wedge \neg\, \mathsf{has}[\mathsf{next}](u)$$

Both of the new instrumentation predicates require additional compatibility constraints:

$$\exists t \bullet \mathsf{tonull}[\mathsf{ss}](t) \wedge \mathsf{ss}(t,n) \rhd \neg\, \mathsf{has}[\mathsf{next}](n)$$
$$\exists t \bullet \mathsf{tonull}[\mathsf{n}](t) \wedge \mathsf{n}(t,n) \rhd \neg\, \mathsf{has}[\mathsf{next}](n)$$

8.2.2 Soft Invariants

Interleaved Locations

To begin selecting soft invariants for the stack model, let us initially consider only locations outside of atomic blocks. There can be at most one thread in a state that is at a location within an atomic block, and if present it is scheduled, so is not collapsed due to the tr_scheduled predicate. Thus, any single predicate invariant property will be preserved implicitly, as was the case for all threads without the "collapse" approach.

Recall that Table 7.2 displays the possible values of instrumentation predicates for all such thread objects, obtained from the set of states output by TVLA. This information is reformulated in Table 8.1, with inferred data for the new tonull[n] and tonull[ss] predicates, to show the invariant properties. For each predicate at each state, the table records:

- 1 if it is always true;
- 0 if it is always false;
- p if it is always true when predicate p is true;
- ¬ p if it is always true when predicate p is false; and
- $\frac{1}{2}$ otherwise, i.e. it is sometimes true and sometimes false.

The highlighted entries are the ones I determined were necessary to record and preserve in the abstraction. For example, if doneLP can be sharpened to true at push_9 or pop_{38}, then the specification step after the CAS will give an error; if doneLP can be sharpened to true at pop_{35}, then it can be true

	idle	push_9	pop_{35}	pop_{38}
doneLP	0	0	0	0
has[lv]	0	(1)	$\frac{1}{2}$	(1)
has[n]	0	(1)	0	0
opval[n]	0	1	0	0
tonull[n]	0	$\neg$ has[ss]	0	0
succ[n, ss]	0	has[ss]	0	0
has[ss]	0	$\frac{1}{2}$	1	(1)
opval[ss]	0	$\frac{1}{2}$	$\frac{1}{2}$	1
has[ssnext]	0	0	$\frac{1}{2}$	$\frac{1}{2}$
tonull[ss]	0	$\frac{1}{2}$	$\frac{1}{2}$	$\neg$ has[ssnext]
succ[ss, ssnext]	0	0	$\frac{1}{2}$	has[ssnext]

Table 8.1: Invariant properties of stack model with restricted interleaving

at pop_{38}. Thus, I defined the soft invariant instrumentation predicates in Figure 8.6.

The entries of Table 8.1 in parentheses are properties that need to be preserved in the abstraction, but do not need to be explicitly recorded because they are implied by other properties. For example, has[lv] and has[n] are logical consequences of opval[n], so the definition of the soft invariant if[push_9, opval[n]] means that additionally defining if[push_9, has[lv]] and if[push_9, has[n]] is unnecessary.

The remaining entries are properties that do not need to be preserved by the abstraction. For example, has[ssnext] is false throughout a push operation, but this is because the operation does not assign or use the ssnext field. Thus, even if it was concretised to true, it would not affect the correctness of the model. Similarly, since beginPush and beginPop both initialise all fields used in the operation, there is no need to record that every predicate for an idle thread is false.

One important exception is for the field n. This field is only used in the push operation, so we might expect that it is unnecessary to record that has[n] is false at idle and during a pop operation. However, recall from Section 7.2.3 that the instrumentation predicate waiting is defined using n:

$$\text{waiting}(v) \Leftrightarrow \exists\, t \bullet \text{n}(t, v) \wedge \neg\, \text{doneLP}(t)$$

Thus, whenever doneLP is updated for a thread, the value of waiting must

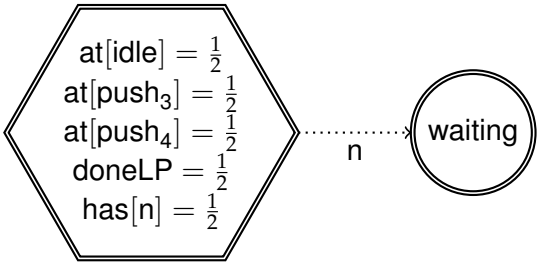
$\mathsf{if}[\mathsf{idle}, \mathsf{has}[\mathsf{n}]](v) \Leftrightarrow \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{idle}](v) \rightarrow \neg\, \mathsf{has}[\mathsf{n}](v))$
$\mathsf{if}[\mathsf{push}_9, \mathsf{doneLP}](v) \Leftrightarrow \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{push}_9](v) \rightarrow \neg\, \mathsf{doneLP}(v))$
$\mathsf{if}[\mathsf{push}_9, \mathsf{opval}[\mathsf{n}]](v) \Leftrightarrow \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{push}_9](v) \rightarrow \mathsf{opval}[\mathsf{n}](v))$
$\mathsf{if}[\mathsf{push}_9, \mathsf{succ}[\mathsf{n}, \mathsf{ss}]](v) \Leftrightarrow$
$\quad \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{push}_9](v) \rightarrow (\mathsf{has}[\mathsf{ss}](v) \rightarrow \mathsf{succ}[\mathsf{n}, \mathsf{ss}](v)))$
$\mathsf{if}[\mathsf{push}_9, \mathsf{tonull}[\mathsf{n}]](v) \Leftrightarrow$
$\quad \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{push}_9](v) \rightarrow (\neg\, \mathsf{has}[\mathsf{ss}](v) \rightarrow \mathsf{tonull}[\mathsf{n}](v)))$
$\mathsf{if}[\mathsf{pop}_{35}, \mathsf{doneLP}](v) \Leftrightarrow \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{pop}_{35}](v) \rightarrow \neg\, \mathsf{doneLP}(v))$
$\mathsf{if}[\mathsf{pop}_{35}, \mathsf{has}[\mathsf{n}]](v) \Leftrightarrow \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{pop}_{35}](v) \rightarrow \neg\, \mathsf{has}[\mathsf{n}](v))$
$\mathsf{if}[\mathsf{pop}_{35}, \mathsf{has}[\mathsf{ss}]](v) \Leftrightarrow \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{pop}_{35}](v) \rightarrow \mathsf{has}[\mathsf{ss}](v))$
$\mathsf{if}[\mathsf{pop}_{36}, \mathsf{succ}[\mathsf{ss}, \mathsf{ssnext}]](v) \Leftrightarrow$
$\quad \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{pop}_{36}](v) \rightarrow (\mathsf{has}[\mathsf{ssnext}](v) \rightarrow \mathsf{succ}[\mathsf{ss}, \mathsf{ssnext}](v)))$
$\mathsf{if}[\mathsf{pop}_{36}, \mathsf{tonull}[\mathsf{ss}]](v) \Leftrightarrow$
$\quad \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{pop}_{36}](v) \rightarrow (\neg\, \mathsf{has}[\mathsf{ssnext}](v) \rightarrow \mathsf{tonull}[\mathsf{ss}](v)))$
$\mathsf{if}[\mathsf{pop}_{38}, \mathsf{doneLP}](v) \Leftrightarrow \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{pop}_{38}](v) \rightarrow \neg\, \mathsf{doneLP}(v))$
$\mathsf{if}[\mathsf{pop}_{38}, \mathsf{has}[\mathsf{n}]](v) \Leftrightarrow \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{pop}_{38}](v) \rightarrow \neg\, \mathsf{has}[\mathsf{n}](v))$
$\mathsf{if}[\mathsf{pop}_{38}, \mathsf{opval}[\mathsf{ss}]](v) \Leftrightarrow \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{pop}_{38}](v) \rightarrow \mathsf{opval}[\mathsf{ss}](v))$
$\mathsf{if}[\mathsf{pop}_{38}, \mathsf{succ}[\mathsf{ss}, \mathsf{ssnext}]](v) \Leftrightarrow$
$\quad \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{pop}_{38}](v) \rightarrow (\mathsf{has}[\mathsf{ssnext}](v) \rightarrow \mathsf{succ}[\mathsf{ss}, \mathsf{ssnext}](v)))$
$\mathsf{if}[\mathsf{pop}_{38}, \mathsf{tonull}[\mathsf{ss}]](v) \Leftrightarrow$
$\quad \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{pop}_{38}](v) \rightarrow (\neg\, \mathsf{has}[\mathsf{ssnext}](v) \rightarrow \mathsf{tonull}[\mathsf{ss}](v)))$

Figure 8.6: Soft invariant instrumentation predicates for interleaving locations

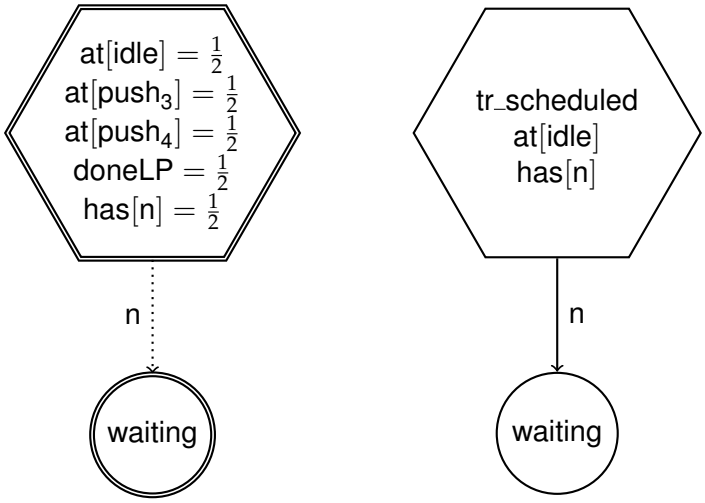
be updated for all nodes that the thread's n field points to. If $\mathsf{has}[\mathsf{n}]$ is not preserved as false by a soft invariant, then it is possible to concretise unreachable states when pop and idle threads have n-nodes, which in turn will orphan these nodes when n is set to null. For example, in Figure 8.7 we see a collapsed thread object representing threads at locations idle, push_3 and push_4 (8.7a). If one of the idle threads begins a push operation then one outcome of performing Focus with the formula

$$\mathsf{tr\_scheduled}(v_1) \wedge \mathsf{at}[\mathsf{idle}](v_1) \wedge \mathsf{n}(v_1, v_2)$$

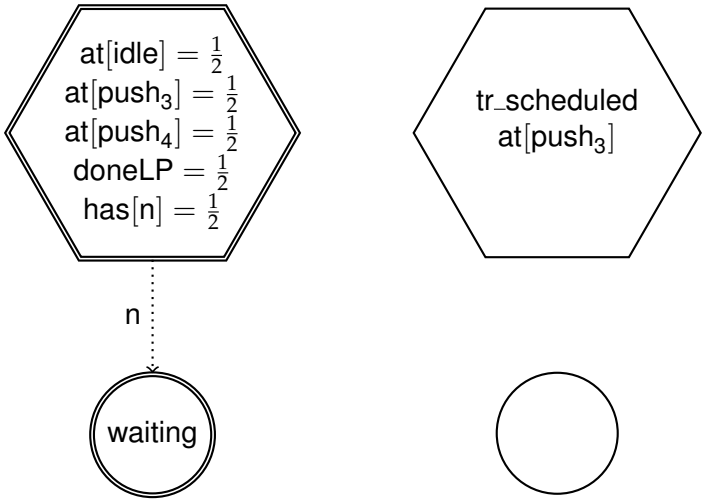
is that the concretised idle thread has an n-node (8.7b). Then the update is applied, which sets n to null and the node is orphaned (8.7c). To avoid these unreachable states, I defined soft invariants to preserve the value of $\mathsf{has}[\mathsf{n}]$ at each of the interleaving locations.



(a) Abstract state



(b) After Focus and Coerce



(c) After Update

Figure 8.7: Concretising an unreachable state

Non-interleaved Locations

As mentioned above, the invariant property of a single predicate does not need to be explicitly recorded at a location that occurs within an atomic block, as the property will be recorded by the distinguished (scheduled) thread object. However, for conditional invariant properties, the property is lost when states are merged together in a partially isomorphic analysis.

I defined two pairs of conditional soft invariants for the interleaving locations: recording $\mathsf{n.next} = \mathsf{ss}$ at push_9 and $\mathsf{ss.next} = \mathsf{ssnext}$ at pop_{38}. The properties at push_9 do not need to be preserved at any other location — they are only set during the transition from push_7 to push_9, and after the CAS step (whether successful or not) the property is not assumed again.

In contrast, the properties at pop_{38} are set during the transition from pop_{35} to pop_{36}, which is within an atomic block. Thus I needed to define the same soft invariants for pop_{36} too:

$$\mathsf{if}[\mathrm{pop}_{36}, \mathsf{succ}[\mathsf{ss}, \mathsf{ssnext}]](v) \Leftrightarrow \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathrm{pop}_{36}](v) \rightarrow (\mathsf{has}[\mathsf{ssnext}](v) \rightarrow \mathsf{succ}[\mathsf{ss}, \mathsf{ssnext}](v)))$$
$$\mathsf{if}[\mathrm{pop}_{36}, \mathsf{tonull}[\mathsf{ss}]](v) \Leftrightarrow \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathrm{pop}_{36}](v) \rightarrow (\neg\, \mathsf{has}[\mathsf{ssnext}](v) \rightarrow \mathsf{tonull}[\mathsf{ss}](v)))$$

8.2.3 Results

I analysed the modified stack model using the same hardware and software as in Chapter 7, successfully verifying linearisability for unbounded threads, data values and list lengths. Table 8.2 shows the results, with comparisons against the equivalent figures in Table 7.7. Table 8.2b shows that collapsing the thread objects results in a large reduction of the statespace, with 99.4% reduction in states stored. Table 8.2a shows that the approach is far more practical, taking 25 seconds and less than 300 MB of RAM, rather than over half an hour and more than 1 GB of RAM.

Overall, the analysis is less efficient, in terms of the seconds per state and MB per state averages, but this is expected as there are a greater proportion of the states merged by the partially isomorphic analysis.

8.2.4 Full Interleaving

I modified the stack model with full interleaving, from Section 7.8.3, to collapse the thread objects. Previously, the model was exponentially larger

Th.	Heap Limit (MB)	Time (s)	%	Ave RAM (MB)	%	Max RAM (MB)	%
∞	800	25	1.3	115	17.8	280	26.5
∞	2,048	24	1.2	190	29.3	663	62.7

(a) Resources

Th.	Unsch. States (uns.)	%	Stored States (stored)	%	Total States (total)	%	Breaches	%
∞	16	0.8	447	0.6	2,526	0.9	27,256	1.6

(b) Statespace

Table 8.2: Stack verification results for restricted interleaving

than when interleaving was restricted and the analysis ran out of memory. When the threads are collapsed, the model can be verified with a relatively small memory increase.

Soft Invariants

Table 8.3 extends Table 8.1 to include the additional locations that are now interleaved, i.e. everything except for $push_{11}$, pop_{25}, pop_{26} and pop_{40}. The unbounded model was too large to analyse previously (see Section 7.8.3), so I was unable to rely on output from TVLA to determine these properties. Instead, I examined the transitions of the model, inferring predicate values in conjunction with those in Table 8.3.

As before, the highlighted values are the ones I determined to be essential for verification, and required soft invariant instrumentation predicates. For $push_3$–$push_7$, pop_{24} and pop_{36} the properties are the same as for $push_9$ and pop_{38}, from when they are first set. For $push_{18}$ and pop_{48}, only the values of doneLP and has[n] need to be preserved: the former to avoid errors in endPush and endPop, the latter to preserve the value of the predicate in pop locations as discussed above.

Results

Table 8.4 shows the results of verifying linearisability with full interleaving and collapsed threads, with the figures compared against the results in

	idle	push3	push4	push6	push7	push9	push18	pop24	pop35	pop36	pop38	pop48
doneLP	0	0	0	0	0	0	1	0	0	0	0	1
has⌈lv⌉	0	1	1	(1)	(1)	(1)	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	(1)	$\frac{1}{2}$
has⌈n⌉	0	0	1	(1)	(1)	(1)	1	0	0	0	0	0
opval⌈n⌉	0	0	0	1	1	1	1	0	0	0	0	0
tonull⌈n⌉	0	0	1	$\frac{1}{2}$	$\frac{1}{2}$	¬ has⌈ss⌉	¬ has⌈ss⌉	0	0	0	0	0
succ⌈n, ss⌉	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	has⌈ss⌉	has⌈ss⌉	0	0	0	0	0
has⌈ss⌉	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1	(1)	(1)	$\frac{1}{2}$
opval⌈ss⌉	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1	$\frac{1}{2}$
has⌈ssnext⌉	0	0	0	0	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
tonull⌈ss⌉	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	¬ has⌈ssnext⌉	¬ has⌈ssnext⌉	$\frac{1}{2}$
succ⌈ss, ssnext⌉	0	0	0	0	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	has⌈ssnext⌉	has⌈ssnext⌉	$\frac{1}{2}$

Table 8.3: Invariant properties of stack with full interleaving

Th.	Heap Limit (MB)	Time (s)	%	Ave RAM (MB)	%	Max RAM (MB)	%
∞	800	141	572	149	130	295	105
∞	2,048	142	594	255	134	685	103

(a) Resources

Th.	Unsch. States (uns.)	%	Stored States (stored)	%	Total States (total)	%	Breaches	%
∞	32	200	932	209	5,596	222	264,375	970

(b) Statespace

Table 8.4: Stack verification results for full interleaving

Table 8.2.

The model stores more than twice the number of states compared to when interleaving is restricted. This is a great improvement over the results from Table 7.9, where the increase was nearly six times with two threads and nearly 23 times with three threads.

The amount of memory used by TVLA only increases slightly, but the analysis takes nearly six times as long. However, the analysis is still 13 times faster than the model *with* restricted interleaving and *without* collapsed threads.

8.2.5 Extra Predicates

Collapsing thread objects provides an exponential reduction in the size of the statespace, as evidenced by the results in Tables 8.2 and 8.4 (and below in Tables 8.7 and 8.8). One drawback is that defining soft invariant predicates to refine the abstraction sufficiently adds an extra amount of time-consuming and error-prone manual effort to the approach.

However, we can mitigate this somewhat by exploiting the "soft" property of the soft invariants. The predicates remain true in all thread objects and states only if the property in question is actually invariant, thus, a soft invariant can safely be defined even if we are not sure of its validity.

One consequence is that the selection of soft invariant predicates can begin more quickly with rough guesses of properties required, and be re-

fined as needed. If a required property is missing, then analysing the resulting error should indicate what to define additionally. If superfluous properties are defined — whether redundant or non-invariant — the analysis will still complete successfully.[1]

Extrapolating the idea of superfluous predicates, we could instead define a wide range of soft invariants without investigating which ones were actually correct and necessary. The extra predicates would make the analysis less efficient, but would reduce the amount of manual effort required.

To confirm and quantify this idea, I modified the stack model with full interleaving from Section 8.2.4 and defined two soft invariants for every thread predicate p at every location loc_X:

$$\mathsf{if}[\mathsf{loc_X}, \mathsf{p}](v) \Leftrightarrow \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{loc_X}](v) \rightarrow \mathsf{p}(v))$$
$$\mathsf{nif}[\mathsf{loc_X}, \mathsf{p}](v) \Leftrightarrow \mathsf{is\_thread}(v) \wedge (\mathsf{at}[\mathsf{loc_X}](v) \rightarrow \neg\, \mathsf{p}(v))$$

I kept the six conditional soft invariants, prefixing them with a 'c':

$$\mathsf{cif}[\mathsf{push}_9, \mathsf{tonull}[\mathsf{n}]],\ \mathsf{cif}[\mathsf{push}_9, \mathsf{succ}[\mathsf{n}, \mathsf{ss}]],$$
$$\mathsf{cif}[\mathsf{pop}_{36}, \mathsf{tonull}[\mathsf{ss}]],\ \mathsf{cif}[\mathsf{pop}_{36}, \mathsf{succ}[\mathsf{ss}, \mathsf{ssnext}]],$$
$$\mathsf{cif}[\mathsf{pop}_{38}, \mathsf{tonull}[\mathsf{ss}]],\ \mathsf{cif}[\mathsf{pop}_{38}, \mathsf{succ}[\mathsf{ss}, \mathsf{ssnext}]]$$

Table 8.5 shows the results of the analysis, with comparisons against the figures in Table 8.4. The number of states stored is actually reduced by 22%, but the reduction is only in the states that have been focussed to have a single non-summary thread object. The number of constraint breaches reduces by 41%, but the number of constraints to be considered by each application of Coerce more than doubles, from 453 to 1,164. As a consequence, the amount of RAM used increases slightly and the time taken nearly doubles.

8.3 Queue Models

In this section, I apply the approach of collapsing thread objects and defining soft invariants to the queue models from Section 7.9. Section 8.3.1 introduces an additional instrumentation predicate that is needed for the models when threads are collapsed. Section 8.3.2 outlines the soft invariants I chose to define. Section 8.3.3 presents the results of analysing both queue algorithms, with and without restricting interleaving.

[1] Assuming the property is correct.

