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

Acknowledgements

Contents

List of Figures

List of Tables

I  Introduction

1.1 Nonblocking Concurrency ........................................ 1
1.2 Model Checking ................................................... 2
1.3 Bounded Verification ............................................. 3
1.4 Unbounded Verification ........................................... 4
1.5 Outline ........................................................... 5

I  Background

2  Nonblocking Data Structures

2.1 Concurrent Data Structures ........................................ 9
2.2 Linearisability ..................................................... 11
2.3 Mutual Exclusion ................................................ 13
2.4 Nonblocking Progress Properties ................................ 15
  2.4.1 Wait-freedom ................................................. 15
  2.4.2 Lock-freedom .................................................. 15
  2.4.3 Obstruction-freedom ......................................... 16
2.5 Nonblocking Synchronization ..................................... 16
  2.5.1 Compare-and-Swap .......................................... 16
  2.5.2 Load-Linked, Store-Conditional ............................ 17
CONTENTS

2.5.3 ABA Problem ........................................... 18
2.5.4 Memory Management ................................... 18
2.6 Algorithms .............................................. 19
   2.6.1 Lock-free Stack .................................... 20
   2.6.2 Lock-free Unbounded Queue ...................... 22

3 Model Checking ........................................... 27
   3.1 Systems ................................................ 28
   3.2 Specifications ......................................... 29
      3.2.1 CTL* ........................................... 29
      3.2.2 LTL ............................................. 32
      3.2.3 CTL ............................................. 32
      3.2.4 Linear- versus Branching-time ................ 32
   3.3 Model Checking Paradigms ............................. 33
      3.3.1 Explicit State Model Checking .................. 34
      3.3.2 BDD-Based Symbolic Model Checking .......... 36
      3.3.3 Bounded Model Checking ....................... 39
   3.4 Tackling the State Explosion Problem .............. 39
      3.4.1 Bitstate hashing ................................. 40
      3.4.2 Partial order reduction ......................... 40
      3.4.3 Slicing ......................................... 42
      3.4.4 Symmetry reduction ............................. 42
   3.5 Abstraction ........................................... 43
      3.5.1 Constructing Abstract Models .................. 44
   3.6 Tools ................................................ 46
      3.6.1 Spin ............................................ 46
      3.6.2 SAL ............................................. 47
      3.6.3 TVLA/3VMC ..................................... 47

4 Canonical Abstraction .................................. 49
   4.1 Canonical Abstraction ................................ 49
      4.1.1 States as Logical Structures .................. 50
      4.1.2 Embeddings ..................................... 52
      4.1.3 Properties ...................................... 55
         Syntax ............................................. 55
         Semantics ......................................... 56
         Embedding Theorem ................................ 57
   4.2 Refining Abstractions ................................ 57
III Verification 137