Th.	Heap Limit (MB)	Time (s)	%	Ave RAM (MB)	%	Max RAM (MB)	%
∞	800	278	197	202	136	338	115
∞	2,048	264	186	375	147	764	111

(a) Resources

Th.	Unsch. States (uns.)	%	Stored States (stored)	%	Total States (total)	%	Breaches	%
∞	32	100	726	78	5,850	104	155,518	59

(b) Statespace

Table 8.5: Stack verification results for full interleaving with extra predicates

8.3.1 Changes to the model

Full interleaving

In Chapter 8, I only analysed models of the queue algorithm with restricted interleaving. In this chapter, I have additionally analysed models with full interleaving, as shown in Figure 8.8. As with the stack models, only the transitions at linearisation points are left in atomic blocks.

Ordered snapshots

As with the stack model, an extra instrumentation predicate is required to preserve a property that is lost when the thread objects are collapsed, but only for the model with the original dequeue operation (see below). However, the predicate $\mathsf{tonull}[\mathsf{p}]$ is *not* required for the queue models. This was needed for the stack models because a CAS could be performed when the successor value was null; the queues use a dummy node, so the CAS steps are only performed when the successor is non-null.

Example Consider the partial state with collapsed threads shown in Figure 8.9a. One enqueuing thread has appended a node to the end of the list but not yet advanced Tail_l, which is lagging. Other dequeuing threads are about to attempt the CAS step at deq_{50} to advance Head_l. At this location we assume that sshead and sstail differ, given the test at deq_{41}, and that

idle	beginEnqueue()	enq_3
enq_3	newNode(n)	enq_4
enq_4	assign($n.val, lv$)	enq_6
enq_6	assign($sstail, Tail$)	enq_7
enq_7	assign($ssnext, sstail.next$)	enq_8
enq_8	isEqual($sstail, Tail$)	enq_9
enq_8	isNotEqual($sstail, Tail$)	enq_6
enq_9	isNull($ssnext$)	enq_{11}
enq_9	isNotNull($ssnext$)	enq_{20}
enq_{20}	CASfail($Tail, sstail$)	enq_6
enq_{20}	CASsucc($Tail, sstail, ssnext$)	enq_6
enq_{11}	CASfail($sstail.next, ssnext$)	enq_6
atomic{		
enq_{11}	CASsucc($sstail.next, ssnext, n$)	enq_{13}
enq_{13}	specEnqueue()	enq_{24}
}		
enq_{24}	CASfail($Tail, sstail$)	enq_{25}
enq_{24}	CASsucc($Tail, sstail, n$)	enq_{25}
enq_{25}	endEnqueue()	idle

(a) Enqueue operation

Figure 8.8: Transitions of queue models

idle	beginDequeue()	deq_{30}
deq_{30}	assign($sshead, Head$)	deq
deq_{31}	assign($sstail, Tail$)	deq_{33}
atomic{		
deq_{33}	assign($ssnext, sshead.next$)	deq_{34}
deq_{34}	isNull($ssnext$)	deq_{35}
deq_{34}	isNotNull($ssnext$)	deq_{40}
deq_{35}	specDequeueEmpty()	deq_{40}
}		
deq_{40}	isEqual($sshead, Head$)	deq_{41}
deq_{40}	isNotEqual($sshead, Head$)	deq_{30}
deq_{41}	isEqual($sshead, sstail$)	deq_{42}
deq_{41}	isNotEqual($sshead, sstail$)	deq_{48}
deq_{42}	isNull($ssnext$)	deq_{43}
deq_{42}	isNotNull($ssnext$)	deq_{46}
deq_{43}	endDequeueEmpty()	idle
deq_{46}	CASfail($Tail, sstail$)	deq_{30}
deq_{46}	CASsucc($Tail, sstail, ssnext$)	deq_{30}
deq_{48}	assign($lv, ssnext.val$)	deq_{50}
deq_{50}	CASfail($Head, sshead$)	deq_{30}
atomic{		
deq_{50}	CASsucc($Head, sshead, ssnext$)	deq_{52}
deq_{52}	specDequeue()	deq_{61}
}		
deq_{61}	endDequeue()	idle

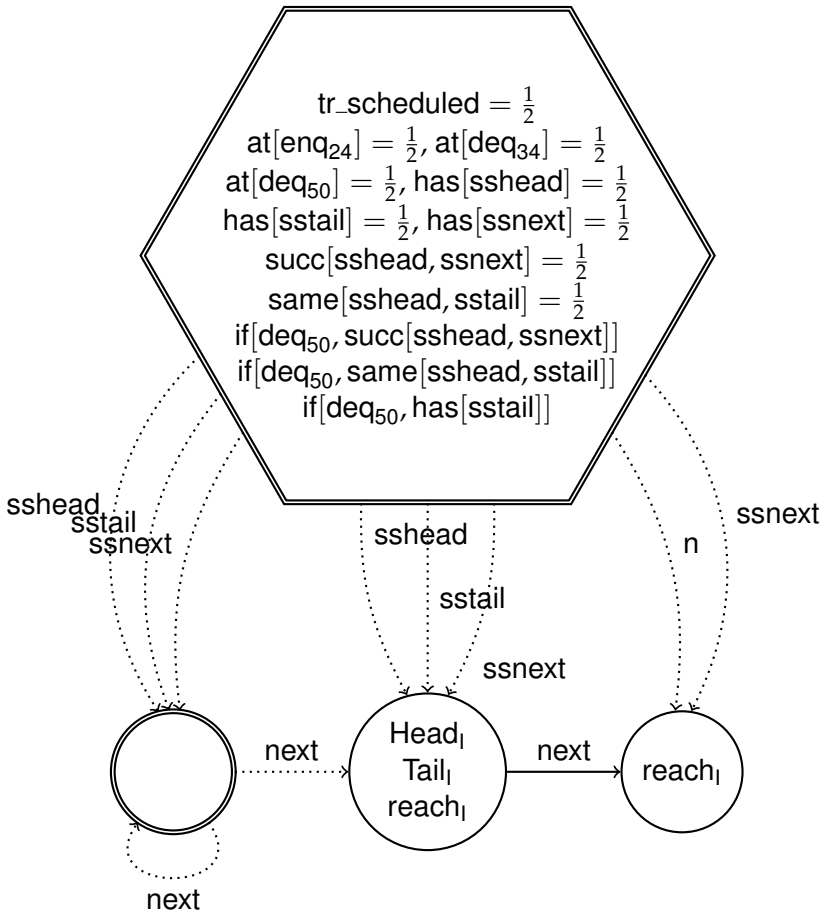
(b) Original Dequeue operation

Figure 8.8: Transitions of queue models

idle	beginDequeue()	deq_{67}
deq_{67}	assign($sshead, Head$)	deq_{69}
atomic{		
deq_{69}	assign($ssnext, sshead.next$)	deq_{70}
deq_{70}	isNull($ssnext$)	deq_{71}
deq_{70}	isNotNull($ssnext$)	deq_{76}
deq_{71}	specDequeueEmpty()	deq_{76}
}		
deq_{76}	isEqual($sshead, Head$)	deq_{77}
deq_{76}	isNotEqual($sshead, Head$)	deq_{67}
deq_{77}	isNull($ssnext$)	deq_{78}
deq_{77}	isNotNull($ssnext$)	deq_{81}
deq_{78}	endDequeueEmpty()	idle
deq_{81}	assign($lv, ssnext.val$)	deq_{83}
deq_{83}	CASfail($Head, sshead$)	deq_{67}
atomic{		
deq_{83}	CASsucc($Head, sshead, ssnext$)	deq_{85}
deq_{85}	specDequeue()	deq_{94}
}		
deq_{94}	assign($sstail, Tail$)	deq_{95}
deq_{95}	isEqual($sshead, sstail$)	deq_{96}
deq_{95}	isNotEqual($sshead, sstail$)	deq_{98}
deq_{96}	CASfail($Tail, sstail$)	deq_{98}
deq_{96}	CASsucc($Tail, sstail, ssnext$)	deq_{98}
deq_{98}	endDequeue()	idle

(c) Simplified Dequeue operation

Figure 8.8: Transitions of queue models



(a) Abstract state

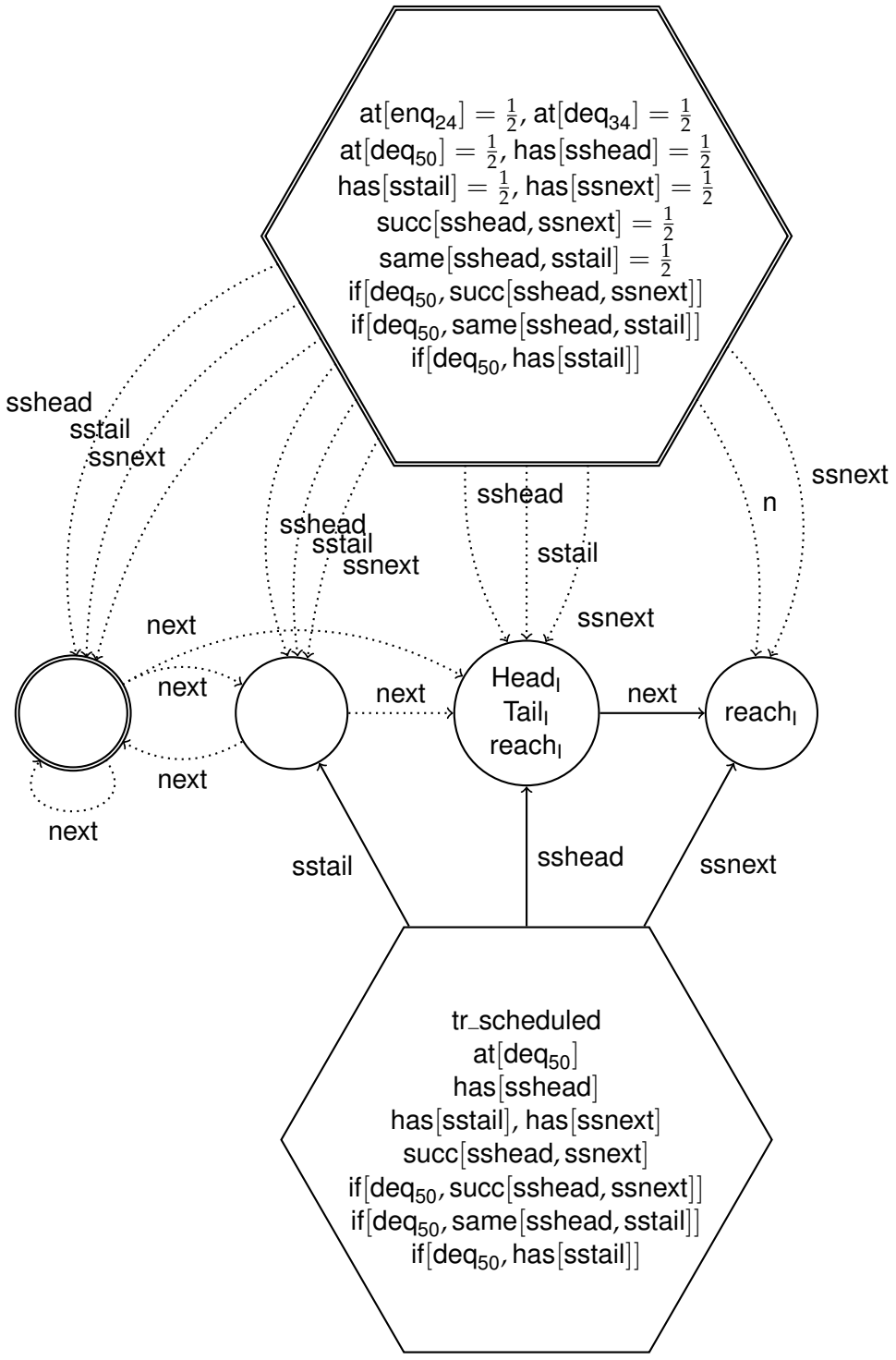
Figure 8.9: Snapshot order not preserved

ssnext is the next-successor of sshead; soft invariants are defined appropriately.

However, one possible outcome of focussing a scheduled dequeuing thread is shown in Figure 8.9b. Now sstail is pointing to a 'garbage' node and sshead is pointing to Head_l; thus the CAS will succeed and Head_l will "cross" Tail_l — not expected behaviour! ■

The property that is being lost in the above example is that sshead and sstail never "cross", alternatively that sstail is reachable from sshead:

$$\forall\, v_1, v_2 \bullet \mathsf{sshead}(t, v_1) \wedge \mathsf{sstail}(t, v_2) \rightarrow \mathsf{next}^*(v_1, v_2)$$



(b) After Focus and Coerce

Figure 8.9: Snapshot order not preserved

Such a property would be *difficult* to use for sharpening with Coerce. Since it is only of concern when both fields are in the list (i.e. not 'garbage') we can use a corollary — that if sshead is in the list then sstail is too. Thus, I defined the following new instrumentation predicate:

$$\begin{aligned}&\mathsf{ordered}[\mathsf{sshead},\mathsf{sstail}](v_1) \Leftrightarrow \exists\, v_2, v_3 \bullet \\ &\quad \mathsf{sshead}(v_1, v_2) \wedge \mathsf{sstail}(v_1, v_3) \wedge \\ &\quad (\mathsf{reach}_{\mathsf{l}}(v_2) \rightarrow \mathsf{reach}_{\mathsf{l}}(v_3))\end{aligned}$$

Additionally, I defined the following compatibility constraint:

$$\begin{aligned}&\exists\, v_1, v_2 \bullet \mathsf{ordered}[\mathsf{sshead},\mathsf{sstail}](v_1) \wedge \mathsf{sshead}(v_1, v_2) \wedge \\ &\quad \mathsf{sstail}(v_1, n_3) \wedge \mathsf{reach}_{\mathsf{l}}(v_2) \rhd \mathsf{reach}_{\mathsf{l}}(n_3)\end{aligned}$$

With ordered[sshead, sstail] defined, the state in Figure 8.9b would be discarded.

In the simplified dequeue operation (Figure 8.8c), sstail is not read until after the CAS step at deq_{83}, and $\mathsf{Head}_{\mathsf{l}}$ and $\mathsf{Tail}_{\mathsf{l}}$ are actually allowed to cross at this point (see Section 2.6.2). Thus, ordered[sshead, sstail] is not needed for these models.

8.3.2 Soft Invariants

Let us first consider the models with full interleaving. There is no output from TVLA to extract statespace information — the restricted model in Section 7.8.3 was too large to be fully analysed. Instead, I examined the transitions in Figure 8.8 and inferred the possible values of the predicates at each location, and determined which properties would be required for verification. The latter involved a small amount of trial and error, as one or two required properties were not included and had to be deduced from the error output of TVLA.

The properties I determined are shown in Table 8.6. As with the stack tables (8.1 and 8.3), the highlighted values are the required properties I defined soft invariant instrumentation predicates for, and the values in parentheses are the required properties that do not need to be preserved explicitly.

The queue models use the instrumentation predicate waiting, so as with the stack models I defined soft invariants for has[n] at idle and throughout the dequeue operations.

	init	enq3	enq4	enq6	enq7	enq8	enq9	enq11	enq20	enq24	enq25
doneLP	0	0	0	0	0	0	0	0	1	1	1
doneELP	0	0	0	0	0	0	0	0	0	0	0
has[lv]	0	1	1	(1)	(1)	(1)	(1)	(1)	1	1	1
has[n]	0	0	1	(1)	(1)	(1)	(1)	(1)	1	1	1
opval[n]	0	0	0	1	1	1	1	1	(1)	(1)	1
has[sshead]	0	0	0	0	0	0	0	0	0	0	0
has[sstail]	0	0	0	$\frac{1}{2}$	1	1	1	1	(1)	(1)	1
same[sshead, sstail]	0	0	0	0	0	0	0	0	0	0	0
has[ssnext]	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	(1)	0	0
succ[sstail, n]	0	0	0	0	0	0	0	0	0	1	1
succ[sshead, ssnext]	0	0	0	0	0	0	0	0	0	0	0
succ[sstail, ssnext]	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	ψ_1	ψ_1	0	1	0	0
opval[ssnext]	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
ordered[sshead, sstail]	0	0	0	0	0	0	0	0	0	0	0

$\psi_1 = \mathsf{has}[\mathsf{ssnext}]$

(a) Enqueue operation

Table 8.6: Invariant properties

	deq$_{30}$	deq$_{31}$	deq$_{33}$	deq$_{34}$	deq$_{40}$	deq$_{41}$	deq$_{42}$	deq$_{43}$	deq$_{48}$	deq$_{49}$	deq$_{50}$	deq$_{61}$
doneLP	0	0	0	(0)	0	0	0	0	0	0	0	1
doneELP	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2} \vee \neg\, \psi_1$	$\frac{1}{2} \vee \neg\, \psi_1$	$\frac{1}{2} \vee \neg\, \psi_1$	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
has[lv]	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	(1)	1	1
has[n]	0	0	0	(0)	0	0	0	0	0	0	0	0
opval[n]	0	0	0	0	0	0	0	0	0	0	0	0
has[sshead]	$\frac{1}{2}$	1	1	(1)	1	1	(1)	1	1	(1)	1	1
has[sstail]	$\frac{1}{2}$	$\frac{1}{2}$	1	(1)	1	1	(1)	1	1	1	1	1
same[sshead, sstail]	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1	1	1	0	0	0
has[ssnext]	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2} \vee \neg\, \psi_2$	$\frac{1}{2} \vee \neg\, \psi_2$	$\frac{1}{2} \vee \neg\, \psi_2$	$\frac{1}{2}$	0	(1)	(1)	(1)	1
succ[sstail, n]	0	0	0	0	0	0	0	0	0	0	0	0
succ[sshead, ssnext]	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	ψ_1	ψ_1	ψ_1	ψ_1	0	1	1	1	1
succ[sstail, ssnext]	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$(\psi_1 \wedge \psi_2)$	$(\psi_1 \wedge \psi_2)$	$(\psi_1 \wedge \psi_2)$	(ψ_1)	0	1	0	0	0
opval[ssnext]	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{1}{2}$	1	1
ordered[sshead, sstail]	$\frac{1}{2}$	$\frac{1}{2}$	1	(1)	1	1	1	1	1	1	1	1

$\psi_1 \;=\; \mathsf{has[ssnext]}$

$\psi_2 \;=\; \mathsf{same[sshead, sstail]}$

(b) Original Dequeue operation

Table 8.6: Invariant properties

	deq_{67}	deq_{69}	deq_{70}	deq_{76}	deq_{77}	deq_{78}	deq_{81}	deq_{83}	deq_{94}	deq_{95}	deq_{96}	deq_{98}
doneLP	0	0	(0)	0	0	0	0	1	1	1	1	1
doneELP	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2} \vee \neg\,\psi_1$	$\frac{1}{2} \vee \neg\,\psi_1$	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
has[lv]	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	(1)	1	1	1	1	1
has[n]	0	0	(0)	0	0	0	0	0	0	0	0	0
opval[n]	0	0	0	0	0	0	0	0	0	0	0	0
has[sshead]	$\frac{1}{2}$	1	(1)	1	1	1	(1)	(1)	(1)	1	1	1
has[sstail]	0	0	0	0	0	0	0	0	1	(1)	1	1
same[sshead, sstail]	0	0	0	0	0	0	0	0	$\frac{1}{2}$	1	$\frac{1}{2}$	$\frac{1}{2}$
has[ssnext]	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	(1)	(1)	(1)	(1)	1	1
succ[sstail, n]	0	0	0	0	0	0	0	0	0	0	0	0
succ[sshead, ssnext]	$\frac{1}{2}$	$\frac{1}{2}$	ψ_1	ψ_1	ψ_1	0	1	1	1	1	1	1
succ[sstail, ssnext]	0	0	0	0	0	0	0	0	(0)	1	ψ_2	
opval[ssnext]	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1	1	1	1	1

$\psi_1 \;=\; \text{has[ssnext]}$

$\psi_2 \;=\; \text{same[sshead, sstail]}$

(c) Simplified Dequeue operation

Table 8.6: Invariant properties

Enqueue

For the enqueue operation (Table 8.6a), the properties are the same for both algorithms, except that ordered[sshead, sstail] is not defined for the simplified one (see Section 8.3.1).

The required properties are similar to those for the stack: notably doneLP must be preserved throughout the operation, and the predicates only used in dequeues — doneELP and anything defined with sshead — can be ignored.

At enq_8 and enq_9 there are conditional properties, as succ[sstail, ssnext] is true when has[ssnext] is true; the transitions from enq_9 test whether ssnext is null, so succ[sstail, ssnext] is unconditionally true at enq_{11} and false at enq_{20}.

Simplified Dequeue

There are two notable features of Table 8.6c, recording properties of the simplified dequeue operation. The first is the inclusion of a location within an atomic block, deq_{70}, due to the presence of a conditional property. The second is the disjunction in the properties of doneELP at deq_{76} and deq_{77}.

By way of comparison, the conditional properties of succ[sshead, ssnext], and those for the enqueue operation and the stack models are actually biconditional. This is due to each antecedent being entailed by the consequent, e.g.

$$\forall\, v \bullet \mathsf{succ}[\mathsf{sshead}, \mathsf{ssnext}](v) \rightarrow \mathsf{has}[\mathsf{ssnext}](v)$$

There is no such relationship between doneELP and has[ssnext] however. At deq_{76}, if has[ssnext] is false then the thread's previous transition will instead have been specDequeueEmpty from deq_{71}, which sets doneELP to true. If has[ssnext] is true, the thread's previous transition will have been isNotNull from deq_{70} so doneELP may be false; however, doneELP may also be true after being set by specDequeueEmpty in a previous iteration of the loop. In both cases, we need only to preserve the conditional property.

Original Dequeue

Most of the predicate values and conditional properties for the original dequeue (Table 8.6b) are relatively straightforward, following from the assignments and comparisons of the transitions, but the value of has[ssnext]

between deq_{34} and deq_{42} is more subtle, relying on the structure of the list. When Head_l and Tail_l are equal, the list may be empty, in which case Head_l has no next-successor; alternatively the list may have size one and have Tail_l lagging, in which case Head_l has a next-successor. Whenever Head_l and Tail_l are not equal then the list is non-empty, and Head_l has a next-successor.

Thus, at deq_{34} after ssnext has been assigned, if same[sshead, sstail] is true then has[ssnext] could be either true or false, but if same[sshead, sstail] is false then has[ssnext] has to be true. The latter property is essential to preserve because after same[sshead, sstail] is checked to be false at deq_{41}, ssnext is assumed to be non-null (e.g. at deq_{48} and deq_{50}) but not explicitly tested for.

Restricted Interleaving

The models with restricted interleaving could use exactly the same soft invariant instrumentation predicates. However, the single-predicate properties of the locations within atomic blocks are preserved anyway, so defining soft invariants would serve only to make the model less efficient. Thus, I did not define soft invariants for enq_3–enq_6, enq_9, enq_{25}, deq_{30}, deq_{41}–deq_{43}, deq_{50}–deq_{61}, deq_{67}, deq_{77}, deq_{78}, deq_{83}, deq_{95}, and deq_{98}, except for the highlighted conditional properties.

8.3.3 Results

The models in Section 7.9 with restricted interleaving quickly ran out of memory if there were more than three threads. Modifying these models, by collapsing their thread objects, allowed me to verify linearisability for unbounded threads, data values and list lengths. The results are contained in Table 8.7, showing that both verifications require only 310 MB of RAM. The statespace of the model with the simplified dequeue operation (S) is slightly larger than for the model with the original dequeue operation (O); this is not unexpected, as it admits additional behaviour in allowing Head_l and Tail_l to "cross". The simplified model also has 2.8 times the number of constraint breaches, which contributes to the analysis taking 2.5 times as long as the original model — 3.2–3.6 minutes, compared to 1.3 minutes.

The models with full interleaving were also able to be verified comfortably within resource limits, needing only 730 MB of RAM. Table 8.8 shows the results, with comparisons against Table 8.7. The statespaces increase

Deq	Th.	Heap Limit (MB)	Time (s)	Ave RAM (MB)	Max RAM (MB)	Uns. States	Stored States	Total States	Breaches
O	∞	800	81	151	309	12	523	2,994	89,869
O	∞	2,048	79	296	717	12	523	3,001	89,869
S	∞	800	216	181	310	14	591	3,254	248,223
S	∞	2,048	194	363	699	14	591	3,255	248,223

Table 8.7: Queue verification results for restricted interleaving

Deq	Th.	Heap Limit (MB)	Time (s)	%	Ave RAM (MB)	%	Max RAM (MB)	%
O	∞	800	3,109	3,838	304	201	727	235
O	∞	2,048	2,810	3,557	581	196	1,691	236
S	∞	800	2,887	1,337	290	160	731	236
S	∞	2,048	2,658	1,370	515	142	1,537	220

(a) Resources

Deq	Th.	Unsch. States (uns.)	%	Stored States (stored)	%	Total States (total)	%	Breaches	%
O	∞	96	800	4,148	793	24,378	814	2,375,953	2,644
S	∞	112	800	4,604	779	27,422	843	2,315,936	933

(b) Statespace

Table 8.8: Queue verification results for full interleaving

by a factor of eight, with that of the simplified model again being slightly larger. The number of constraint breaches are roughly equal, and the analyses took 44–48 minutes for the simplified model and 47–52 minutes for the original model.

8.4 Conclusion

In this chapter, I have presented a novel approach to canonical abstraction, which exponentially reduces the statespace size by collapsing all thread objects into a single summary object. This has allowed me to verify linearisability of the stack and queue algorithms for unbounded threads, data values and list lengths whilst placing no restriction on the interleaving between threads. The analyses required only 310 MB and 750 MB of RAM, respectively, whereas the approach of Chapter 7 alone required much more than 2 GB.

The coarse abstraction on threads is refined by defining "soft invariant" instrumentation predicates, which preserve properties of the threads at specific locations. The identification of sufficient soft invariants is a manual process, requiring some insight and knowledge of the algorithm, which is a disadvantage by decreasing automation. However, the "soft invariants" are safe — correctness is not affected if the properties are not actually invariant (see Section 8.2.5) — so they can be "over defined" to reduce the manual effort.

The quantified canonical abstraction approach of Berdine et al. [2008] is also effective at reducing the statespace, compared to the models in Chapter 7. However, the reduction is caused by decomposing states into subgraphs and only storing one copy of subgraphs that are common to more than one state — the number of canonical thread objects is still a factor in the size of the statespace. My approach removes the thread objects as a factor in the statespace size, so I expect that the size of the statespace will scale better for larger algorithms.

This approach is not limited to linked list based data structures, and could conceivably be applied to many other types of models.

Part IV

Conclusion

Chapter 9

Conclusion

In this thesis I have presented an approach for representing nonblocking data structure algorithms and their properties, and investigated bounded and unbounded verifications.

The representations were detailed in Chapter 5, with algorithm models (5.1) and specifications of linearisability (5.2) being similar to previous work. The main contributions of the chapter are the specifications of nonblocking properties (5.3). The formalisation of these for model checking is novel, though similar to independent formalisations.

9.1 Bounded Verification

In Chapter 1, I raised four questions about bounded verification that I attempted to answer in Chapter 6.

1. What size instances would need to be verified in order to give a reasonable level of confidence in an algorithm?

2. What size instances are needed to trigger known bugs in existing algorithms?

In Section 6.1.2, I investigated the minimum sized instances for a number of known linearisability errors and used these results in Section 6.1.3 to formulate two ranges of instances that, if used for bounded verification, would give me different levels of confidence in the correctness of an algorithm.

3. Is this approach equally applicable to both the correctness properties and the progress properties?

The approach I have used is as applicable to nonblocking properties as to linearisability, and I attempted to answer the same questions in Section 6.2. However, the algorithms investigated (a subset of those from Section 6.1) are too few to draw any conclusion from. Further work is needed to answers Questions 1 and 2 for nonblocking properties.

4. Is this approach worth doing before, or instead of, a full verification (using a theorem prover, model checker with abstraction, etc.)?

The results in Sections 6.1.2 and 6.2.2 show that model checking is very effective for finding small bugs. I highly recommend bounded verification of algorithms before any deeper analysis. The cost of setting up model checking and discovering such bugs is likely to be far less than the cost of beginning an unsuccessful full verification attempt, and diagnosing the existence of the bug from the failure.

If a full verification is not practical, bounded verification provides some degree of confidence in the algorithm. However, given current resources, the practically verifiable instances in Section 6.2.4 are too small to engender a high degree of confidence.

I was able to increase the range of practically verifiable instances by using symmetry reduction on the number of threads. If symmetry reduction was able to be applied to the other two parameters also, I think it likely that the bounds identified to answer Question 1 would be practically verifiable.

9.1.1 Future Research

There are a number of avenues of research that can be explored from this point:

- The instance ranges in Section 6.1.3 could be strengthened by investigating even more incorrect algorithms.

- The bounded verification results presented in Section 6.1.4 could be extended by using a tool that is able to apply symmetry reduction to all three parameters (most likely through the use of scalarsets).

- The results in Section 6.2 for nonblocking properties could be extended as thoroughly as for the linearisability results in Section 6.1. Notably, it would be interesting to see if the ranges of confidence differ markedly from those for linearisability, and if they differ between different nonblocking properties.

9.2 Unbounded Verification

In Chapter 1, I raised three questions about using canonical abstraction to verify nonblocking data structures.

1. Can canonical abstraction be used to verify linearisability for nonblocking data structures, with unbounded instances of all three parameters?

I answered this question affirmatively in Chapter 7, introducing several novel instrumentation predicates that allowed two algorithms to be verified using canonical abstraction. My approach differs from other verifications of nonblocking data structures using canonical abstraction [Amit et al., 2007; Berdine et al., 2008] as I have not added another abstraction technique on top, showing that canonical abstraction is sufficient for this task.

2. Can canonical abstraction be used to verify nonblocking properties for concurrent data structures, with unbounded instances of all three parameters?

I was unable to model check progress properties in canonical abstraction, partly due to TVLA's limitations. TVLA is only able to verify safety properties, but research has begun on the combination of canonical abstraction and temporal properties [e.g. Yahav et al., 2001, 2003, 2006].

3. If so, for either of the above, are the abstractions efficient (i.e. are they small enough to enable practical verification)? If not, is it possible to improve them?

I found the canonically abstract model of the queue algorithm to still be too large to practically verify, primarily due to the exponential permutations of thread properties. In Chapter 8, I described a technique for more

aggressively abstracting the threads in each state, thus reducing the state-space, whilst still being able to verify linearisability. Again, this differs from other similar approaches [e.g. Berdine et al., 2008] as it is achieved entirely within the framework of canonical abstraction.

9.2.1 Future Research

There are a number of avenues of research that can be explored from this point:

- The verified algorithms given in Chapters 7 and 8 are a stack and a queue algorithm — both similar implementations using a singly linked list. Some modifications and further instrumentation predicates will almost certainly be needed to verify other types of data structures and other implementations of stacks and queues.
- Investigate using canonical abstraction to verify nonblocking properties.
- The additional instrumentation predicates used in the verifications in Chapter 8 were determined by hand. For practicality and efficiency it would be useful if they could be derived automatically. One possibility would be to adapt existing abstraction refinement techniques [Loginov et al., 2005]. Alternatively, it may be efficient to over define the soft invariants, as in Section 8.2.5, and automatically identify a subset of the non-invariant properties (such as those that are false in the initial state).

Appendix A

Proofs of Canonical Abstraction Compatibility Constraints

In Chapters 7 and 8, I manually defined some additional compatibility constraints, to improve the precision of Coerce (see Section 7.6). These are relatively simple, but if any was invalid it could undermine the verification results. To give confidence in the results, I have manually determined the 2-valued validity of the formulas using natural deduction.

The proofs use the operator $\rightarrow$ rather than $\rhd$, as they are identical with 2-valued semantics (recall from Section 4.3.2). For most of the compatibility constraints there are other constraints that have the same structure; here we show the proof for only one in each group. For example, the proof given for:

$$\mathsf{if}[\mathsf{push}_9, \mathsf{has}[\mathsf{n}]](v) \wedge \mathsf{at}[\mathsf{push}_9](v) \rhd \mathsf{has}[\mathsf{n}](v)$$

is exactly the same, with renaming, for:

$$\mathsf{if}[\mathsf{deq}_{31}, \mathsf{has}[\mathsf{sshead}]](v) \wedge \mathsf{at}[\mathsf{deq}_{31}](v) \rhd \mathsf{has}[\mathsf{sshead}](v)$$

The chapter begins by listing assumptions that are used in the proofs (A.1). The proofs are categorised as soft invariants (A.2), linear instrumentation predicates (A.3), geometric instrumentation predicates (A.4), and reachability instrumentation predicates (A.5).

A.1 Assumptions

The following assumptions are used for the scope of the proofs, listed here for space reasons:

{ Uniqueness of $\mathsf{Head_I}$ }

[A1] $\forall\, v_1, v_2 \bullet \mathsf{Head_I}(v_1) \wedge \mathsf{Head_I}(v_2) \rightarrow v_1 = v_2$

{ Functionality of n }

[A2] $\forall\, v_1, v_2, v_3 \bullet \mathsf{n}(v_1, v_2) \wedge \mathsf{n}(v_1, v_3) \rightarrow v_2 = v_3$

{ Functionality of ss }

[A3] $\forall\, v_1, v_2, v_3 \bullet \mathsf{ss}(v_1, v_2) \wedge \mathsf{ss}(v_1, v_3) \rightarrow v_2 = v_3$

{ Functionality of sshead }

[A4] $\forall\, v_1, v_2, v_3 \bullet \mathsf{sshead}(v_1, v_2) \wedge \mathsf{sshead}(v_1, v_3) \rightarrow v_2 = v_3$

{ Functionality of sstail }

[A5] $\forall\, v_1, v_2, v_3 \bullet \mathsf{sstail}(v_1, v_2) \wedge \mathsf{sstail}(v_1, v_3) \rightarrow v_2 = v_3$

{ Functionality of next }

[A6] $\forall\, v_1, v_2, v_3 \bullet \mathsf{next}(v_1, v_2) \wedge \mathsf{next}(v_1, v_3) \rightarrow v_2 = v_3$

{ Functionality of spec }

[A7] $\forall\, v_1, v_2, v_3 \bullet \mathsf{spec}(v_1, v_2) \wedge \mathsf{spec}(v_1, v_3) \rightarrow v_2 = v_3$

{ Inverse Functionality of spec }

[A8] $\forall\, v_1, v_2, v_3 \bullet \mathsf{spec}(v_1, v_3) \wedge \mathsf{spec}(v_2, v_3) \rightarrow v_1 = v_2$

{ Definition of $\mathsf{if}[\mathsf{push_9}]$ }

[A9] $\forall\, v_1 \bullet \mathsf{if}[\mathsf{push_9}, \mathsf{has}[\mathsf{n}]](v_1) \rightarrow$
$\mathsf{is\_thread}(v_1) \wedge (\mathsf{at}[\mathsf{push_9}](v_1) \rightarrow \mathsf{has}[\mathsf{n}](v_1))$

{ Definition of $\mathsf{if}[\mathsf{push_9}, \mathsf{succ}[\mathsf{n}, \mathsf{ss}]]$ }

[A10] $\forall\, v_1 \bullet \mathsf{if}[\mathsf{push_9}, \mathsf{succ}[\mathsf{n}, \mathsf{ss}]](v_1) \rightarrow$
$\mathsf{is\_thread}(v_1) \wedge (\mathsf{at}[\mathsf{push_9}](v_1) \rightarrow (\mathsf{has}[\mathsf{ss}](v_1) \rightarrow \mathsf{succ}[\mathsf{n}, \mathsf{ss}](v_1)))$

{ Definition of $\mathsf{tonull}[\mathsf{ss}]$ }

[A11] $\forall\, v_1 \bullet \mathsf{tonull}[\mathsf{ss}](v_1) \rightarrow \exists\, v_2 \bullet \mathsf{ss}(v_1, v_2) \wedge \neg\, \mathsf{has}[\mathsf{next}](v_2)$

{ Definition of $\mathsf{same}[\mathsf{sshead}, \mathsf{sstail}]$ }

[A12] $\forall\, v_1 \bullet \mathsf{same}[\mathsf{sshead}, \mathsf{sstail}](v_1) \rightarrow \exists\, v_2 \bullet \mathsf{sshead}(v_1, v_2) \wedge \mathsf{sstail}(v_1, v_2)$

{ Definition of succ[n, ss] }

[A13] $\forall\, v_1 \bullet \mathsf{succ}[\mathsf{n}, \mathsf{ss}](v_1) \rightarrow \exists\, v_2, v_3 \bullet \mathsf{n}(v_1, v_2) \wedge \mathsf{next}(v_2, v_3) \wedge \mathsf{ss}(v_1, v_3)$

{ Definition of commutes }

[A14] $\forall\, v_1 \bullet \mathsf{commutes}(v_1) \rightarrow$

$\exists\, v_2, v_3, v_4 \bullet \mathsf{next}(v_1, v_2) \wedge \mathsf{spec}(v_1, v_3) \wedge \mathsf{next}(v_3, v_4) \wedge \mathsf{spec}(v_2, v_4)$

{ Definition of ordered[sshead, sstail] }

[A15] $\forall\, v_1 \bullet \mathsf{ordered}[\mathsf{sshead}, \mathsf{sstail}](v_1) \rightarrow$

$\exists\, v_2, v_3 \bullet \mathsf{sshead}(v_1, v_2) \wedge \mathsf{sstail}(v_1, v_3) \wedge (\mathsf{reach}_{\mathsf{l}}(v_2) \rightarrow \mathsf{reach}_{\mathsf{l}}(v_3))$

{ Definition of circ }

[A16] $\forall\, v_1 \bullet \mathsf{circ}(v_1) \rightarrow \mathsf{next}^{+}(v_1, v_1)$

{ Definition of circ }

[A17] $\forall\, v_1 \bullet \mathsf{next}^{+}(v_1, v_1) \rightarrow \mathsf{circ}(v_1)$

{ Definition of $\mathsf{reach}_{\mathsf{l}}$ }

[A18] $\forall\, v_1 \bullet \mathsf{reach}_{\mathsf{l}}(v_1) \rightarrow \exists\, v_2 \bullet \mathsf{Head}_{\mathsf{l}}(v_2) \wedge \mathsf{next}^{*}(v_2, v_1)$

{ Definition of $\mathsf{reach}_{\mathsf{l}}$ }

[A19] $\forall\, v_1 \bullet (\exists\, v_2 \bullet \mathsf{Head}_{\mathsf{l}}(v_2) \wedge \mathsf{next}^{*}(v_2, v_1)) \rightarrow \mathsf{reach}_{\mathsf{l}}(v_1)$

A.2 Soft Invariants

A.2.1 Single predicate

$\mathsf{if}[\mathsf{push}_9, \mathsf{has}[\mathsf{n}]](v) \wedge \mathsf{at}[\mathsf{push}_9](v) \rhd \mathsf{has}[\mathsf{n}](v)$

(1)	$\boxed{\mathsf{if}[\mathsf{push}_9, \mathsf{has}[\mathsf{n}]](t_1) \wedge \mathsf{at}[\mathsf{push}_9](t_1)}$	
(2)	$\mathsf{if}[\mathsf{push}_9, \mathsf{has}[\mathsf{n}]](t_1)$	(1), $\wedge\mathbf{E}$

(3)	$\mathsf{if}[\mathsf{push}_9, \mathsf{has}[\mathsf{n}]](t_1) \rightarrow \mathsf{is\_thread}(t_1)$ $\wedge\ (\mathsf{at}[\mathsf{push}_9](t_1) \rightarrow \mathsf{has}[\mathsf{n}](t_1))$	[A9], $\forall\mathbf{E}$
(4)	$\mathsf{is\_thread}(t_1) \wedge (\mathsf{at}[\mathsf{push}_9](t_1) \rightarrow \mathsf{has}[\mathsf{n}](t_1))$	(2), (3), $\rightarrow\mathbf{E}$
(5)	$\mathsf{at}[\mathsf{push}_9](t_1) \rightarrow \mathsf{has}[\mathsf{n}](t_1)$	(4), $\wedge\mathbf{E}$
(6)	$\mathsf{at}[\mathsf{push}_9](t_1)$	(1), $\wedge\mathbf{E}$
(7)	$\mathsf{has}[\mathsf{n}](t_1)$	(5), (6), $\rightarrow\mathbf{E}$
(8)	$\mathsf{if}[\mathsf{push}_9, \mathsf{has}[\mathsf{n}]](t_1) \wedge \mathsf{at}[\mathsf{push}_9](t_1) \rightarrow \mathsf{has}[\mathsf{n}](t_1)$	(1), (7), $\rightarrow\mathbf{I}$

A.2.2 Conditional property

$\mathsf{if}[\mathsf{push}_9, \mathsf{succ}[\mathsf{n}, \mathsf{ss}]](v) \wedge \mathsf{at}[\mathsf{push}_9](v) \wedge \mathsf{has}[\mathsf{ss}](v) \rhd \mathsf{succ}[\mathsf{n}, \mathsf{ss}](v)$