7 Canonical Abstraction for Linearisability 139

7.1 Basic Stack Model 141

7.2 Three-Valued Model 143

7.2.1 Core Predicates 145

7.2.2 Integrity Rules 145

7.2.3 Instrumentation Predicates 146

Reachability and circularity 146

Has-a-field 147

Referenced-by-field 147

Shared 148

7.3 Preserving Linearisability Information 148

7.3.1 Matching Values 151

7.3.2 Ordered Values 154

7.3.3 Hanging Head 156

7.4 (Un)bounded Threads 159

7.4.1 Bounded Threads 161

7.4.2 Thread Field Properties 162

Snapshots 162

Data Values 164

7.5 Initial State 167

7.6 Additional Compatibility Constraints 169

7.6.1 Reachability Predicates 169

7.6.2 Geometric Predicates 177

7.7 Stack Results 179

7.8 Stack Variations 181

7.8.1 TVLA Changes 181

Isomorphic State Comparison 181

TVLA 2 182

7.8.2 Definitions 183

Unnecessary Instrumentation Predicates 183

Pure Initial State 184

Named Threads 185

Thread Bounding Constraint 185

7.8.3 Full Interleaving 186

7.9 Queue Models 188

7.9.1 Three-Valued Models 188

Core Predicates 188
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.9.2</td>
<td>Results</td>
<td>194</td>
</tr>
<tr>
<td>7.10</td>
<td>Related Work</td>
<td>195</td>
</tr>
<tr>
<td>7.11</td>
<td>Conclusion</td>
<td>196</td>
</tr>
<tr>
<td>8</td>
<td>Collapsing Threads Safely with Soft Invariants</td>
<td>199</td>
</tr>
<tr>
<td>8.1</td>
<td>Overview</td>
<td>200</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Instrumentation Predicates</td>
<td>200</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Selection of Predicates</td>
<td>202</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Compatibility Constraints</td>
<td>205</td>
</tr>
<tr>
<td>8.2</td>
<td>Stack Models</td>
<td>206</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Changes to the Model</td>
<td>206</td>
</tr>
<tr>
<td></td>
<td>Collapse</td>
<td>206</td>
</tr>
<tr>
<td></td>
<td>Null Equivalence</td>
<td>207</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Soft Invariants</td>
<td>209</td>
</tr>
<tr>
<td></td>
<td>Interleaved Locations</td>
<td>209</td>
</tr>
<tr>
<td></td>
<td>Non-interleaved Locations</td>
<td>213</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Results</td>
<td>213</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Full Interleaving</td>
<td>213</td>
</tr>
<tr>
<td></td>
<td>Soft Invariants</td>
<td>214</td>
</tr>
<tr>
<td></td>
<td>Results</td>
<td>214</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Extra Predicates</td>
<td>216</td>
</tr>
<tr>
<td>8.3</td>
<td>Queue Models</td>
<td>217</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Changes to the model</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>Full interleaving</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>Ordered snapshots</td>
<td>218</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Soft Invariants</td>
<td>224</td>
</tr>
<tr>
<td></td>
<td>Enqueue</td>
<td>228</td>
</tr>
<tr>
<td></td>
<td>Simplified Dequeue</td>
<td>228</td>
</tr>
<tr>
<td></td>
<td>Original Dequeue</td>
<td>228</td>
</tr>
<tr>
<td></td>
<td>Restricted Interleaving</td>
<td>229</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Results</td>
<td>229</td>
</tr>
<tr>
<td>8.4</td>
<td>Conclusion</td>
<td>231</td>
</tr>
<tr>
<td>IV</td>
<td>Conclusion</td>
<td>233</td>
</tr>
<tr>
<td>9</td>
<td>Conclusion</td>
<td>235</td>
</tr>
<tr>
<td>9.1</td>
<td>Bounded Verification</td>
<td>235</td>
</tr>
</tbody>
</table>
CONTENTS

9.1.1 Future Research ................................................. 236
9.2 Unbounded Verification ........................................... 237
  9.2.1 Future Research ............................................. 238

A Proofs of Canonical Abstraction Compatibility Constraints 239
  A.1 Assumptions ...................................................... 239
  A.2 Soft Invariants .................................................. 241
    A.2.1 Single predicate ........................................... 241
    A.2.2 Conditional property ...................................... 242
  A.3 Linear .......................................................... 243
    A.3.1 To null ...................................................... 243
    A.3.2 Same 1 ...................................................... 243
    A.3.3 Same 2 ...................................................... 244
    A.3.4 Ordered ...................................................... 245
  A.4 Geometric ...................................................... 246
    A.4.1 Triangle 1 ................................................... 246
    A.4.2 Triangle 2 ................................................... 247
    A.4.3 Square 1 ..................................................... 249
    A.4.4 Square 2 ..................................................... 250
    A.4.5 Square 3 ..................................................... 252
  A.5 Reachability ................................................... 253
    A.5.1 No self loop ................................................ 253
    A.5.2 No loop back ............................................... 254
    A.5.3 No loop to head ............................................ 255
    A.5.4 Unreachable ................................................ 256
    A.5.5 Chain ....................................................... 256