(1)	$\boxed{\mathsf{if}[\mathsf{push}_9, \mathsf{succ}[\mathsf{n}, \mathsf{ss}]](t_1) \wedge \mathsf{at}[\mathsf{push}_9](t_1) \wedge \mathsf{has}[\mathsf{ss}](t_1)}$	
(2)	$\mathsf{if}[\mathsf{push}_9, \mathsf{succ}[\mathsf{n}, \mathsf{ss}]](t_1)$	(1), $\wedge\mathbf{E}$
(3)	$\mathsf{if}[\mathsf{push}_9, \mathsf{succ}[\mathsf{n}, \mathsf{ss}]](t_1) \rightarrow \mathsf{is\_thread}(t_1) \wedge$ $(\mathsf{at}[\mathsf{push}_9](t_1) \rightarrow (\mathsf{has}[\mathsf{ss}](t_1) \rightarrow \mathsf{succ}[\mathsf{n}, \mathsf{ss}](t_1)))$	[A10], $\forall\mathbf{E}$
(4)	$\mathsf{is\_thread}(t_1) \wedge (\mathsf{at}[\mathsf{push}_9](t_1) \rightarrow (\mathsf{has}[\mathsf{ss}](t_1) \rightarrow \mathsf{succ}[\mathsf{n}, \mathsf{ss}](t_1)))$	(2), (3), $\rightarrow\mathbf{E}$
(5)	$\mathsf{at}[\mathsf{push}_9](t_1) \rightarrow (\mathsf{has}[\mathsf{ss}](t_1) \rightarrow \mathsf{succ}[\mathsf{n}, \mathsf{ss}](t_1))$	(4), $\wedge\mathbf{E}$
(6)	$\mathsf{at}[\mathsf{push}_9](t_1)$	(1), $\wedge\mathbf{E}$
(7)	$\mathsf{has}[\mathsf{ss}](t_1) \rightarrow \mathsf{succ}[\mathsf{n}, \mathsf{ss}](t_1))$	(5), (6), $\rightarrow\mathbf{E}$
(8)	$\mathsf{has}[\mathsf{ss}](t_1)$	(1), $\wedge\mathbf{E}$
(9)	$\mathsf{succ}[\mathsf{n}, \mathsf{ss}](t_1)$	(7), (8), $\rightarrow\mathbf{E}$
(10)	$\mathsf{if}[\mathsf{push}_9, \mathsf{succ}[\mathsf{n}, \mathsf{ss}]](t_1) \wedge \mathsf{at}[\mathsf{push}_9](t_1) \wedge \mathsf{has}[\mathsf{ss}](t_1) \rightarrow$ $\mathsf{succ}[\mathsf{n}, \mathsf{ss}](t_1)$	(1), (9), $\rightarrow\mathbf{I}$

A.3 Linear

A.3.1 To null

$\exists\, t \bullet \mathsf{tonull}[\mathsf{ss}](t) \wedge \mathsf{ss}(t, n) \rhd \neg\, \mathsf{has}[\mathsf{next}](n)$

(1)	$\boxed{\exists\, v_1 \bullet \mathsf{tonull}[\mathsf{ss}](v_1) \wedge \mathsf{ss}(v_1, n_1)}$	
(2)	$\quad\boxed{\mathsf{tonull}[\mathsf{ss}](t_1) \wedge \mathsf{ss}(t_1, n_1)}$	
(3)	$\quad\mathsf{tonull}[\mathsf{ss}](t_1)$	(2), $\wedge\mathbf{E}$
(4)	$\quad\mathsf{tonull}[\mathsf{ss}](t_1) \rightarrow \exists\, v_2 \bullet \mathsf{ss}(t_1, v_2) \wedge \neg\, \mathsf{has}[\mathsf{next}](v_2)$	[A11], $\forall\mathbf{E}$
(5)	$\quad\exists\, v_2 \bullet \mathsf{ss}(t_1, v_2) \wedge \neg\, \mathsf{has}[\mathsf{next}](v_2)$	(3), (4), $\rightarrow\mathbf{E}$
(6)	$\quad\boxed{\mathsf{ss}(t_1, n_2) \wedge \neg\, \mathsf{has}[\mathsf{next}](n_2)}$	
(7)	$\quad\quad\mathsf{ss}(t_1, n_1) \wedge \mathsf{ss}(t_1, n_2) \rightarrow n_1 = n_2$	[A3], $\forall\mathbf{E}$
(8)	$\quad\quad\mathsf{ss}(t_1, n_1)$	(2), $\wedge\mathbf{E}$
(9)	$\quad\quad\mathsf{ss}(t_1, n_2)$	(6), $\wedge\mathbf{E}$
(10)	$\quad\quad\mathsf{ss}(t_1, n_1) \wedge \mathsf{ss}(t_1, n_2)$	(8), (9), $\wedge\mathbf{I}$
(11)	$\quad\quad n_1 = n_2$	(7), (10), $\rightarrow\mathbf{E}$
(12)	$\quad\quad\neg\, \mathsf{has}[\mathsf{next}](n_2)$	(6), $\wedge\mathbf{E}$
(13)	$\quad\quad\neg\, \mathsf{has}[\mathsf{next}](n_1)$	(12), $=\mathbf{E}$
(14)	$\quad\neg\, \mathsf{has}[\mathsf{next}](n_1)$	(5), (6), (13), $\exists\mathbf{E}$
(15)	$\neg\, \mathsf{has}[\mathsf{next}](n_1)$	(1), (2), (14), $\exists\mathbf{E}$
(16)	$(\exists\, v_1 \bullet \mathsf{tonull}[\mathsf{ss}](v_1) \wedge \mathsf{ss}(v_1, n_1)) \rightarrow \neg\, \mathsf{has}[\mathsf{next}](n_1)$	(1), (15), $\rightarrow\mathbf{I}$

A.3.2 Same 1

$\mathsf{same}[\mathsf{sshead}, \mathsf{sstail}](t) \wedge \mathsf{sshead}(t, n) \rhd \mathsf{sstail}(t, n)$

(1)	$\boxed{\mathsf{same}[\mathsf{sshead}, \mathsf{sstail}](t_1) \wedge \mathsf{sshead}(t_1, n_1)}$	
(2)	$\quad\mathsf{same}[\mathsf{sshead}, \mathsf{sstail}](t_1)$	(1), $\wedge\mathbf{E}$

(3)	$\mathsf{same}[\mathsf{sshead}, \mathsf{sstail}](t_1) \rightarrow$	[A12], $\forall\mathbf{E}$
	$\quad \exists\, v_2 \bullet \mathsf{sshead}(t_1, v_2) \wedge \mathsf{sstail}(t_1, v_2)$	
(4)	$\exists\, v_2 \bullet \mathsf{sshead}(t_1, v_2) \wedge \mathsf{sstail}(t_1, v_2)$	(2), (3), $\rightarrow\mathbf{E}$
(5)	$\boxed{\mathsf{sshead}(t_1, n_2) \wedge \mathsf{sstail}(t_1, n_2)}$	
(6)	$\mathsf{sshead}(t_1, n_1) \wedge \mathsf{sshead}(t_1, n_2) \rightarrow n_1 = n_2$	[A4], $\forall\mathbf{E}$
(7)	$\mathsf{sshead}(t_1, n_1)$	(1), $\wedge\mathbf{E}$
(8)	$\mathsf{sshead}(t_1, n_2)$	(5), $\wedge\mathbf{E}$
(9)	$\mathsf{sshead}(t_1, n_1) \wedge \mathsf{sshead}(t_1, n_2)$	(7), (8), $\wedge\mathbf{I}$
(10)	$n_1 = n_2$	(6), (9), $\rightarrow\mathbf{E}$
(11)	$\mathsf{sstail}(t_1, n_2)$	(5), $\wedge\mathbf{E}$
(12)	$\mathsf{sstail}(t_1, n_1)$	(10), (11), $=\mathbf{E}$
(13)	$\mathsf{sstail}(t_1, n_1)$	(4), (5), (12), $\exists\mathbf{E}$
(14)	$\mathsf{same}[\mathsf{sshead}, \mathsf{sstail}](t_1) \wedge \mathsf{sshead}(t_1, n_1)$	(1), (13), $\rightarrow\mathbf{I}$
	$\quad \rightarrow \mathsf{sstail}(t_1, n_1)$	

A.3.3 Same 2

$\mathsf{same}[\mathsf{sshead}, \mathsf{sstail}](t) \wedge \mathsf{sstail}(t, n) \rhd \mathsf{sshead}(t, n)$

(1)	$\boxed{\mathsf{same}[\mathsf{sshead}, \mathsf{sstail}](t_1) \wedge \mathsf{sstail}(t_1, n_1)}$	
(2)	$\mathsf{same}[\mathsf{sshead}, \mathsf{sstail}](t_1)$	(1), $\wedge\mathbf{E}$
(3)	$\mathsf{same}[\mathsf{sshead}, \mathsf{sstail}](t_1) \rightarrow$	[A12], $\forall\mathbf{E}$
	$\quad \exists\, v_2 \bullet \mathsf{sshead}(t_1, v_2) \wedge \mathsf{sstail}(t_1, v_2)$	
(4)	$\exists\, v_2 \bullet \mathsf{sshead}(t_1, v_2) \wedge \mathsf{sstail}(t_1, v_2)$	(2), (3), $\rightarrow\mathbf{E}$
(5)	$\boxed{\mathsf{sshead}(t_1, n_2) \wedge \mathsf{sstail}(t_1, n_2)}$	
(6)	$\mathsf{sstail}(t_1, n_1) \wedge \mathsf{sstail}(t_1, n_2) \rightarrow n_1 = n_2$	[A5], $\forall\mathbf{E}$

(7)	$\mathsf{sstail}(t_1, n_1)$	(1), $\wedge\mathbf{E}$
(8)	$\mathsf{sstail}(t_1, n_2)$	(5), $\wedge\mathbf{E}$
(9)	$\mathsf{sstail}(t_1, n_1) \wedge \mathsf{sstail}(t_1, n_2)$	(7), (8), $\wedge\mathbf{I}$
(10)	$n_1 = n_2$	(6), (9), $\rightarrow\mathbf{E}$
(11)	$\mathsf{sshead}(t_1, n_2)$	(5), $\wedge\mathbf{E}$
(12)	$\mathsf{sshead}(t_1, n_1)$	(10), (11), $=\mathbf{E}$
(13)	$\mathsf{sshead}(t_1, n_1)$	(4), (5), (12), $\exists\mathbf{E}$
(14)	$\mathsf{same}[\mathsf{sshead}, \mathsf{sstail}](t_1) \wedge \mathsf{sstail}(t_1, n_1) \rightarrow \mathsf{sshead}(t_1, n_1)$	(1), (13), $\rightarrow\mathbf{I}$

A.3.4 Ordered

$\exists\, t_1, n_1 \bullet \mathsf{ordered}[\mathsf{sshead}, \mathsf{sstail}](t_1) \wedge \mathsf{sshead}(t_1, n_1) \wedge \mathsf{sstail}(t_1, n_2) \wedge \mathsf{reach}_{\mathsf{l}}(n_1) \rhd \mathsf{reach}_{\mathsf{l}}(n_2)$

(1)	$\exists\, v_1, v_2 \bullet \mathsf{ordered}[\mathsf{sshead}, \mathsf{sstail}](v_1) \wedge \mathsf{sshead}(v_1, v_2) \wedge \mathsf{sstail}(v_1, n_2) \wedge \mathsf{reach}_{\mathsf{l}}(v_2)$		
(2)	$\mathsf{ordered}[\mathsf{sshead}, \mathsf{sstail}](t_1) \wedge \mathsf{sshead}(t_1, n_1) \wedge \mathsf{sstail}(t_1, n_2) \wedge \mathsf{reach}_{\mathsf{l}}(n_1)$		
(3)	$\mathsf{ordered}[\mathsf{sshead}, \mathsf{sstail}](t_1)$	(2), $\wedge\mathbf{E}$	
(4)	$\mathsf{ordered}[\mathsf{sshead}, \mathsf{sstail}](t_1) \rightarrow \exists\, v_2, v_3 \bullet \mathsf{sshead}(t_1, v_2) \wedge \mathsf{sstail}(t_1, v_3) \wedge (\mathsf{reach}_{\mathsf{l}}(v_2) \rightarrow \mathsf{reach}_{\mathsf{l}}(v_3))$	[A15], $\forall\mathbf{E}$	
(5)	$\exists\, v_2, v_3 \bullet \mathsf{sshead}(t_1, v_2) \wedge \mathsf{sstail}(t_1, v_3) \wedge (\mathsf{reach}_{\mathsf{l}}(v_2) \rightarrow \mathsf{reach}_{\mathsf{l}}(v_3))$	(3), (4), $\rightarrow\mathbf{E}$	
(6)	$\mathsf{sshead}(t_1, n_3) \wedge \mathsf{sstail}(t_1, n_4) \wedge (\mathsf{reach}_{\mathsf{l}}(n_3) \rightarrow \mathsf{reach}_{\mathsf{l}}(n_4))$		

(7)	$\mathsf{sshead}(t_1, n_1) \wedge \mathsf{sshead}(t_1, n_3) \rightarrow n_1 = n_3$	[A4], $\forall\mathbf{E}$
(8)	$\mathsf{sshead}(t_1, n_1)$	(2), $\wedge\mathbf{E}$
(9)	$\mathsf{sshead}(t_1, n_3)$	(6), $\wedge\mathbf{E}$
(10)	$\mathsf{sshead}(t_1, n_1) \wedge \mathsf{sshead}(t_1, n_3)$	(8), (9), $\wedge\mathbf{I}$
(11)	$n_1 = n_3$	(7), (10), $\rightarrow\mathbf{E}$
(12)	$\mathsf{sstail}(t_1, n_2) \wedge \mathsf{sstail}(t_1, n_4) \rightarrow n_2 = n_4$	[A5], $\forall\mathbf{E}$
(13)	$\mathsf{sstail}(t_1, n_2)$	(2), $\wedge\mathbf{E}$
(14)	$\mathsf{sstail}(t_1, n_4)$	(6), $\wedge\mathbf{E}$
(15)	$\mathsf{sstail}(t_1, n_2) \wedge \mathsf{sstail}(t_1, n_4)$	(13), (14), $\wedge\mathbf{I}$
(16)	$n_2 = n_4$	(12), (15), $\rightarrow\mathbf{E}$
(17)	$\mathsf{reach}_{\mathsf{I}}(n_3) \rightarrow \mathsf{reach}_{\mathsf{I}}(n_4)$	(6), $\wedge\mathbf{E}$
(18)	$\mathsf{reach}_{\mathsf{I}}(n_1)$	(2), $\wedge\mathbf{E}$
(19)	$\mathsf{reach}_{\mathsf{I}}(n_3)$	(11), (18), $=\mathbf{E}$
(20)	$\mathsf{reach}_{\mathsf{I}}(n_4)$	(17), (19), $\rightarrow\mathbf{E}$
(21)	$\mathsf{reach}_{\mathsf{I}}(n_2)$	(16), (20), $=\mathbf{E}$
(22)	$\mathsf{reach}_{\mathsf{I}}(n_2)$	(5), (6), (21), $\exists\mathbf{E}$
(23)	$\mathsf{reach}_{\mathsf{I}}(n_2)$	(1), (2), (22), $\exists\mathbf{E}$
(24)	$(\exists\, v_1, v_2 \bullet \mathsf{ordered}[\mathsf{sshead}, \mathsf{sstail}](v_1) \wedge \mathsf{sshead}(v_1, v_2) \wedge \mathsf{sstail}(v_1, n_2) \wedge \mathsf{reach}_{\mathsf{I}}(v_2)) \rightarrow \mathsf{reach}_{\mathsf{I}}(n_2)$	(1), (23), $\rightarrow\mathbf{I}$

A.4 Geometric

A.4.1 Triangle 1

$\exists\, t_1 \bullet \mathsf{succ}[\mathsf{n}, \mathsf{ss}](t_1) \wedge \mathsf{n}(t_1, n_1) \wedge \mathsf{ss}(t_1, n_2) \rhd \mathsf{next}(n_1, n_2)$

(1)	$\boxed{\exists\, v_1 \bullet \mathsf{succ}[\mathsf{n}, \mathsf{ss}](v_1) \wedge \mathsf{n}(v_1, n_1) \wedge \mathsf{ss}(v_1, n_2)}$	

(2)	$\boxed{\mathsf{succ}[\mathsf{n},\mathsf{ss}](t_1) \wedge \mathsf{n}(t_1,n_1) \wedge \mathsf{ss}(t_1,n_2)}$	
(3)	$\mathsf{succ}[\mathsf{n},\mathsf{ss}](t_1)$	(2), $\wedge\mathbf{E}$
(4)	$\mathsf{succ}[\mathsf{n},\mathsf{ss}](t_1) \rightarrow \exists v_2, v_3 \bullet \mathsf{n}(t_1,v_2)$ $\wedge\, \mathsf{next}(v_2,v_3) \wedge \mathsf{ss}(t_1,v_3)$	[A13], $\forall\mathbf{E}$
(5)	$\exists v_2, v_3 \bullet \mathsf{n}(t_1,v_2) \wedge \mathsf{next}(v_2,v_3) \wedge \mathsf{ss}(t_1,v_3)$	(3), (4), $\rightarrow\mathbf{E}$
(6)	$\boxed{\mathsf{n}(t_1,n_3) \wedge \mathsf{next}(n_3,n_4) \wedge \mathsf{ss}(t_1,n_4)}$	
(7)	$\mathsf{n}(t_1,n_1) \wedge \mathsf{n}(t_1,n_3) \rightarrow n_1 = n_3$	[A2], $\forall\mathbf{E}$
(8)	$\mathsf{n}(t_1,n_1)$	(2), $\wedge\mathbf{E}$
(9)	$\mathsf{n}(t_1,n_3)$	(6), $\wedge\mathbf{E}$
(10)	$\mathsf{n}(t_1,n_1) \wedge \mathsf{n}(t_1,n_3)$	(8), (9), $\wedge\mathbf{I}$
(11)	$n_1 = n_3$	(7), (10), $\rightarrow\mathbf{E}$
(12)	$\mathsf{ss}(t_1,n_2) \wedge \mathsf{ss}(t_1,n_4) \rightarrow n_2 = n_4$	[A3], $\forall\mathbf{E}$
(13)	$\mathsf{ss}(t_1,n_2)$	(2), $\wedge\mathbf{E}$
(14)	$\mathsf{ss}(t_1,n_4)$	(6), $\wedge\mathbf{E}$
(15)	$\mathsf{ss}(t_1,n_2) \wedge \mathsf{ss}(t_1,n_4)$	(13), (14), $\wedge\mathbf{I}$
(16)	$n_2 = n_4$	(12), (15), $\rightarrow\mathbf{E}$
(17)	$\mathsf{next}(n_3,n_4)$	(6), $\wedge\mathbf{E}$
(18)	$\mathsf{next}(n_1,n_2)$	(11), (16), (17), $=\mathbf{E}$
(19)	$\mathsf{next}(n_1,n_2)$	(5), (6), (18), $\exists\mathbf{E}$
(20)	$\mathsf{next}(n_1,n_2)$	(1), (2), (19), $\exists\mathbf{E}$
(21)	$(\exists v_1 \bullet \mathsf{succ}[\mathsf{n},\mathsf{ss}](v_1) \wedge \mathsf{n}(v_1,n_1) \wedge \mathsf{ss}(v_1,n_2))$ $\rightarrow \mathsf{next}(n_1,n_2)$	(1), (20), $\rightarrow\mathbf{I}$

A.4.2 Triangle 2

$\exists n_1 \bullet \mathsf{succ}[\mathsf{n},\mathsf{ss}](t_1) \wedge \mathsf{n}(t_1,n_1) \wedge \mathsf{next}(n_1,n_2) \rhd \mathsf{ss}(t_1,n_2)$

(1)	$\boxed{\exists v_1 \bullet \mathsf{succ}[\mathsf{n},\mathsf{ss}](t_1) \wedge \mathsf{n}(t_1, v_1) \wedge \mathsf{next}(v_1, n_2)}$	
(2)	$\quad\boxed{\mathsf{succ}[\mathsf{n},\mathsf{ss}](t_1) \wedge \mathsf{n}(t_1, n_1) \wedge \mathsf{next}(n_1, n_2)}$	
(3)	$\quad\mathsf{succ}[\mathsf{n},\mathsf{ss}](t_1)$	(2), $\wedge\mathbf{E}$
(4)	$\quad\mathsf{succ}[\mathsf{n},\mathsf{ss}](t_1) \rightarrow \exists v_2, v_3 \bullet \mathsf{n}(t_1, v_2)$	[A13], $\forall\mathbf{E}$
	$\qquad\wedge\, \mathsf{next}(v_2, v_3) \wedge \mathsf{ss}(t_1, v_3)$	
(5)	$\quad\exists v_2, v_3 \bullet \mathsf{n}(t_1, v_2) \wedge \mathsf{next}(v_2, v_3) \wedge \mathsf{ss}(t_1, v_3)$	(3), (4), $\rightarrow\mathbf{E}$
(6)	$\quad\boxed{\mathsf{n}(t_1, n_3) \wedge \mathsf{next}(n_3, n_4) \wedge \mathsf{ss}(t_1, n_4)}$	
(7)	$\qquad\mathsf{n}(t_1, n_1) \wedge \mathsf{n}(t_1, n_3) \rightarrow n_1 = n_3$	[A2], $\forall\mathbf{E}$
(8)	$\qquad\mathsf{n}(t_1, n_1)$	(2), $\wedge\mathbf{E}$
(9)	$\qquad\mathsf{n}(t_1, n_3)$	(6), $\wedge\mathbf{E}$
(10)	$\qquad\mathsf{n}(t_1, n_1) \wedge \mathsf{n}(t_1, n_3)$	(8), (9), $\wedge\mathbf{I}$
(11)	$\qquad n_1 = n_3$	(7), (10), $\rightarrow\mathbf{E}$
(12)	$\qquad\mathsf{next}(n_1, n_2) \wedge \mathsf{next}(n_1, n_4) \rightarrow n_2 = n_4$	[A6], $\forall\mathbf{E}$
(13)	$\qquad\mathsf{next}(n_1, n_2)$	(2), $\wedge\mathbf{E}$
(14)	$\qquad\mathsf{next}(n_3, n_4)$	(6), $\wedge\mathbf{E}$
(15)	$\qquad\mathsf{next}(n_1, n_4)$	(11), (14), $=\mathbf{E}$
(16)	$\qquad\mathsf{next}(n_1, n_2) \wedge \mathsf{next}(n_1, n_4)$	(13), (14), $\wedge\mathbf{I}$
(17)	$\qquad n_2 = n_4$	(12), (16), $\rightarrow\mathbf{E}$
(18)	$\qquad\mathsf{ss}(t_1, n_4)$	(6), $\wedge\mathbf{E}$
(19)	$\qquad\mathsf{ss}(t_1, n_2)$	(17), (18), $=\mathbf{E}$
(20)	$\quad\mathsf{ss}(t_1, n_2)$	(5), (6), (19), $\exists\mathbf{E}$
(21)	$\mathsf{ss}(t_1, n_2)$	(1), (2), (20), $\exists\mathbf{E}$
(22)	$(\exists v_1 \bullet \mathsf{succ}[\mathsf{n},\mathsf{ss}](t_1) \wedge \mathsf{n}(t_1, v_1) \wedge \mathsf{next}(v_1, n_2))$	(1), (21), $\rightarrow\mathbf{I}$
	$\qquad\rightarrow \mathsf{ss}(t_1, n_2)$	