Bibliography .......................................................... 259
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Basic counter implementation</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Initial part of the counter execution graph</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Two counter executions</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Counter algorithm with a lock</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Pseudocode for Compare-and-Swap</td>
<td>17</td>
</tr>
<tr>
<td>2.6</td>
<td>Nonblocking counter increment operations</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>(a) Using CAS</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>(b) Using LL/SC</td>
<td>17</td>
</tr>
<tr>
<td>2.7</td>
<td>A lock-free stack algorithm</td>
<td>20</td>
</tr>
<tr>
<td>2.8</td>
<td>A lock-free linked list based queue</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>(a) Initialisation and enqueue operation</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>(b) Original dequeue operation</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>(c) Simplified dequeue operation</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>Example Kripke structure</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>A Kripke structure and its computation tree</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Two equivalent system representations</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>(a) Kripke structure</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>(b) Büchi automaton</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>A Büchi automaton for $\text{AG}p$</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>A binary decision tree for a two-bit comparator</td>
<td>37</td>
</tr>
<tr>
<td>3.6</td>
<td>OBDDs for a two-bit comparator</td>
<td>38</td>
</tr>
<tr>
<td>3.7</td>
<td>Independent transition execution</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>A list of length three</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>Graph of concrete configuration</td>
<td>51</td>
</tr>
<tr>
<td>4.3</td>
<td>Graph of abstract configuration</td>
<td>53</td>
</tr>
<tr>
<td>4.4</td>
<td>Canonical abstraction of a non-functional field</td>
<td>59</td>
</tr>
<tr>
<td>4.5</td>
<td>Three different lists have the same canonical abstraction</td>
<td>61</td>
</tr>
</tbody>
</table>
4.6 Using instrumentation predicates to distinguish three different list structures ........................................ 62
   (a) Connected, acyclic ........................................ 62
   (b) Disconnected, acyclic .................................. 63
   (c) Connected, cyclic .................................... 64
4.7 Focus on a single predicate: Focus(p(v1, v2)) ................. 68
4.8 Graph of a stack algorithm state with concurrent threads ... 78
5.1 A lock-free stack algorithm with merged transitions ......... 89
5.2 Diagram of thread i in the concurrent stack specification .... 91
5.3 Operations of thread i in the concurrent stack specification .. 91
5.4 A lock-free linked list based queue with prophecy variables 101
5.5 Diagrams of Dequeue operation of Thread i of a concurrent stack specification .................................. 103
   (a) Original ................................................. 103
   (b) With multiple linearisation points ..................... 103
5.6 A lock-free stack algorithm with merged specification ..... 105
5.7 A lock-free queue algorithm with merged specification ..... 107
   (a) Enqueue operation .................................. 107
   (b) Original dequeue operation .......................... 108
   (c) Simplified dequeue operation ......................... 109
6.1 Simplified execution of stack showing ABA error counterexample ...................................................... 120
6.2 Promela never claim for LTL formula WF-1 ................. 131
6.3 Simplified execution of stack showing wait-freedom counterexample .................................................... 132
7.1 Instrumentation predicate structures ................................ 140
   (a) Linear predicates .................................... 140
   (b) Geometric predicates ................................. 140
7.2 Transitions of stack model .................................. 142
7.3 Update actions used in stack model .......................... 144
7.4 Canonical abstraction of two lists: the property of matching values is lost ........................................... 149
7.5 Stack specification update operations, incorporating the spec relation .................................................... 150
7.6 Canonical abstraction of stack lists using the predicate matching: the property of matching values is retained 152
## LIST OF FIGURES

(a) All values match ........................................... 152
(b) Head values are different ................................. 153

### 7.7 Canonical abstraction with “crossed” spec predicates: the property of ordered values is lost ......................... 155

### 7.8 Canonical abstraction using the commutes predicate: the property of ordered values is retained .................. 157

(a) Ordered values ........................................... 157
(b) Unordered values ......................................... 158

### 7.9 In between the implementation and specification push updates the old implementation head node is abstracted with the list body ..................................................... 159

### 7.10 With the hasSpec predicate the old implementation head node is distinguished ........................................... 160

### 7.11 Canonical abstraction of threads: the relationships between fields’ values are lost ........................................... 163

### 7.12 Canonical abstraction using the succ[ss, ssnext] predicate: the relationships between fields’ values are retained .......... 165

(a) All next-successors ........................................ 165
(b) None next-successors ..................................... 166

### 7.13 Initial state of the stack model (plain) ..................... 168

### 7.14 Initial state of the stack model (with “garbage” nodes) ..................................................................... 168

### 7.15 Steps of a Pop update transition ................................ 170

(a) Initial state ........................................... 170
(b) After Focus (one of several states) ..................... 170
(c) After Coerce ........................................... 171
(d) Final state after Update, Coerce and Blur ............ 171

### 7.16 Transition to an inconsistent state .................................................. 173

(a) Initial state ........................................... 173
(b) After Focus (one of several states) ..................... 173
(c) After Coerce ........................................... 174
(d) After Update and Coerce ................................ 174
(e) Final state, after Blur .................................... 175

### 7.17 Incomplete matching triangle .................................... 177

### 7.18 Transitions of stack model .................................. 187

### 7.19 Transitions of queue models ................................ 189

(a) Enqueue operation ........................................ 189
(b) Original dequeue operation ............................. 190
(c) Simplified dequeue operation ........................... 191

### 7.20 Additional update operations used in queue models .......... 192
LIST OF FIGURES

8.1 Collapsing threads: loss of precision .............................................. 201
   (a) Three previously canonical thread objects ......................... 201
   (b) Collapsed thread object ................................................. 201
   (c) Thread with different properties is concretised .................. 201
8.2 Collapsing threads: properties preserved with soft invariants .......... 203
   (a) Three previously canonical thread objects ......................... 203
   (b) Collapsed thread object ................................................. 203
8.3 Collapsing threads: soft invariant for non-invariant property .......... 204
   (a) Three previously canonical thread objects ......................... 204
   (b) Collapsed thread object ................................................. 204
8.4 Concretised thread object to be sharpened .................................. 205
8.5 Property of null equivalence is lost when threads are collapsed ....... 208
   (a) Before collapse ......................................................... 208
   (b) After collapse .......................................................... 208
8.6 Soft invariant instrumentation predicates for interleaving locations .... 211
8.7 Concretising an unreachable state .............................................. 212
   (a) Abstract state ............................................................ 212
   (b) After Focus and Coerce ................................................. 212
   (c) After Update ............................................................. 212
8.8 Transitions of queue models ..................................................... 219
   (a) Enqueue operation ....................................................... 219
   (b) Original Dequeue operation .......................................... 220
   (c) Simplified Dequeue operation ....................................... 221
8.9 Snapshot order not preserved .................................................... 222
   (a) Abstract state ............................................................ 222
   (b) After Focus and Coerce ................................................. 223
List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Example integrity rules</td>
<td>60</td>
</tr>
<tr>
<td>4.2</td>
<td>Example instrumentation predicates</td>
<td>65</td>
</tr>
<tr>
<td>6.1</td>
<td>Minimum parameters for counterexamples to linearisability</td>
<td>122</td>
</tr>
<tr>
<td>6.2</td>
<td>Treiber Stack Bounded Verification</td>
<td>124</td>
</tr>
<tr>
<td>6.3</td>
<td>MS Queue Bounded Verification</td>
<td>125</td>
</tr>
<tr>
<td>6.4</td>
<td>DGLM Queue Bounded Verification</td>
<td>125</td>
</tr>
<tr>
<td>6.5</td>
<td>LMS Queue Bounded Verification</td>
<td>126</td>
</tr>
<tr>
<td>6.6</td>
<td>Treiber Stack Bounded Verification With Symmetry Reduction</td>