A.4.3 Square 1

$\exists n_1, n_2 \bullet \mathsf{commutes}(n_1) \wedge$
$\mathsf{next}(n_1, n_2) \wedge \mathsf{spec}(n_1, n_3) \wedge \mathsf{spec}(n_2, n_4) \rhd \mathsf{next}(n_3, n_4)$

(1)	$\exists v_1, v_2 \bullet \mathsf{commutes}(v_1) \wedge \mathsf{next}(v_1, v_2) \wedge \mathsf{spec}(v_1, n_3) \wedge \mathsf{spec}(v_2, n_4)$	
(2)	$\mathsf{commutes}(n_1) \wedge \mathsf{next}(n_1, n_2) \wedge \mathsf{spec}(n_1, n_3) \wedge \mathsf{spec}(n_2, n_4)$	
(3)	$\mathsf{commutes}(n_1)$	(2), $\wedge$**E**
(4)	$\mathsf{commutes}(n_1) \rightarrow \exists v_2, v_3, v_4 \bullet \mathsf{next}(n_1, v_2) \wedge \mathsf{spec}(n_1, v_3) \wedge \mathsf{next}(v_3, v_4) \wedge \mathsf{spec}(v_2, v_4)$	[A14], $\forall$**E**
(5)	$\exists v_2, v_3, v_4 \bullet \mathsf{next}(n_1, v_2) \wedge \mathsf{spec}(n_1, v_3) \wedge \mathsf{next}(v_3, v_4) \wedge \mathsf{spec}(v_2, v_4)$	(3), (4), $\rightarrow$**E**
(6)	$\mathsf{next}(n_1, n_5) \wedge \mathsf{spec}(n_1, n_6) \wedge \mathsf{next}(n_6, n_7) \wedge \mathsf{spec}(n_5, n_7)$	
(7)	$\mathsf{next}(n_1, n_2) \wedge \mathsf{next}(n_1, n_5) \rightarrow n_2 = n_5$	[A6], $\forall$**E**
(8)	$\mathsf{next}(n_1, n_2)$	(2), $\wedge$**E**
(9)	$\mathsf{next}(n_1, n_5)$	(6), $\wedge$**E**
(10)	$\mathsf{next}(n_1, n_2) \wedge \mathsf{next}(n_1, n_5)$	(8), (9), $\wedge$**I**
(11)	$n_2 = n_5$	(7), (10), $\rightarrow$**E**
(12)	$\mathsf{spec}(n_1, n_3) \wedge \mathsf{spec}(n_1, n_6) \rightarrow n_3 = n_6$	[A7], $\forall$**E**
(13)	$\mathsf{spec}(n_1, n_3)$	(2), $\wedge$**E**
(14)	$\mathsf{spec}(n_1, n_6)$	(6), $\wedge$**E**
(15)	$\mathsf{spec}(n_1, n_3) \wedge \mathsf{spec}(n_1, n_6)$	(13), (14), $\wedge$**I**
(16)	$n_3 = n_6$	(12), (15), $\rightarrow$**E**

(17)	$\mathsf{spec}(n_2, n_4) \wedge \mathsf{spec}(n_2, n_7) \rightarrow n_4 = n_7$	[A7], $\forall\mathbf{E}$
(18)	$\mathsf{spec}(n_2, n_4)$	(2), $\wedge\mathbf{E}$
(19)	$\mathsf{spec}(n_5, n_7)$	(6), $\wedge\mathbf{E}$
(20)	$\mathsf{spec}(n_2, n_7)$	(11), (19), $=\mathbf{E}$
(21)	$\mathsf{spec}(n_2, n_4) \wedge \mathsf{spec}(n_2, n_7)$	(18), (20), $\wedge\mathbf{I}$
(22)	$n_4 = n_7$	(17), (21), $\rightarrow\mathbf{E}$
(23)	$\mathsf{next}(n_6, n_7)$	(6), $\wedge\mathbf{E}$
(24)	$\mathsf{next}(n_3, n_4)$	(16), (22), (23), $=\mathbf{E}$
(25)	$\mathsf{next}(n_3, n_4)$	(5), (6), (24), $\exists\mathbf{E}$
(26)	$\mathsf{next}(n_3, n_4)$	(1), (2), (25), $\exists\mathbf{E}$
(27)	$(\exists v_1, v_2 \bullet \mathsf{commutes}(v_1) \wedge \mathsf{next}(v_1, v_2) \wedge \mathsf{spec}(v_1, n_3) \wedge \mathsf{spec}(v_2, n_4)) \rightarrow \mathsf{next}(n_3, n_4)$	(1), (26), $\rightarrow\mathbf{I}$

A.4.4 Square 2

$$\exists n_1, n_3 \bullet \mathsf{commutes}(n_1) \wedge \mathsf{next}(n_1, n_2) \wedge \mathsf{spec}(n_1, n_3) \wedge \mathsf{next}(n_3, n_4) \rhd \mathsf{spec}(n_2, n_4)$$

(1)	$\exists v_1, v_2 \bullet \mathsf{commutes}(v_1) \wedge \mathsf{next}(v_1, n_2) \wedge \mathsf{spec}(v_1, v_2) \wedge \mathsf{next}(v_2, n_4)$	
(2)	$\mathsf{commutes}(n_1) \wedge \mathsf{next}(n_1, n_2) \wedge \mathsf{spec}(n_1, n_3) \wedge \mathsf{next}(n_3, n_4)$	
(3)	$\mathsf{commutes}(n_1)$	(2), $\wedge\mathbf{E}$
(4)	$\mathsf{commutes}(n_1) \rightarrow \exists v_2, v_3, v_4 \bullet \mathsf{next}(n_1, v_2) \wedge \mathsf{spec}(n_1, v_3) \wedge \mathsf{next}(v_3, v_4) \wedge \mathsf{spec}(v_2, v_4)$	[A14], $\forall\mathbf{E}$

(5)	$\exists v_2, v_3, v_4 \bullet \mathsf{next}(n_1, v_2) \wedge \mathsf{spec}(n_1, v_3) \wedge \mathsf{next}(v_3, v_4) \wedge \mathsf{spec}(v_2, v_4)$	(3), (4), $\rightarrow$**E**
(6)	$\mathsf{next}(n_1, n_5) \wedge \mathsf{spec}(n_1, n_6) \wedge \mathsf{next}(n_6, n_7) \wedge \mathsf{spec}(n_5, n_7)$	
(7)	$\mathsf{next}(n_1, n_2) \wedge \mathsf{next}(n_1, n_5) \rightarrow n_2 = n_5$	[A6], $\forall$**E**
(8)	$\mathsf{next}(n_1, n_2)$	(2), $\wedge$**E**
(9)	$\mathsf{next}(n_1, n_5)$	(6), $\wedge$**E**
(10)	$\mathsf{next}(n_1, n_2) \wedge \mathsf{next}(n_1, n_5)$	(8), (9), $\wedge$**I**
(11)	$n_2 = n_5$	(7), (10), $\rightarrow$**E**
(12)	$\mathsf{spec}(n_1, n_3) \wedge \mathsf{spec}(n_1, n_6) \rightarrow n_3 = n_6$	[A7], $\forall$**E**
(13)	$\mathsf{spec}(n_1, n_3)$	(2), $\wedge$**E**
(14)	$\mathsf{spec}(n_1, n_6)$	(6), $\wedge$**E**
(15)	$\mathsf{spec}(n_1, n_3) \wedge \mathsf{spec}(n_1, n_6)$	(13), (14), $\wedge$**I**
(16)	$n_3 = n_6$	(12), (15), $\rightarrow$**E**
(17)	$\mathsf{next}(n_3, n_4) \wedge \mathsf{next}(n_3, n_7) \rightarrow n_4 = n_7$	[A6], $\forall$**E**
(18)	$\mathsf{next}(n_3, n_4)$	(2), $\wedge$**E**
(19)	$\mathsf{next}(n_6, n_7)$	(6), $\wedge$**E**
(20)	$\mathsf{next}(n_3, n_7)$	(16), (19), $=$**E**
(21)	$\mathsf{next}(n_3, n_4) \wedge \mathsf{next}(n_3, n_7)$	(18), (20), $\wedge$**I**
(22)	$n_4 = n_7$	(17), (21), $\rightarrow$**E**
(23)	$\mathsf{spec}(n_5, n_7)$	(6), $\wedge$**E**
(24)	$\mathsf{spec}(n_2, n_4)$	(11), (22), (23), $=$**E**
(25)	$\mathsf{spec}(n_2, n_4)$	(5), (6), (24), $\exists$**E**
(26)	$\mathsf{spec}(n_2, n_4)$	(1), (2), (25), $\exists$**E**

(27) $(\exists v_1, v_2 \bullet \mathsf{commutes}(v_1) \wedge \mathsf{next}(v_1, n_2) \wedge \mathsf{spec}(v_1, v_2) \wedge \mathsf{next}(v_2, n_4)) \rightarrow \mathsf{spec}(n_2, n_4)$ (1), (26), $\rightarrow$**I**

A.4.5 Square 3

$\exists n_3, n_4 \bullet \mathsf{commutes}(n_1) \wedge \mathsf{spec}(n_1, n_3) \wedge \mathsf{next}(n_3, n_4) \wedge \mathsf{spec}(n_2, n_4) \rhd \mathsf{next}(n_1, n_2)$

(1)	$\exists v_1, v_2 \bullet \mathsf{commutes}(n_1) \wedge \mathsf{spec}(n_1, v_1) \wedge \mathsf{next}(v_1, v_2) \wedge \mathsf{spec}(n_2, v_2)$	
(2)	$\mathsf{commutes}(n_1) \wedge \mathsf{spec}(n_1, n_3) \wedge \mathsf{next}(n_3, n_4) \wedge \mathsf{spec}(n_2, n_4)$	
(3)	$\mathsf{commutes}(n_1)$	(2), $\wedge$**E**
(4)	$\mathsf{commutes}(n_1) \rightarrow \exists v_2, v_3, v_4 \bullet \mathsf{next}(n_1, v_2) \wedge \mathsf{spec}(n_1, v_3) \wedge \mathsf{next}(v_3, v_4) \wedge \mathsf{spec}(v_2, v_4)$	[A14], $\forall$**E**
(5)	$\exists v_2, v_3, v_4 \bullet \mathsf{next}(n_1, v_2) \wedge \mathsf{spec}(n_1, v_3) \wedge \mathsf{next}(v_3, v_4) \wedge \mathsf{spec}(v_2, v_4)$	(3), (4), $\rightarrow$**E**
(6)	$\mathsf{next}(n_1, n_5) \wedge \mathsf{spec}(n_1, n_6) \wedge \mathsf{next}(n_6, n_7) \wedge \mathsf{spec}(n_5, n_7)$	
(7)	$\mathsf{spec}(n_1, n_3) \wedge \mathsf{spec}(n_1, n_6) \rightarrow n_3 = n_6$	[A7], $\forall$**E**
(8)	$\mathsf{spec}(n_1, n_3)$	(2), $\wedge$**E**
(9)	$\mathsf{spec}(n_1, n_6)$	(6), $\wedge$**E**
(10)	$\mathsf{spec}(n_1, n_3) \wedge \mathsf{spec}(n_1, n_6)$	(8), (9), $\wedge$**I**
(11)	$n_3 = n_6$	(7), (10), $\rightarrow$**E**
(12)	$\mathsf{next}(n_3, n_4) \wedge \mathsf{next}(n_3, n_7) \rightarrow n_4 = n_7$	[A6], $\forall$**E**
(13)	$\mathsf{next}(n_3, n_4)$	(2), $\wedge$**E**

(14)	$\mathsf{next}(n_6, n_7)$	(6), $\wedge$**E**
(15)	$\mathsf{next}(n_3, n_7)$	(11), (14), $=$**E**
(16)	$\mathsf{next}(n_3, n_4) \wedge \mathsf{next}(n_3, n_7)$	(13), (15), $\wedge$**I**
(17)	$n_4 = n_7$	(12), (16), $\rightarrow$**E**
(18)	$\mathsf{spec}(n_2, n_4) \wedge \mathsf{spec}(n_5, n_4) \rightarrow n_2 = n_5$	[A8], $\forall$**E**
(19)	$\mathsf{spec}(n_2, n_4)$	(2), $\wedge$**E**
(20)	$\mathsf{spec}(n_5, n_7)$	(6), $\wedge$**E**
(21)	$\mathsf{spec}(n_5, n_4)$	(17), (20), $=$**E**
(22)	$\mathsf{spec}(n_2, n_4) \wedge \mathsf{spec}(n_5, n_4)$	(19), (21), $\wedge$**I**
(23)	$n_2 = n_5$	(18), (22), $\rightarrow$**E**
(24)	$\mathsf{next}(n_1, n_5)$	(6), $\wedge$**E**
(25)	$\mathsf{next}(n_1, n_2)$	(23), (24), $=$**E**
(26)	$\mathsf{next}(n_1, n_2)$	(5), (6), (25), $\exists$**E**
(27)	$\mathsf{next}(n_1, n_2)$	(1), (2), (26), $\exists$**E**
(28)	$(\exists\, v_1, v_2 \bullet \mathsf{commutes}(n_1) \wedge \mathsf{spec}(n_1, v_1) \wedge \mathsf{next}(v_1, v_2) \wedge \mathsf{spec}(n_2, v_2)) \rightarrow \mathsf{next}(n_1, n_2)$	(1), (27), $\rightarrow$**I**

A.5 Reachability

The compatibility constraints in this section use transitive closure. The proof steps marked "TC" combine several natural deduction steps. For the first four proofs these are fairly straightforward, from the definitions of p^+ and p^*, but the fifth proof takes much larger leaps, indicated by the ellipses.

A.5.1 No self loop

$\neg\, \mathsf{circ}(n_1) \wedge n_1 = n_2 \rhd \neg\, \mathsf{next}(n_1, n_2)$

(1)	$\boxed{\neg\,\mathsf{circ}(n_1) \wedge n_1 = n_2}$	
(2)	$n_1 = n_2$	(1), $\wedge\mathbf{E}$
(3)	$\boxed{\mathsf{next}(n_1, n_2)}$	
(4)	$\mathsf{next}(n_1, n_1)$	(2), (3), $=\mathbf{E}$
(5)	$\mathsf{next}^+(n_1, n_1)$	(4), TC
(6)	$\mathsf{next}^+(n_1, n_1) \rightarrow \mathsf{circ}(n_1)$	[A17], $\forall\mathbf{E}$
(7)	$\mathsf{circ}(n_1)$	(5), (6), $\rightarrow\mathbf{E}$
(8)	$\neg\,\mathsf{circ}(n_1)$	(1), $\wedge\mathbf{E}$
(9)	$\bot$	(7), (8), $\bot\mathbf{I}$
(10)	$\neg\,\mathsf{next}(n_1, n_2)$	(3), (9), $\neg\,\mathbf{I}$
(11)	$\neg\,\mathsf{circ}(n_1) \wedge n_1 = n_2 \rightarrow \neg\,\mathsf{next}(n_1, n_2)$	(1), (10), $\rightarrow\mathbf{I}$

A.5.2 No loop back

$\neg\,\mathsf{circ}(n_1) \wedge \mathsf{next}(n_1, n_2) \rhd \neg\,\mathsf{next}(n_2, n_1)$

(1)	$\boxed{\neg\,\mathsf{circ}(n_1) \wedge \mathsf{next}(n_1, n_2)}$	
(2)	$\mathsf{next}(n_1, n_2)$	(1), $\wedge\mathbf{E}$
(3)	$\boxed{\mathsf{next}(n_2, n_1)}$	
(4)	$\mathsf{next}^+(n_1)$	(2), (3), TC
(5)	$\mathsf{next}^+(n_1) \rightarrow \mathsf{circ}(n_1)$	[A17], $\forall\mathbf{E}$
(6)	$\mathsf{circ}(n_1)$	(4), (5), $\rightarrow\mathbf{E}$
(7)	$\neg\,\mathsf{circ}(n_1)$	(1), $\wedge\mathbf{E}$
(8)	$\bot$	(6), (7), $\bot\mathbf{I}$
(9)	$\neg\,\mathsf{next}(n_2, n_1)$	(3), (8), $\neg\,\mathbf{I}$
(10)	$\neg\,\mathsf{circ}(n_1) \wedge \mathsf{next}(n_1, n_2) \rightarrow \neg\,\mathsf{next}(n_2, n_1)$	(1), (9), $\rightarrow\mathbf{I}$

A.5.3 No loop to head

$\mathsf{Head_I}(n_1) \wedge \neg\, \mathsf{circ}(n_1) \wedge \mathsf{reach_I}(n_2) \rhd \neg\, \mathsf{next}(n_2, n_1)$

(1)	$\boxed{\mathsf{Head_I}(n_1) \wedge \neg\, \mathsf{circ}(n_1) \wedge \mathsf{reach_I}(n_2)}$	
(2)	$\mathsf{reach_I}(n_2)$	(1), $\wedge\mathbf{E}$
(3)	$\mathsf{reach_I}(n_2) \rightarrow \exists\, v_2 \bullet \mathsf{Head_I}(v_2) \wedge \mathsf{next}^*(v_2, n_2)$	[A18], $\forall\mathbf{E}$
(4)	$\exists\, v_2 \bullet \mathsf{Head_I}(v_2) \wedge \mathsf{next}^*(v_2, n_2)$	(2), (3), $\rightarrow\mathbf{E}$
(5)	$\boxed{\mathsf{Head_I}(n_3) \wedge \mathsf{next}^*(n_3, n_2)}$	
(6)	$\mathsf{Head_I}(n_1) \wedge \mathsf{Head_I}(n_3) \rightarrow n_1 = n_3$	[A1], $\forall\mathbf{E}$
(7)	$\mathsf{Head_I}(n_1)$	(1), $\wedge\mathbf{E}$
(8)	$\mathsf{Head_I}(n_3)$	(5), $\wedge\mathbf{E}$
(9)	$\mathsf{Head_I}(n_1) \wedge \mathsf{Head_I}(n_3)$	(7), (8), $\wedge\mathbf{I}$
(10)	$n_1 = n_3$	(6), (9), $\rightarrow\mathbf{E}$
(11)	$\mathsf{next}^*(n_3, n_2)$	(5), $\wedge\mathbf{E}$
(12)	$\mathsf{next}^*(n_1, n_2)$	(10), (11), $=\mathbf{E}$
(13)	$\boxed{\mathsf{next}(n_2, n_1)}$	
(14)	$\mathsf{next}^+(n_1, n_1)$	(12), (13), TC
(15)	$\mathsf{next}^+(n_1, n_1) \rightarrow \mathsf{circ}(n_1)$	[A17], $\forall\mathbf{E}$
(16)	$\mathsf{circ}(n_1)$	(14), (15), $\rightarrow\mathbf{E}$
(17)	$\neg\, \mathsf{circ}(n_1)$	(1), $\wedge\mathbf{E}$
(18)	$\bot$	(16), (17), $\bot\mathbf{I}$
(19)	$\neg\, \mathsf{next}(n_2, n_1)$	(13), (18), $\neg\,\mathbf{I}$
(20)	$\neg\, \mathsf{next}(n_2, n_1)$	(4), (5), (19), $\exists\mathbf{E}$
(21)	$\mathsf{Head_I}(n_1) \wedge \neg\, \mathsf{circ}(n_1) \wedge \mathsf{reach_I}(n_2) \rightarrow \neg\, \mathsf{next}(n_2, n_1)$	(1), (20), $\rightarrow\mathbf{I}$

A.5.4 Unreachable

$\mathsf{reach_l}(n_1) \wedge \neg\, \mathsf{reach_l}(n_2) \rhd \neg\, \mathsf{next}(n_1, n_2)$

(1)	$\boxed{\mathsf{reach_l}(n_1) \wedge \neg\, \mathsf{reach_l}(n_2)}$	
(2)	$\mathsf{reach_l}(n_1)$	(1), $\wedge\mathbf{E}$
(3)	$\mathsf{reach_l}(n_1) \rightarrow \exists\, v_2 \bullet \mathsf{Head_l}(v_2) \wedge \mathsf{next}^*(v_2, n_1)$	[A18], $\forall\mathbf{E}$
(4)	$\exists\, v_2 \bullet \mathsf{Head_l}(v_2) \wedge \mathsf{next}^*(v_2, n_1)$	(2), (3), $\rightarrow\mathbf{E}$
(5)	$\boxed{\mathsf{next}(n_1, n_2)}$	
(6)	$\exists\, v_2 \bullet \mathsf{Head_l}(v_2) \wedge \mathsf{next}^*(v_2, n_2)$	(4), (5), TC
(7)	$(\exists\, v_2 \bullet \mathsf{Head_l}(v_2) \wedge \mathsf{next}^*(v_2, n_2)) \rightarrow \mathsf{reach_l}(n_2)$	[A19], $\forall\mathbf{E}$
(8)	$\mathsf{reach_l}(n_2)$	(6), (7), $\rightarrow\mathbf{E}$
(9)	$\neg\, \mathsf{reach_l}(n_2)$	(1), $\wedge\mathbf{E}$
(10)	$\bot$	(8), (9), $\bot\mathbf{I}$
(11)	$\neg\, \mathsf{next}(n_1, n_2)$	(5), (10), $\neg\,\mathbf{I}$
(12)	$\mathsf{reach_l}(n_1) \wedge \neg\, \mathsf{reach_l}(n_2) \rightarrow \neg\, \mathsf{next}(n_1, n_2)$	(1), (11), $\rightarrow\mathbf{I}$

A.5.5 Chain

$\exists\, n_1 \bullet \mathsf{reach_l}(n_1) \wedge \mathsf{reach_l}(n_2) \wedge \neg\, \mathsf{circ}(n_2) \wedge$
$\mathsf{next}(n_1, n_2) \wedge \mathsf{reach_l}(n_3) \wedge \neg\, \mathsf{eq}(n_1, n_3) \rhd \neg\, \mathsf{next}(n_3, n_2)$

(1)	$\boxed{\exists\, v_1 \bullet \mathsf{reach_l}(v_1) \wedge \mathsf{reach_l}(n_2) \wedge \neg\, \mathsf{circ}(n_2) \wedge \mathsf{next}(v_1, n_2) \wedge \mathsf{reach_l}(n_3) \wedge v_1 \neq n_3}$	
(2)	$\boxed{\mathsf{reach_l}(n_1) \wedge \mathsf{reach_l}(n_2) \wedge \neg\, \mathsf{circ}(n_2) \wedge \mathsf{next}(n_1, n_2) \wedge \mathsf{reach_l}(n_3) \wedge n_1 \neq n_3}$	
(3)	$\mathsf{reach_l}(n_1)$	(2), $\wedge\mathbf{E}$
(4)	$\mathsf{reach_l}(n_2)$	(2), $\wedge\mathbf{E}$

(5)	$\mathsf{reach}_{\mathsf{l}}(n_3)$	(2), $\wedge\mathbf{E}$
(6)	$\mathsf{reach}_{\mathsf{l}}(n_1) \rightarrow \exists\, v_2 \bullet \mathsf{Head}_{\mathsf{l}}(v_2) \wedge \mathsf{next}^*(v_2, n_1)$	[A18], $\forall\mathbf{E}$
(7)	$\mathsf{reach}_{\mathsf{l}}(n_2) \rightarrow \exists\, v_2 \bullet \mathsf{Head}_{\mathsf{l}}(v_2) \wedge \mathsf{next}^*(v_2, n_2)$	[A18], $\forall\mathbf{E}$
(8)	$\mathsf{reach}_{\mathsf{l}}(n_3) \rightarrow \exists\, v_2 \bullet \mathsf{Head}_{\mathsf{l}}(v_2) \wedge \mathsf{next}^*(v_2, n_3)$	[A18], $\forall\mathbf{E}$
(9)	$\exists\, v_2 \bullet \mathsf{Head}_{\mathsf{l}}(v_2) \wedge \mathsf{next}^*(v_2, n_1)$	(3), (6), $\rightarrow\mathbf{E}$
(10)	$\exists\, v_2 \bullet \mathsf{Head}_{\mathsf{l}}(v_2) \wedge \mathsf{next}^*(v_2, n_2)$	(4), (7), $\rightarrow\mathbf{E}$
(11)	$\exists\, v_2 \bullet \mathsf{Head}_{\mathsf{l}}(v_2) \wedge \mathsf{next}^*(v_2, n_3)$	(5), (8), $\rightarrow\mathbf{E}$
(12)	$\mathsf{next}(n_1, n_2)$	(2), $\wedge\mathbf{E}$
(13)	$n_1 \neq n_3$	(2), $\wedge\mathbf{E}$
	...	
(14)	$\mathsf{next}^+(n_3, n_1) \vee \mathsf{next}^*(n_2, n_3)$	[A1], [A6], (9)–(13), TC
(15)	$\boxed{\mathsf{next}(n_3, n_2)}$	
(16)	$\neg\, \mathsf{circ}(n_2)$	(2), $\wedge\mathbf{E}$
(17)	$\mathsf{next}^+(n_2, n_2) \rightarrow \mathsf{circ}(n_2)$	[A17], $\forall\mathbf{E}$
(18)	$\boxed{\mathsf{next}^+(n_3, n_1)}$	
	...	
(19)	$\mathsf{next}^*(n_2, n_1)$	(15), (18), [A6], TC
(20)	$\mathsf{next}^+(n_2, n_2)$	(12), (19), TC
(21)	$\mathsf{circ}(n_2)$	(17), (20), $\rightarrow\mathbf{E}$
(22)	$\bot$	(16), (21), $\bot\mathbf{I}$
(23)	$\boxed{\mathsf{next}^*(n_2, n_3)}$	
(24)	$\mathsf{next}^+(n_2, n_2)$	(15), (23), TC
(25)	$\mathsf{circ}(n_2)$	(17), (24), $\rightarrow\mathbf{E}$
(26)	$\bot$	(16), (25), $\bot\mathbf{I}$

(27)	$\bot$	(14), (18), (22), (23), (26), $\vee$**E**
(28)	$\neg\, \mathsf{next}(n_3, n_2)$	(15), (27), $\neg$ **I**
(29)	$\neg\, \mathsf{next}(n_3, n_2)$	(1), (2), (28), $\exists$**E**
(30)	$\exists\, v_1 \bullet \mathsf{reach}_{\mathsf{I}}(v_1) \wedge \mathsf{reach}_{\mathsf{I}}(n_2) \wedge \neg\, \mathsf{circ}(n_2) \wedge$ $\mathsf{next}(v_1, n_2) \wedge \mathsf{reach}_{\mathsf{I}}(n_3) \wedge v_1 \neq n_3$ $\rightarrow \neg\, \mathsf{next}(n_3, n_2)$	(1), (29), $\rightarrow$**I**

Bibliography

Jean-Raymond Abrial and Dominique Cansell. Formal construction of a non-blocking concurrent queue algorithm. *Journal of Universal Computer Science*, 11(5):744–770, 2005. (p. 2).

Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. (p. 99).

Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996. (p. 11).

Juan Alemany and Edward W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC), 10–12 August 1992, Vancouver, British Columbia, Canada*, pages 125–134. ACM Press, 1992. (p. 2).

Rajeev Alur, Kenneth L. McMillan, and Doron A. Peled. Model-checking of correctness conditions for concurrent objects. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS), 27–30 July 1996, New Brunswick, NJ, USA*, pages 219–228. IEEE Computer Society Press, 1996. (p. 92).

Rajeev Alur, Kenneth L. McMillan, and Doron A. Peled. Model-checking of correctness conditions for concurrent objects. *Information and Computation*, 160(1–2):167–188, 2000. (p. 92).

Daphna Amit, Noam Rinetzky, Thomas Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV), 3–7 July 2007, Berlin, Germany*, volume 4590 of *Lecture Notes in Computer Science*, pages 477–490. Springer-Verlag, 2007. (pp. 5, 159, 185, 195, 196, 237).

James H. Anderson and Mark Moir. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, 1999. (p. 18).

Krzysztof R. Apt and Dexter C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6): 307–309, 1986. (p. 4).

Uwe Aßmann and Markus Weinhardt. Interprocedural heap analysis for parallelizing imperative programs. In Wolfgang K. Giloi, Stefan Jähnichen, and Bruce Shriver, editors, *Proceedings of the 1st Conference on Programming Models for Massively Parallel Computers, 20–23 September 1993, Berlin, Germany*, pages 74–82. IEEE Computer Society Press, 1993. (p. 49).

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008. (p. 27).

Michael Balser, Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel. KIV 3.0 for provably correct systems. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *Proceedings of the International Workshop on Current Trends in Applied Formal Methods (FM-Trends), 7–9 October 1998, Boppard, Germany*, volume 1641 of *Lecture Notes in Computer Science*, pages 330–337. Springer-Verlag, 1998. (p. 95).

Greg Barnes. A method for implementing lock-free shared data structures. In Lawrence Snyder, editor, *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), 30 June – 2 July 1993, Velen, Germany*, pages 261–270. ACM Press, 1993. (p. 2).

Sharon Barner and Orna Grumberg. Combining symmetry reduction and under-approximation for symbolic model checking. *Formal Methods in System Design*, 27(1–2):29–66, 2005. (p. 42).

Vincent Beaudenon, Emmanuelle Encrenaz, and Sami Taktak. Data decision diagrams for Promela systems analysis. *International Journal on Software Tools for Technology Transfer*, 12:337–352, 2010. (p. 33).

Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In John White, Richard Lipton, and Patricia C.

Goldberg, editors, *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 26–28 January 1981, Williamsburg, VA, USA*, pages 164–176. ACM Press, 1981. (p. 32).

Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983. (p. 32).

Beatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer-Verlag, 2001. (pp. 27, 39).

Josh Berdine, Tal Lev-Ami, Roman Manevich, Ganesan Ramalingam, and Mooly Sagiv. Thread quantification for concurrent shape analysis. In Aarti Gupta and Sharad Malik, editors, *Proceedings of the 20th International Conference on Computer Aided Verification (CAV), 7–14 July 2008, Princeton, NJ, USA*, volume 5123 of *Lecture Notes in Computer Science*, pages 399–413. Springer-Verlag, 2008. (pp. 5, 196, 197, 231, 237, 238).

Orna Bernholtz, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. In David L. Dill, editor, *Proceedings of the 6th International Conference on Computer Aided Verification (CAV), 21–23 June 1994, Stanford, CA, USA*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155. Springer-Verlag, 1994. (p. 34).

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In Mary Jane Irwin, editor, *Proceedings of the 36th Conference on Design Automation (DAC), 21–25 June 1999, New Orleans, LA, USA*, pages 317–320. ACM Press, 1999a. (p. 39).

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), 22–28 March 1999, Amsterdam, The Netherlands*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999b. (p. 39).

Igor Bogudlov, Tal Lev-Ami, Thomas Reps, and Mooly Sagiv. Revamping TVLA: Making parametric shape analysis competitive. In Werner

Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV), 3–7 July 2007, Berlin, Germany*, volume 4590 of *Lecture Notes in Computer Science*, pages 221–225. Springer-Verlag, 2007a. (pp. 47, 80, 182).

Igor Bogudlov, Tal Lev-Ami, Thomas Reps, and Mooly Sagiv. Revamping TVLA: Making parametric shape analysis competitive. Technical Report TR-2007-01-01, Tel Aviv University, 2007b. (p. 80).

Dragan Bošnački. A light-weight algorithm for model checking with symmetry reduction and weak fairness. In Thomas Ball and Sriram K. Rajamani, editors, *Proceedings of the 10th International SPIN Workshop, 9–10 May 2003, Portland, OR, USA*, volume 2648 of *Lecture Notes in Computer Science*, pages 89–103. Springer-Verlag, 2003. (p. 135).

Dragan Bošnački, Dennis Dams, and Leszek Holenderski. Symmetric Spin. In Klaus Havelund, John Penix, and Willem Visser, editors, *Proceedings of the 7th International SPIN Workshop, 30 August – 1 September 2000, Stanford, CA, USA*, volume 1885 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2000. (pp. 42, 46).

Dragan Bošnački, Dennis Dams, and Leszek Holenderski. Symmetric Spin. *International Journal on Software Tools for Technology Transfer*, 4(1): 92–106, 2002. (pp. 42, 46).

Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. (p. 37).

Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992. (p. 37).

J. Richard Büchi. On a decision method in restricted second order arithmetic. In Ernest Nagel, Patrick Suppes, and Alfred Tarski, editors, *Proceedings of the 1st International Congress for Logic, Methodology and Philosophy of Science, 24 August – 2 September 1960, Stanford, CA, USA*, pages 1–11. Stanford University Press, 1962. (p. 34).

Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer*

Science (LICS), 4–7 June 1990, Philadelphia, PA, USA, pages 428–439. IEEE Computer Society Press, 1990. (p. 36).

Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992. (p. 36).

Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Bounded model checking of concurrent data types on relaxed memory models: A case study. In Thomas Ball and Robert B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV), 16–21 August 2006, Seattle, WA, USA*, volume 4144 of *Lecture Notes in Computer Science*, pages 489–502. Springer-Verlag, 2006. (p. 3).

Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 11–13 June 2007, San Diego, CA, USA*, pages 12–21. ACM Press, 2007. (p. 3).

David Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In Bernard N. Fischer, editor, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 20–22 June 1990, White Plains, NY, USA*, pages 296–310. ACM Press, 1990. (pp. 49, 50).

Edmund M. Clarke and I. Anca Draghicescu. Expressibility results for linear-time and branching-time logics. In Jaco W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Proceedings of the REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, 30 May – 3 June 1988, Noordwijkerhout, The Netherlands*, volume 354 of *Lecture Notes in Computer Science*, pages 428–437. Springer-Verlag, 1988. (p. 32).

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter C. Kozen, editor, *Proceedings of the Workshop on Logic of Programs, 1 May 1981, Yorktown Heights, NY, USA*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981. (p. 27).

Edmund M. Clarke and Orna Grumberg. Avoiding the state explosion problem in temporal logic model checking. In Fred B. Schneider, editor, *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC), 10–12 August 1987, Vancouver, British Columbia, Canada*, pages 294–303. ACM Press, 1987. (p. 39).

Edmund M. Clarke and Bernd-Holger Schlingloff. Model checking. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, pages 1367–1447. Elsevier, 2001. (p. 27).

Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: A practical approach. In John R. Wright, Larry Landweber, Alan Demers, and Tim Teitelbaum, editors, *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), 24–26 January 1983, Austin, TX, USA*, pages 117–126. ACM Press, 1983. (p. 33).

Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2): 244–263, 1986. (p. 33).

Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In Costas Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification (CAV), 28 June – 1 July 1993, Elounda, Greece*, volume 697 of *Lecture Notes in Computer Science*, pages 450–462. Springer-Verlag, 1993. (p. 42).

Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994. (p. 43).

Edmund M. Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996. (p. 42).

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999. (pp. 27, 34, 36, 41, 42).

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen

Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV), 15–19 July 2000, Chicago, IL, USA*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, 2000. (p. 44).

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the Association for Computing Machinery*, 50(5): 752–794, September 2003. (p. 44).

Robert Colvin and Brijesh Dongol. Verifying lock-freedom using well-founded orders. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors, *Proceedings of the 4th International Colloquium on Theoretical Aspects of Computing (ICTAC), 26–28 September 2007, Macau, China*, volume 4711 of *Lecture Notes in Computer Science*, pages 124–138. Springer-Verlag, 2007. (p. 2).

Robert Colvin and Brijesh Dongol. A general technique for proving lock-freedom. *Science of Computer Programming*, 74(3):143–165, 2009. (pp. 2, 116).

Robert Colvin and Lindsay Groves. Formal verification of an array-based nonblocking queue. In Carlo Ghezzi, Yuxi Fu, Shaoying Liu, and Jim Woodcock, editors, *Proceedings of the 10th International Conference on Engineering of Complex Computer Systems (ICECCS), 16–20 June 2005, Shanghai, China*, pages 507–516. IEEE Computer Society Press, 2005. (pp. 2, 95, 122).

Robert Colvin and Lindsay Groves. A scalable lock-free stack algorithm and its verification. In *Proceedings of the 5th International Conference on Software Engineering and Formal Methods (SEFM), 10–14 September 2007, London, UK*, pages 339–348. IEEE Computer Society Press, 2007. (p. 2).

Robert Colvin, Simon Doherty, and Lindsay Groves. Verifying concurrent data structures by simulation. In John Derrick and Eerke A. Boiten, editors, *Proceedings of the BCS FACS Refinement Workshop (REFINE), 12 April 2005, Guildford, England, UK*, volume 137.2 of *Electronic Notes in Theoretical Computer Science*, pages 93–110. Elsevier, 2005. (pp. 2, 21, 95).

Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a lazy concurrent list-based set algorithm. In Thomas Ball

and Robert B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV), 16–21 August 2006, Seattle, WA, USA*, volume 4144 of *Lecture Notes in Computer Science*, pages 475–488. Springer-Verlag, 2006. (pp. 2, 3, 92, 95, 135).

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), 17–19 January 1977, Los Angeles, CA, USA*, pages 238–252. ACM Press, 1977. (pp. 3, 44).

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), 29–31 January 1979, San Antonio, TX, USA*, pages 269–282. ACM Press, 1979. (pp. 3, 44, 66).

Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997. (p. 44).

Leonardo de Moura, Sam Owre, and Natarajan Shankar. The SAL language manual. Technical Report SRI-CSL-01-02, Computer Science Laboratory, SRI International, 2001. Revised August 2003. (p. 47).

Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE), 27–30 July 2002, Copenhagen, Denmark*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455. Springer-Verlag, 2002. (p. 39).

Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, Natarajan Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV), 13–17 July 2004, Boston, MA, USA*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500. Springer-Verlag, 2004. (p. 47).

Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV), 3–7 July 2007, Berlin, Germany*, volume 4590 of *Lecture Notes in Computer Science*, pages 20–36. Springer-Verlag, 2007. (p. 39).

Rocco De Nicola and Frits W. Vaandrager. Action versus state based logics for transition systems. In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes. Proceedings of the LITP Spring School on Theoretical Computer Science, 23–27 April 1990, La Roche Posay, France*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer-Verlag, 1990. (p. 28).

Claudio Demartini, Radu Iosif, and Riccardo Sisto. dSpin: A dynamic extension of Spin. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *Proceedings of the 5th and 6th International SPIN Workshops, 5 July 1999, Trento, Italy and 21–24 September 1999, Tolouse, France*, volume 1680 of *Lecture Notes in Computer Science*, pages 261–276. Springer-Verlag, 1999. (p. 46).

Stéphane Demri, François Laroussinie, and Philippe Schnoebelen. A parametric analysis of the state explosion problem in model checking. In Helmut Alt and Afonso Ferreira, editors, *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS), 14–16 March 2002, Antibes Juan-les-Pins, France*, volume 2285 of *Lecture Notes in Computer Science*, pages 620–631. Springer-Verlag, 2002. (p. 40).

Stéphane Demri, François Laroussinie, and Philippe Schnoebelen. A parametric analysis of the state-explosion problem in model checking. *Journal of Computer and System Sciences*, 72(4):547–575, 2006. (p. 40).

John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Proving linearizability via non-atomic refinement. In Jim Davies and Jeremy Gibbons, editors, *Proceedings of the 6th International Conference on Integrated Formal Methods, 2–5 July 2007, Oxford, UK*, volume 4591 of *Lecture Notes in Computer Science*, pages 195–214. Springer-Verlag, 2007. (pp. 2, 95).

John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Mechanizing a correctness proof for a lock-free concurrent stack. In Gilles Barthe and Frank S. de Boer, editors, *Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distribution Systems*

(FMOODS), 4–6 June 2008, Oslo, Norway, volume 5051 of *Lecture Notes in Computer Science*, pages 78–95. Springer-Verlag, 2008. (pp. 2, 95).

John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Transactions on Programming Languages and Systems*, 33(1), 2011a. (p. 95).

John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Verifying linearisability with potential linearisation points. In Michael J. Butler and Wolfram Schulte, editors, *Proceedings of the 17th International Symposium on Formal Methods (FM), 20–24 June 2011, Limerick, Ireland*, volume 6664 of *Lecture Notes in Computer Science*, pages 323–337. Springer-Verlag, 2011b. (p. 95).

David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, and Guy L. Steele, Jr. Even better DCAS-based concurrent deques. In Maurice P. Herlihy, editor, *Proceedings of the 14th International Conference on Distributed Computing (DISC), 4–6 October 2000, Toledo, Spain*, volume 1914 of *Lecture Notes in Computer Science*, pages 59–73. Springer-Verlag, 2000. (pp. 2, 122, 133, 197).

Edsger W. Dijkstra. Cooperating sequential processes. `http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF`, September 1965. (p. 13).

Simon Doherty. Modelling and verifying non-blocking algorithms that use dynamically allocated memory. M.Sc. thesis, Victoria University of Wellington, 2003. (pp. 2, 123).

Simon Doherty. *The Design and Verification of Dynamic-Sized Nonblocking Data Structures*. Ph.D. thesis, Victoria University of Wellington, 2010. (p. 2).

Simon Doherty and Mark Moir. Nonblocking algorithms and backward simulation. In Idit Keidar, editor, *Proceedings of the 23rd International Symposium on Distributed Computing (DISC), 23–25 September 2009, Elche, Spain*, volume 5805 of *Lecture Notes in Computer Science*, pages 274–288. Springer-Verlag, 2009. (p. 2).

Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir N. Shavit, and Guy L.

Steele, Jr. DCAS is not a silver bullet for nonblocking algorithm design. In Phillip B. Gibbons and Micah Adler, editors, *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 27–30 June 2004, Barcelona, Spain*, pages 216–224. ACM Press, 2004a. (p. 123).

Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In David de Frutos-Escrig and Manuel Núñez, editors, *Proceedings of the 24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE), 27–30 September 2004, Madrid, Spain*, volume 3235 of *Lecture Notes in Computer Science*, pages 97–114. Springer-Verlag, 2004b. (pp. 2, 25, 95, 99, 124).

Simon Doherty, Maurice P. Herlihy, Victor Luchangco, and Mark Moir. Bringing practical lock-free synchronization to 64-bit applications. In Soma Chaudhuri and Shay Kutten, editors, *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC), 25–28 July 2004, St John's, Newfoundland, Canada*, pages 31–39. ACM Press, 2004c. (p. 18).

Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a SAT solver. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 11th European Software Engineering Conference (ESEC) and the 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 3–7 September 2007, Dubrovnik, Croatia*, pages 195–204. ACM Press, 2007. (p. 3).

Alastair F. Donaldson and Alice Miller. Automatic symmetry detection for model checking using computational group theory. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *Proceedings of the International Symposium of Formal Methods Europe (FME), 18–22 July 2005, Newcastle upon Tyne, England, UK*, volume 3582 of *Lecture Notes in Computer Science*, pages 481–496. Springer-Verlag, 2005. (p. 42).

Alastair F. Donaldson and Alice Miller. A computational group theoretic symmetry reduction package for the Spin model checker. In Michael Johnson and Varmo Vene, editors, *Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology (AMAST), 5–8 July 2006, Kuressaare, Estonia*, volume 4019 of *Lecture Notes in Computer Science*, pages 374–380. Springer-Verlag, 2006. (pp. 42, 47, 129).

Alastair F. Donaldson, Alice Miller, and Muffy Calder. Finding symmetry in models of concurrent systems by static channel diagram analysis. In Michael Huth, editor, *Proceedings of the 4th International Workshop on Automated Verification of Critical Systems (AVoCS), 4 September 2004, London, England, UK*, volume 128.6 of *Electronic Notes in Theoretical Computer Science*, pages 161–177. Elsevier, 2005a. (p. 42).

Alastair F. Donaldson, Alice Miller, and Muffy Calder. Spin-to-Grape: A tool for analysing symmetry in Promela models. In Irek Ulidowski, editor, *Proceedings of the 6th AMAST Workshop on Real-Time Systems (ARTS), 12 July 2004, Stirling, Scotland, UK*, volume 139.1 of *Electronic Notes in Theoretical Computer Science*, pages 3–23. Elsevier, 2005b. (p. 42).

Brijesh Dongol. Formalising progress properties of non-blocking programs. In Zhiming Liu and Jifeng He, editors, *Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM), 1–3 November 2006, Macao, China*, volume 4260 of *Lecture Notes in Computer Science*, pages 284–303. Springer-Verlag, 2006. (pp. 115, 116).

Brijesh Dongol. *Progress-Based Verification and Derivation of Concurrent Programs*. Ph.D. thesis, University of Queensland, 2009. (pp. 115, 116).

Brijesh Dongol and Doug Goldson. Extending the theory of Owicki and Gries with a logic of progress. *Logical Methods in Computer Science*, 2(1), 2006. (p. 115).

Brijesh Dongol and Ian J. Hayes. Enforcing safety and progress properties: An approach to concurrent program derivation. In *Proceedings of the 20th Australian Software Engineering Conference (ASWEC), 14–17 April 2009, Gold Coast, Australia*, pages 3–12. IEEE Computer Society Press, 2009. (p. 2).

Brijesh Dongol and Arjan J. Mooij. Streamlining progress-based derivations of concurrent programs. *Formal Aspects of Computing*, 20(2):141–160, 2008. (p. 2).

Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999. (p. 40).

E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier, 1990. (p. 29).

E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In Jaco W. de Bakker and Jan van Leeuwen, editors, *Proceedings of the 7th International Colloquium on Automata, Languages and Programming (ICALP), 14–18 July 1980, Noordweijkerhout, The Netherlands*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer-Verlag, 1980. (p. 32).

E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" revisited: On branching versus linear time. In John R. Wright, Larry Landweber, Alan Demers, and Tim Teitelbaum, editors, *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), 24–26 January 1983, Austin, TX, USA*, pages 127–140. ACM Press, 1983. (p. 32).

E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" revisited: On branching versus linear time temporal logic. *Journal of the Association for Computing Machinery*, 33(1):151–178, 1986. (p. 32).

E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: Branching time strikes back. In Mary S. Van Deusen, Zvi Galil, and Brian K. Reid, editors, *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), 14–16 January 1985, New Orleans, LA, USA*, pages 84–96. ACM Press, 1985. (p. 32).

E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3): 275–306, 1987. (p. 32).

E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. In Costas Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification (CAV), 28 June – 1 July 1993, Elounda, Greece*, volume 697 of *Lecture Notes in Computer Science*, pages 463–478. Springer-Verlag, 1993. (p. 42).

E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996. (p. 42).

E. Allen Emerson and Richard J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In Laurence Pierre and Thomas Kropf, editors, *Proceedings of the 10th IFIP*

WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME), 27–29 September 1999, Bad Herrenalb, Germany, volume 1703 of *Lecture Notes in Computer Science*, pages 142–156. Springer-Verlag, 1999. (p. 42).

E. Allen Emerson and Thomas Wahl. On combining symmetry reduction and symbolic representation for efficient model checking. In Daniel Geist and Enrico Tronci, editors, *Proceedings of the 12th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME), 21–24 October 2003, L'Aquila, Italy*, volume 2860 of *Lecture Notes in Computer Science*, pages 216–230. Springer-Verlag, 2003. (p. 42).

Keir Fraser. *Practical Lock-freedom*. Ph.D. thesis, University of Cambridge, 2003. Also available as Fraser [2004]. (p. 3).

Keir Fraser. Practical lock-freedom. Technical Report 579, Computer Laboratory, University of Cambridge, 2004. (p. 272).

Keir Fraser and Timothy L. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2), 2007. (p. 3).

Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects, Volume 1*. Clarendon Press, 1994. (p. 29).

Hui Gao and Wim H. Hesselink. A formal reduction for lock-free parallel algorithms. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV), 13–17 July 2004, Boston, MA, USA*, volume 3114 of *Lecture Notes in Computer Science*, pages 44–56. Springer-Verlag, 2004. (pp. 2, 110).

Hui Gao and Wim H. Hesselink. A general lock-free algorithm using compare-and-swap. *Information and Computation*, 205(2):225–241, 2007. (p. 116).

Hui Gao, Jan Friso Groote, and Wim H. Hesselink. Almost wait-free resizable hashtable. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS), 26–30 April 2004, Santa Fe, NM, USA*. IEEE Computer Society Press, 2004. (p. 2).

Hui Gao, Jan Friso Groote, and Wim H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Computing*, 18(1):21–42, 2005. (p. 2).

Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV), 18–22 July 2001, Paris, France*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer-Verlag, 2001. (p. 36).

Rob Gerth, Doron A. Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In Piotr Dembinski and Marek Sredniawa, editors, *Proceedings of the 15th IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification (PSTV), 13–16 June 1995, Warsaw, Poland*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman and Hall, 1995. (p. 36).

Dimitra Giannakopoulou and Flavio Lerda. From states to transitions: Improving translation of LTL formulae to Büchi automata. In Doron A. Peled and Moshe Y. Vardi, editors, *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE), 11–14 November 2002, Houston, TX, USA*, volume 2529 of *Lecture Notes in Computer Science*, pages 308–326. Springer-Verlag, 2002. (p. 36).

Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. In *Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN), 7–9 December 2005, Las Vegas, NV, USA*, pages 202–207. IEEE Computer Society Press, 2005. (p. 19).

Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1173–1187, 2009. (p. 19).

Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996. (p. 41).

Robert Goldblatt. *Logics of Time and Computation*. Number 7 in CSLI Lecture Notes. CSLI Publications, Stanford University, 2nd edition, 1992. (pp. 28, 29).

Robert Goldblatt. Mathematical modal logic: A view of its evolution. *Journal of Applied Logic*, 1(5–6):309–392, 2003. (p. 29).

Robert Goldblatt. Mathematical modal logic: A view of its evolution. In Dov M. Gabbay and John Woods, editors, *Handbook of the History of Logic, Volume 7: Logic and the Modalities in the Twentieth Century*, pages 1–98. Elsevier, 2006. (p. 29).

Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV), 22–25 June 1997, Haifa, Israel*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, 1997. (pp. 43, 76).

Michael Greenwald and David R. Cheriton. The synergy between non-blocking synchronization and system design. In Karin Petersen and Willy Zwaenepoel, editors, *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation (OSDI), 29 October – 1 November 1996, Seattle, WA, USA*, pages 123–136. ACM Press, 1996. (p. 1).

Lindsay Groves. Trace-based derivation of a lock-free queue algorithm. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proceedings of the BCS-FACS Refinement Workshop (REFINE), 2 July 2007, Oxford, UK*, volume 201 of *Electronic Notes in Theoretical Computer Science*, pages 69–98. Elsevier, 2008a. (p. 2).

Lindsay Groves. Verifying Michael and Scott's lock-free queue algorithm using trace reduction. In James Harland and Prabhu Manyem, editors, *Proceedings of Computing: The Australasian Theory Symposium (CATS), 22–25 January 2008, Wollongong, Australia*, volume 77 of *Conferences in Research and Practice in Information Technology*, pages 133–142. Australian Computer Society, 2008b. (p. 2).

Lindsay Groves and Robert Colvin. Derivation of a scalable lock-free stack algorithm. In Bernhard K. Aichernig, Eerke A. Boiten, John Derrick, and

Lindsay Groves, editors, *Proceedings of the International Refinement Workshop (REFINE), 31 October 2006, Macau*, volume 187 of *Electronic Notes in Theoretical Computer Science*, pages 55–74. Elsevier, 2006. (p. 2).

Orna Grumberg. Abstractions and reductions in model checking. In Helmut Schwichtenberg and Ralf Steinbrüggen, editors, *Proceedings of the NATO Advanced Study Institute on Proof and System-Reliability, 24 July – 5 August 2001, Marktoberdorf, Germany*, volume 62 of *NATO Science Series II*, pages 289–322. Springer-Verlag, 2002a. (p. 44).

Orna Grumberg. Different directions in parallel and distributed model checking. In Luboš Brim and Orna Grumberg, editors, *Proceedings of the 1st International Workshop on Parallel and Distributed Model Checking (PDMC), 19 August 2002, Brno, Czech Republic*, volume 68.4 of *Electronic Notes in Theoretical Computer Science*, page 485. Elsevier, 2002b. (p. 3).

Orna Grumberg and Helmut Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, 2008. Springer-Verlag. (pp. 27, 291).

Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In Jennifer L. Welch, editor, *Proceedings of the 15th International Conference on Distributed Computing (DISC), 3–5 October 2001, Lisbon, Portugal*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer-Verlag, 2001. (pp. 3, 135).

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word Compare-and-Swap operation. In Dahlia Malkhi, editor, *Proceedings of the 16th International Conference on Distributed Computing (DISC), 28–30 October 2002, Toulouse, France*, volume 2508 of *Lecture Notes in Computer Science*, pages 265–279. Springer-Verlag, 2002. (p. 102).

Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In Bernard Robinet and Reinhard Wilhelm, editors, *Proceedings of the European Symposium on Programming (ESOP), 17–19 March 1986, Saarbrücken, Federal Republic of Germany*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag, 1986. (p. 94).

Steve Heller, Maurice P. Herlihy, Victor Luchangco, Mark Moir, Nir N. Shavit, and William N. Scherer, III. A lazy concurrent list-based set algorithm. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Proceedings of the 9th International Conference on Principles of*

Distributed Systems (OPODIS), 12–14 December 2005, Pisa, Italy, volume 3974 of *Lecture Notes in Computer Science*, pages 3–16. Springer-Verlag, 2005. (pp. 92, 197).

Steve Heller, Maurice P. Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, III, and Nir N. Shavit. A lazy concurrent list-based set algorithm. *Parallel Processing Letters*, 17(4):411–424, 2007. (pp. 92, 197).

Danny Hendler, Nir N. Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In Phillip B. Gibbons and Micah Adler, editors, *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 27–30 June 2004, Barcelona, Spain*, pages 206–215. ACM Press, 2004. (p. 102).

Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, 1991. (pp. 15, 116).

Maurice P. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993. (p. 2).

Maurice P. Herlihy and J. Eliot B. Moss. Lock-free garbage collection for multiprocessors. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), 21–24 July 1991, Hilton Head, CA, USA*, pages 229–236. ACM Press, 1991. (p. 19).

Maurice P. Herlihy and J. Eliot B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3 (3):304–311, 1992. (p. 19).

Maurice P. Herlihy and Nir N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008. (p. 14).

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 (POPL), 21–23 January 1987, Munich, West Germany*, pages 13–26. ACM Press, 1987. (p. 11).

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, July 1990. (pp. 1, 11).

Maurice P. Herlihy, Victor Luchangco, Paul A. Martin, and Mark Moir. Dynamic-sized lock-free data structures. In Aleta Ricciardi, editor, *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing (PODC), 21–24 July 2002, Monteray, CA, USA*, page 131. ACM Press, 2002a. (p. 19).

Maurice P. Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In Dahlia Malkhi, editor, *Proceedings of the 16th International Conference on Distributed Computing (DISC), 28–30 October 2002, Toulouse, France*, volume 2508 of *Lecture Notes in Computer Science*, pages 339–353. Springer-Verlag, 2002b. (p. 19).

Maurice P. Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In Jack Stankovic, Wei Zhao, Philip McKinley, and Sol Shatz, editors, *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS), 19–22 May 2003, Providence, RI, USA*, pages 522–529. IEEE Computer Society Press, 2003. (p. 16).

Maurice P. Herlihy, Victor Luchangco, Paul A. Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems*, 23(2):146–196, 2005. (p. 19).

Wim H. Hesselink. A criterion for atomicity revisited. *Acta Informatica*, 44 (2):123–151, 2007. (p. 95).

Gerard J. Holzmann. On limits and possibilities of automated protocol analysis. In Harry Rudin and Colin H. West, editors, *Proceedings of the 7th IFIP WG6.1 International Conference on Protocol Specification, Testing and Verification (PSTV), 5–8 May 1987, Zürich, Switzerland*, pages 339–344. North-Holland, 1987. (p. 40).

Gerard J. Holzmann. An analysis of bit-state hashing. In Piotr Dembinski and Marek Sredniawa, editors, *Proceedings of the 15th IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification (PSTV), 13–16 June 1995, Warsaw, Poland*, volume 38 of *IFIP Conference Proceedings*, pages 301–314. Chapman and Hall, 1995. (p. 40).

Gerard J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. (p. 46).

Gerard J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307, 1998. (p. 40).

Gerard J. Holzmann. *The Spin Model Checker: Primer and reference manual*. Addison-Wesley, 2004. (pp. 27, 33, 46, 127, 130).

Gerard J. Holzmann and Dragan Bošnački. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007. (p. 46).

Gerard J. Holzmann and Doron A. Peled. An improvement in formal verification. In Dieter Hogrefe and Stefan Leue, editors, *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques (FORTE), 4–7 October 1994, Berne, Switzerland*, volume 6 of *IFIP Conference Proceedings*, pages 197–211. Chapman and Hall, 1995. (p. 41).

Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Tackling large verification problems with the Swarm tool. In Klaus Havelund, Rupak Majumdar, and Jens Palsberg, editors, *Proceedings of the 15th SPIN Workshop on Model Checking Software, 10–12 August 2008, Los Angeles, CA, USA*, volume 5156 of *Lecture Notes in Computer Science*, pages 134–143. Springer-Verlag, 2008. (p. 36).

Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 21–23 June 1989, Portland, OR,USA*, pages 28–40. ACM Press, 1989. (p. 49).

Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. In Debra J. Richardson, Martin Feather, and Michael Goedicke, editors, *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE), 26–29 November 2001, San Diego, CA, USA*, pages 254–261. IEEE Computer Society Press, 2001. (pp. 42, 47).

Radu Iosif. Symmetry reduction criteria for software model checking. In Dragan Bošnački and Stefan Leue, editors, *Proceedings of the 9th International SPIN Workshop, 11–13 April 2002, Grenoble, France*, volume 2318 of *Lecture Notes in Computer Science*, pages 22–41. Springer-Verlag, 2002. (pp. 42, 47).

Radu Iosif. Symmetry reductions for model checking of concurrent dynamic software. *International Journal on Software Tools for Technology Transfer*, 6(4):302–319, 2004. (pp. 42, 47).

Radu Iosif and Riccardo Sisto. Using garbage collection in model checking. In Klaus Havelund, John Penix, and Willem Visser, editors, *Proceedings of the 7th International SPIN Workshop, 30 August – 1 September 2000, Stanford, CA, USA*, volume 1885 of *Lecture Notes in Computer Science*, pages 20–33. Springer-Verlag, 2000. (p. 47).

C. Norris Ip and David L. Dill. Better verification through symmetry. In David Agnew, Luc J. M. Claesen, and Raul Camposano, editors, *Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications (CHDL), 26–28 April 1993, Ottawa, Ontario, Canada*, volume A-32 of *IFIP Transactions*, pages 97–111. North-Holland, 1993. (p. 42).

C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996. (p. 42).

Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2nd edition, 2012. (pp. 3, 123).

Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, 1996. (p. 3).

Prasad Jayanti and Srdjan Petrovic. Efficient and practical constructions of LL/SC variables. In Elizabeth Borowsky and Sergio Rajsbaum, editors, *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing (PODC), 13–16 July 2003, Boston, MA, USA*, pages 285–294. ACM Press, 2003. (p. 18).

Prasad Jayanti and Srdjan Petrovic. Efficient wait-free implementation of multiword LL/SC variables. In Ten H. Lai and Anish Arora, editors, *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS), 6–10 June 2005, Columbus, OH, USA*, pages 59–68. IEEE Computer Society Press, 2005. (p. 18).

Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of LISP-like structures. In Alfred V. Aho, Stephen N. Zilles, and Barry K.

Rosen, editors, *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), 29–31 January 1979, San Antonio, TX, USA*, pages 244–256. ACM Press, 1979. (p. 49).

Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In Richard DeMillo, editor, *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 25–27 January 1982, Albuquerque, NM, USA*, pages 66–74. ACM Press, 1982. (p. 49).

Saul A. Kripke. Semantic analysis of modal logic (abstract). *Journal of Symbolic Logic*, 24:323–324, 1959. (p. 28).

Saul A. Kripke. Semantical analysis of modal logic I. Normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963. (p. 28).

Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the Association for Computing Machinery*, 47(2):312–360, 2000. (p. 34).

Edya Ladan-Mozes and Nir N. Shavit. An optimistic approach to lock-free FIFO queues. In Rachid Guerraoui, editor, *Proceedings of the 18th International Conference on Distributed Computing (DISC), 4–7 October 2004, Amsterdam, The Netherlands*, volume 3274 of *Lecture Notes in Computer Science*, pages 117–131. Springer-Verlag, 2004. (pp. 122, 126, 133, 197).

Edya Ladan-Mozes and Nir N. Shavit. An optimistic approach to lock-free FIFO queues. *Distributed Computing*, 20(5):323–341, 2008. (pp. 122, 126, 133, 197).

Anthony LaMarca. A performance evaluation of lock-free synchronization protocols. In James H. Anderson, David Peleg, and Elizabeth Borowsky, editors, *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC), 14–17 August 1994, Los Angeles, CA, USA*, pages 130–140. ACM Press, 1994. (p. 2).

Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977. (p. 15).

Leslie Lamport. "Sometime" is sometimes "Not Never". In Paul Abrahams, Richard Lipton, and Stephen Bourne, editors, *Proceedings of the 7th*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 28–30 January 1980, Las Vegas, NV, USA, pages 174–185. ACM Press, 1980. (p. 32).

Leslie Lamport. Checking a multithreaded algorithm with +CAL. In Shlomi Dolev, editor, *Proceedings of the 20th International Symposium on Distributed Computing (DISC), 18–21 September 2006, Stockholm, Sweden*, volume 4167 of *Lecture Notes in Computer Science*, pages 151–163. Springer-Verlag, 2006. (pp. 3, 135).

James Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 20–24 June 1988, Atlanta, GA, USA*, pages 24–31. ACM Press, 1988. (p. 49).

Tal Lev-Ami. TVLA: A framework for Kleene logic based static analyses. M.Sc. thesis, Tel-Aviv University, 2000. (pp. 47, 67, 143).

Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In Jens Palsberg, editor, *Proceedings of the 7th International Symposium on Static Analysis (SAS), 29 June – 1 July 2000, Santa Barbara, CA, USA*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer-Verlag, 2000. (p. 47).

Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In Debra J. Richardson and Mary Jean Harold, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 21–24 August 2000, Portland, OR, USA*, pages 26–38. Association for Computing Machinery, ACM Press, 2000. (p. 197).

Tal Lev-Ami, Roman Manevich, and Mooly Sagiv. TVLA: A system for generating abstract interpreters. In René Jacquart, editor, *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22–27 August 2004, Toulouse, France*, pages 367–376. Kluwer, 2004. (p. 47).

Tal Lev-Ami, Neil Immerman, Thomas Reps, Mooly Sagiv, Siddharth Srivastava, and Greta Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. *Logical Methods in Computer Science*, 5(2:12), 2009. (p. 76).

Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In Mary S. Van Deusen, Zvi Galil, and Brian K. Reid, editors, *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), 14–16 January 1985, New Orleans, LA, USA*, pages 97–107. ACM Press, 1985. (p. 33).

Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. Model checking linearizability via refinement. In Ana Cavalcanti and Dennis Dams, editors, *Proceedings of the 2nd World Congress on Formal Methods (FM), 2–6 November 2009, Eindhoven, The Netherlands*, volume 5850 of *Lecture Notes in Computer Science*, pages 321–337. Springer-Verlag, 2009. (pp. 92, 136).

Alexey Loginov, Thomas Reps, and Mooly Sagiv. Abstraction refinement via inductive learning. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV), 6–10 July 2005, Edinburgh, Scotland*, volume 3576 of *Lecture Notes in Computer Science*, pages 239–249. Springer-Verlag, 2005. (p. 238).

Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995. (p. 44).

Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996. (p. 95).

Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995. (p. 93).

Roman Manevich, Mooly Sagiv, Ganesan Ramalingam, and John Field. Partially disjunctive heap abstraction. In Roberto Giacobazzi, editor, *Proceedings of the 11th International Symposium on Static Analysis (SAS), 26–28 August 2004, Verona, Italy*, volume 3148 of *Lecture Notes in Computer Science*, pages 265–279. Springer-Verlag, 2004. (pp. 80, 181).

Roman Manevich, Josh Berdine, Byron Cook, Ganesan Ramalingam, and Mooly Sagiv. Shape analysis by graph decomposition. In Orna Grumberg and Michael Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*

(TACAS), 24 March – 1 April 2007, Braga, Portugal, volume 4424 of *Lecture Notes in Computer Science*, pages 3–18. Springer-Verlag, 2007. (p. 81).

Roman Manevich, Tal Lev-Ami, Mooly Sagiv, Ganesan Ramalingam, and Josh Berdine. Heap decomposition for concurrent shape analysis. In María Alpuente and Germán Vidal, editors, *Proceedings of the 15th International Symposium on Static Analysis (SAS), 16–18 July 2008, Valencia, Spain*, volume 5079 of *Lecture Notes in Computer Science*, pages 363–377. Springer-Verlag, 2008. (pp. 81, 195, 196, 197).

Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991. (p. 111).

Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT Computer Science and Artificial Intelligence Laboratory, September 2003. (p. 3).

Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-01, Computer Science Department, Columbia University, October 1991. (pp. 2, 15).

Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Ph.D. thesis, Carnegie Mellon University, 1992. (p. 36).

Igor Melatti, Robert Palmer, Geoffrey Sawaya, Yu Yang, Robert M. Kirby, and Ganesh Gopalakrishnan. Parallel *and* distributed model checking in Eddy. In Antti Valmari, editor, *Proceedings of the 13th SPIN Workshop on Model Checking Software, 30 March – 1 April 2006, Vienna, Austria*, volume 3925, pages 108–125. Springer-Verlag, 2006. (p. 3).

Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In Aleta Ricciardi, editor, *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing (PODC), 21–24 July 2002, Monteray, CA, USA*, pages 21–30. ACM Press, 2002. (p. 19).

Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6): 491–504, 2004. (p. 19).

Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In James E. Burns and Yoram Moses, editors, *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC), 23–26 May 1996, Philadelphia, PA, USA*, pages 267–275. ACM Press, 1996. (pp. 19, 25, 122, 124, 133, 195).

Maged M. Michael and Michael L. Scott. Non-blocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998. (pp. 1, 21, 25, 122, 124, 133).

Lynette I. Millett and Tim Teitelbaum. Issues in slicing Promela and its applications to model checking, protocol understanding, and simulation. *International Journal on Software Tools for Technology Transfer*, 2(4):343–349, 2000. (p. 42).

Mark Moir. Practical implementations of non-blocking synchronization primitives. In James E. Burns and Hagit Attiya, editors, *Proceedings of the 16th Annual Symposium on Principles of Distributed Computing (PODC), 21–24 August 1997, Santa Barbara, CA, USA*, pages 219–228. ACM Press, 1997. Correction in Moir [2001]. (p. 18).

Mark Moir. Correction: "Practical implementations of non-blocking synchronization primitives". In Ajay Kshemkalyani and Nir N. Shavit, editors, *Proceedings of the 20th Annual Symposium on Principles of Distributed Computing (PODC), 26–29 August 2001, Newport, RI, USA*, page 323. ACM Press, 2001. (p. 284).

Markus Müller-Olm, David A. Schmidt, and Bernhard Steffen. Model-checking: A tutorial introducton. In Agostino Cortesi and Gilberto Filé, editors, *Proceedings of the 6th International Symposium on Static Analysis (SAS), 22–24 September 1999, Venice, Italy*, volume 1694 of *Lecture Notes in Computer Science*, pages 330–354. Springer-Verlag, 1999. (p. 28).

Sumit Nain and Moshe Y. Vardi. Branching vs linear time: Semantical perspective. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA), 22–25 October 2007, Tōkyō, Japan*, volume 4762 of *Lecture Notes in Computer Science*, pages 19–34. Springer-Verlag, 2007. (p. 32).

Johannes Oetsch, Michael Prischink, Jörg Pührer, Martin Schwengerer, and Hans Tompits. On the small-scope hypothesis for testing answer-set programs. In Gerhard Brewka, Thomas Eiter, and Sheila A. McIlraith, editors, *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR), 10–14 June 2012, Rome, Italy*, pages 43–53. AAAI Press, 2012. (p. 3).

Sam Owre, John Rushby, Natarajan Shankar, and David W. J. Stringer-Calvert. PVS: An experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *Proceedings of the International Workshop on Current Trends in Applied Formal Methods (FM-Trends), 7–9 October 1998, Boppard, Germany*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345. Springer-Verlag, 1998. (p. 95).

Doron A. Peled. Combining partial order reductions with on-the-fly model-checking. In David L. Dill, editor, *Proceedings of the 6th International Conference on Computer Aided Verification (CAV), 21–23 June 1994, Stanford, CA, USA*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer-Verlag, 1994. (p. 41).

Doron A. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, 1996. (p. 41).

Dominique Perrin and Jean-Éric Pin. *Infinite Words: Automata, Semigroups, Logic and Games*. Elsevier, 2004. (p. 34).

John Plevyak, Andrew A. Chien, and Vijay Karamcheti. Analysis of dynamic structures for efficient parallel execution. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing (LCPC), 12–14 August 1993, Portland, OR, USA*, volume 768 of *Lecture Notes in Computer Science*, pages 37–56. Springer-Verlag, 1993. (p. 49).

Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS), 31 October – 2 November 1977, Providence, RI, USA*, pages 46–57. IEEE Computer Society Press, 1977. (pp. 29, 32).

Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981. (p. 29).

Amir Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In Wilfried Brauer, editor, *Proceedings of the 12th International Colloquium on Automata, Languages and Programming (ICALP), 15–19 July 1985, Nafplion, Greece*, volume 194 of *Lecture Notes in Computer Science*, pages 15–32. Springer-Verlag, 1985. (p. 32).

Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In Ed Brinksma and Kim G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV), 27–31 July 2002, Copenhagen, Denmark*, volume 2404 of *Lecture Notes in Computer Science*, pages 107–122. Springer-Verlag, 2002. (p. 43).

Sundeep Prakash, Yann Hang Lee, and Theodore Johnson. Non-blocking algorithms for concurrent data structures. Technical Report 91-002, University of Florida, July, 1991. (p. 2).

Arthur N. Prior. *Time and Modality*. Oxford University Press, 1957. (p. 29).

Arthur N. Prior. *Past, Present and Future*. Clarendon Press, Oxford, 1967. (p. 29).

Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *Proceedings of the 5th International Symposium on Programming, 6–8 April 1982, Torino, Italy*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982. (p. 27).

Michel A. Reniers and Tim A. C. Willemse. Folk theorems on the correspondence between state-based and event-based systems. In Ivana Cerná, Tibor Gyimóthy, Juraj Hromkovic, Keith G. Jeffery, Rastislav Královic, Marko Vukolić, and Stefan Wolf, editors, *Proceedings of the 37th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), 22–28 January 2011, Nový Smokovec, Slovakia*, volume 6543 of *Lecture Notes in Computer Science*, pages 494–505. Springer-Verlag, 2011. (p. 28).

Thomas Reps, Mooly Sagiv, and Alexey Loginov. Finite differencing of logical formulas for static analysis. In Pierpaolo Degano, editor, *Proceedings of the 12th European Symposium on Programming (ESOP), 7–11 April 2003, Warsaw, Poland*, volume 2618 of *Lecture Notes in Computer Science*, pages 330–398. Springer-Verlag, 2003. (p. 81).

Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Static program analysis via 3-valued logic. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV), 13–17 July 2004, Boston, MA, USA*, volume 3114 of *Lecture Notes in Computer Science*, pages 15–30. Springer-Verlag, 2004a. (p. 49).

Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In Bernhard Steffen and Giorgio Levi, editors, *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI), 11–13 January 2004, Venice, Italy*, volume 2937 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2004b. (p. 76).

Thomas Reps, Mooly Sagiv, and Alexey Loginov. Finite differencing of logical formulas for static analysis. *ACM Transactions on Programming Languages and Systems*, 32(6), 2010. Article 24. (p. 81).

John C. Reynold. Separation logic: A logic for shared mutable data structures. In Samson Abramsky and Gordon D. Plotkin, editors, *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS), 22–25 July 2002, Copenhagen, Denmark*, pages 55–74. IEEE Computer Society Press, 2002. (p. 2).

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In Hans-Juergen Boehm and Guy L. Steele, Jr, editors, *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 21–24 January 1996, St Petersburg Beach, FL, USA*, pages 16–31. ACM Press, 1996. (p. 49).

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998. (p. 49).

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In Andrew Appel and Alex Aiken, editors, *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 20–22 January 1999, San Antonio, TX, USA*, pages 105–118. ACM Press, 1999. (p. 49).

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002. (pp. 4, 43, 49, 55, 57, 64, 66, 67, 70, 79, 80, 197).

Mooly Sagiv, Thomas Reps, Reinhard Wilhelm, and Eran Yahav. On the utility of canonical abstraction. In Manfred Broy, Johannes Gruenbauer, David Harel, and C. A. R. Hoare, editors, *Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems, 3–15 August 2004, Marktoberdorf, Germany*, volume 195 of *NATO Science Series II*, pages 215–253. Springer-Verlag, 2005. (p. 49).

Philippe Schnoebelen. The complexity of temporal logic model checking. In Philippe Balbiani, Nobu-Yuki Suzuki, Frank Wolter, and Michael Zakharyaschev, editors, *Advanced in Modal Logic 4*, pages 393–436. King's College Publications, 2003. (p. 33).

Chien-Hua Shann, Ting-Lu Huang, and Cheng Chen. A practical nonblocking queue algorithm using Compare-and-Swap. In Masatoshi Miyazaki and Makoto Takizawa, editors, *Proceedings of the 7th International Conference on Parallel and Distributed Systems (ICPADS), 4–7 July 2000, Iwate, Japan*, pages 470–475. IEEE Computer Society Press, 2000. (pp. 2, 122, 133).

A. Prasad Sistla. Employing symmetry reductions in model checking. *Computer Languages, Systems and Structures*, 30(3–4):99–137, 2004. (p. 42).

A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *Journal of the Association for Computing Machinery*, 32(3):733–749, 1985. (pp. 39, 92).

A. Prasad Sistla and Patrice Godefroid. Symmetry and reduced symmetry in model checking. *ACM Transactions on Programming Languages and Systems*, 26(4):702–734, 2004. (p. 42).

A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. SMC: A symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, 9(2):133–166, 2000. (p. 42).

Graeme Smith and John Derrick. Model checking downwards simulations. In John Derrick and Eerke A. Boiten, editors, *Proceedings of the*

BCS FACS Refinement Workshop (REFINE), 12 April 2005, Guildford, England, UK, volume 137.2 of *Electronic Notes in Theoretical Computer Science*, pages 205–224. Elsevier, 2005. (pp. 9, 95, 96).

Graeme Smith and John Derrick. Verifying data refinements using a model checker. *Formal Aspects of Computing*, 18(3):264–287, 2006. (pp. 95, 96).

J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992. (p. 95).

Colin Stirling. Modal and temporal logics. In Samson Abramsky, Dov M. Gabbay, and Thomas S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2. Background: Computational Structures*, pages 477–563. Oxford University Press, 1992. (p. 29).

Janice M. Stone. A simple and correct shared-queue algorithm using Compare-and-Swap. In Joanne L. Martin, editor, *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing (SC), 12–16 November 1990, New York, NY, USA*, pages 495–504. IEEE Computer Society Press, 1990. (p. 2).

Janice M. Stone. A nonblocking Compare-and-Swap algorithm for a shared circular queue. In Spyros G. Tzafestas, Pierre Borne, and Lucio Grandinetti, editors, *Proceedings of the IMACS/IFAC International Symposium on Parallel and Distributed Computing in Engineering Systems, 23–28 June 1991, Corfu, Greece*, pages 147–152. North-Holland, 1992. (p. 2).

Jan Stransky. A lattice for abstract interpretation of dynamic (LISP-like) structures. *Information and Computation*, 101(1):70–102, 1992. (p. 49).

Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards flexible verificaton under fairness. In Ahmed Bouajjani and Oded Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV), 26 June – 2 July 2009, Grenoble, France*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer-Verlag, 2009. (p. 136).

Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995. (p. 42).

R. Kent Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Centre, April 1986. (pp. 21, 122, 124, 133).

Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In Arnold Rosenberg, editor, *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), 4–6 July 2001, Heraklion, Greece*, pages 134–143. ACM Press, 2001. (p. 2).

John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the 11th ACM Symposium on Principles of Database Systems (PODS), 2–4 June 1991, San Diego, CA, USA*, pages 212–222. ACM Press, 1992. (p. 2).

Viktor Vafeiadis. *Modular Fine-Grained Concurrency Verification*. Ph.D. thesis, University of Cambridge, 2007. Available as Vafeiadis [2008]. (p. 2).

Viktor Vafeiadis. Modular fine-grained concurrency verification. Technical Report 726, Computer Laboratory, University of Cambridge, 2008. (pp. 110, 290).

Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In Neil D. Jones and Markus Müller-Olm, editors, *Proceedings of the 10th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI), 18–20 January 2009, Savannah, GA, USA*, volume 5403 of *Lecture Notes in Computer Science*, pages 335–348. Springer-Verlag, 2009. (pp. 2, 110).

Viktor Vafeiadis, Maurice P. Herlihy, C. A. R. Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In Josep Torrellas and Siddhartha Chatterjee, editors, *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 29–31 March 2006, New York City, NY, USA*, pages 129–136. ACM Press, 2006. (p. 92).

John Valois. Implementing lock-free queues. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems (PDCS), 6–8 October 1994, Las Vegas, NV, USA*, pages 64–69. International Society for Computers and their Applications, 1994. (p. 2).

John Valois. Lock-free linked lists using Compare-and-Swap. In James H. Anderson, editor, *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC), 20–23 August 1995, Ottawa, Ontario, Canada*, pages 214–222. ACM Press, 1995. (p. 2).

Moshe Y. Vardi. Linear vs branching time: A complexity-theoretic perspective. In John Mitchell and Vaughan Pratt, editors, *Proceedings of the 13th IEEE Symposium on Logic in Computer Science (LICS), 21–24 June 1998, Indianapolis, IN, USA*, pages 394–405. IEEE Computer Society Press, 1998a. (p. 32).

Moshe Y. Vardi. Sometimes and Not Never re-revisited: On branching versus linear time. In Davide Sangiorgi and Robert de Simone, editors, *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR), 8–11 September 1998, Nice, France*, volume 1466 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 1998b. (p. 32).

Moshe Y. Vardi. Branching vs linear time: Final showdown. In Tiziana Margaria and Wang Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2–6 April 2001, Genova, Italy*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, 2001. (p. 32).

Moshe Y. Vardi. From Church and Prior to PSL. In Grumberg and Veith [2008], pages 150–171. (p. 29).

Moshe Y. Vardi. From monadic logic to PSL. In Arnon Avron, Nachum Dershowitz, and Alexander M. Rabinovich, editors, *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of *Lecture Notes in Computer Science*, pages 656–681. Springer-Verlag, 2008b. (p. 29).

Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st Symposium on Logic in Computer Science (LICS), 16–18 June 1986, Cambridge, MA, USA*, pages 332–344. IEEE Computer Society Press, 1986. (p. 34).

Shobha Vasudevan, E. Allen Emerson, and Jacob A. Abraham. Efficient model checking of hardware using conditioned slicing. In Michael Huth, editor, *Proceedings of the 4th International Workshop on Automated*

Verification of Critical Systems (AVoCS), 4 September 2004, London, England, UK, volume 128.6 of *Electronic Notes in Theoretical Computer Science*, pages 279–294. Elsevier, 2005. (p. 42).

Martin T. Vechev, Eran Yahav, and Greta Yorsh. Experience with model checking linearizability. In Corina S. Păsăreanu, editor, *Proceedings of the 16th International SPIN Workshop on Model Checking Software, 26–28 June 2009, Grenoble, France*, volume 5578 of *Lecture Notes in Computer Science*, pages 261–278. Springer-Verlag, 2009. (p. 135).

Edward Yan-Bing Wang. *Analysis of Recursive Types in an Imperative Language*. Ph.D. thesis, University of California, Berkeley, 1994. (p. 49).

Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2), March 2005. (p. 42).

Eran Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In Chris Hankin and David A. Schmidt, editors, *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 17–19 January 2001, London, England, UK*, pages 27–40. ACM Press, 2001. (pp. 47, 64, 76, 77).

Eran Yahav. *Property-Guided Verification of Concurrent Heap-Manipulating Programs*. Ph.D. thesis, University of Tel-Aviv, 2004. (pp. 47, 76, 77).

Eran Yahav and Mooly Sagiv. Automatically verifying concurrent queue algorithms. In Byron Cook, Scott Stoller, and Willem Visser, editors, *Proceedings of the 2nd Workshop on Software Model Checking (SoftMC), 14 July 2003, Boulder, CO, USA*, volume 89.3 of *Electronic Notes in Theoretical Computer Science*, pages 450–463. Elsevier, 2003. (p. 195).

Eran Yahav and Mooly Sagiv. Verifying safety properties of concurrent heap-manipulating programs. *ACM Transactions on Programming Languages and Systems*, 32(5), 2010. Article 18. (pp. 64, 76).

Eran Yahav, Thomas Reps, and Mooly Sagiv. LTL model checking for systems with unbounded number of dynamically created threads and objects. Technical Report 1424, Computer Sciences Department, University of Wisconsin, Madison, March 2001. (p. 237).

Eran Yahav, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Verifying temporal heap properties specified via evolution logic. In Pierpaolo Degano, editor, *Proceedings of the 12th European Symposium on Programming (ESOP), 7–11 April 2003, Warsaw, Poland*, volume 2618 of *Lecture Notes in Computer Science*, pages 204–222. Springer-Verlag, 2003. (p. 237).

Eran Yahav, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Verifying temporal heap properties specified via evolution logic. *Logic Journal of the Interest Group in Pure and Applied Logics*, 14(5):755–783, 2006. (p. 237).

Greta Yorsh, Thomas Reps, and Mooly Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 29 March – 2 April 2004, Barcelona, Spain*, volume 2988 of *Lecture Notes in Computer Science*, pages 530–545. Springer-Verlag, 2004. (p. 76).

Greta Yorsh, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Logical characterizations of heap abstractions. *ACM Transactions on Computational Logic*, 8(1), 2007. (p. 76).

Shao Jie Zhang and Yang Liu. Model checking a lazy concurrent list-based set algorithm. In *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement (SSIRI), 9–11 June 2010, Singapore*, pages 43–52. IEEE Computer Society, 2010. (p. 136).

Shao Jie Zhang, Yang Liu, Jun Sun, Jin Song Dong, Wei Chen, and Yanhong A. Liu. Formal verification of Scalable NonZero Indicators. In *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE), 1–3 July 2009, Boston, MA, USA*, pages 406–411, Skokie, IL, USA, 2009. Knowledge Systems Institute. ISBN 1-891706-24-1. (p. 136).