<td>128</td>
</tr>
<tr>
<td>6.7</td>
<td>Minimum parameters for counterexamples to nonblocking properties</td>
<td>133</td>
</tr>
<tr>
<td>6.8</td>
<td>Treiber Stack Bounded Verification</td>
<td>134</td>
</tr>
<tr>
<td>6.9</td>
<td>MS Queue Bounded Verification</td>
<td>134</td>
</tr>
<tr>
<td>6.10</td>
<td>DGLM Queue Bounded Verification</td>
<td>135</td>
</tr>
<tr>
<td>6.11</td>
<td>LMS Queue Bounded Verification</td>
<td>135</td>
</tr>
<tr>
<td>6.12</td>
<td>Treiber Stack Bounded Verification With Symmetry Reduction</td>
<td>128</td>
</tr>
<tr>
<td>7.1</td>
<td>Stack verification results</td>
<td>179</td>
</tr>
<tr>
<td>7.2</td>
<td>Canonical thread objects in unscheduled abstract stack states</td>
<td>181</td>
</tr>
<tr>
<td>7.3</td>
<td>Stack analyses using isomorphic state comparisons</td>
<td>182</td>
</tr>
<tr>
<td>7.4</td>
<td>Comparison with TVLA 2</td>
<td>182</td>
</tr>
<tr>
<td>7.5</td>
<td>Stack analyses with unnecessary referenced-by instrumentation predicates</td>
<td>183</td>
</tr>
<tr>
<td>7.6</td>
<td>Stack analyses with a pure initial state</td>
<td>184</td>
</tr>
<tr>
<td>7.7</td>
<td>Stack analyses with named threads</td>
<td>185</td>
</tr>
<tr>
<td>7.8</td>
<td>Stack verification results for bounded threads</td>
<td>186</td>
</tr>
<tr>
<td>(a) Resources</td>
<td></td>
<td>186</td>
</tr>
<tr>
<td>(b) Statespace</td>
<td></td>
<td>186</td>
</tr>
<tr>
<td>7.9</td>
<td>Stack verification results with no partial order reduction</td>
<td>187</td>
</tr>
<tr>
<td>7.10</td>
<td>Queue verification results</td>
<td>194</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Invariant properties of stack model with restricted interleaving</td>
<td>210</td>
</tr>
<tr>
<td>8.2</td>
<td>Stack verification results for restricted interleaving</td>
<td>214</td>
</tr>
<tr>
<td></td>
<td>(a) Resources</td>
<td>214</td>
</tr>
<tr>
<td></td>
<td>(b) Statespace</td>
<td>214</td>
</tr>
<tr>
<td>8.3</td>
<td>Invariant properties of stack with full interleaving</td>
<td>215</td>
</tr>
<tr>
<td>8.4</td>
<td>Stack verification results for full interleaving</td>
<td>216</td>
</tr>
<tr>
<td></td>
<td>(a) Resources</td>
<td>216</td>
</tr>
<tr>
<td></td>
<td>(b) Statespace</td>
<td>216</td>
</tr>
<tr>
<td>8.5</td>
<td>Stack verification results for full interleaving with extra predicates</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>(a) Resources</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>(b) Statespace</td>
<td>218</td>
</tr>
<tr>
<td>8.6</td>
<td>Invariant properties</td>
<td>225</td>
</tr>
<tr>
<td></td>
<td>(a) Enqueue operation</td>
<td>225</td>
</tr>
<tr>
<td></td>
<td>(b) Original Dequeue operation</td>
<td>226</td>
</tr>
<tr>
<td></td>
<td>(c) Simplified Dequeue operation</td>
<td>227</td>
</tr>
<tr>
<td>8.7</td>
<td>Queue verification results for restricted interleaving</td>
<td>230</td>
</tr>
<tr>
<td>8.8</td>
<td>Queue verification results for full interleaving</td>
<td>230</td>
</tr>
<tr>
<td></td>
<td>(a) Resources</td>
<td>230</td>
</tr>
<tr>
<td></td>
<td>(b) Statespace</td>
<td>230</td>
</tr>
</tbody>
</table>
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.
CHAPTER 1. INTRODUCTION

[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 non-blocking 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 Cansel, 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 $M$, checking a specification property $\phi$ (traditionally formalised in a temporal logic). Thus, it is able to determine the satisfiability of the property in the system $(M \models \phi)$, 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-
presentation 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 non-blocking 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”) 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

\footnote{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: \( c : \text{integer} := 0 \)

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 2.1](image.png)

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

\[
c := c + 1
\]
2.2. LINEARISABILITY

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 \( \text{inv}_p(\text{OP}, \text{val}^*) \) and responses as \( \text{resp}_p(\text{OP}, \text{val}^*) \), where \( p \) is a process/thread name, \( \text{OP} \) is an operation, and \( \text{val}^* \) 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 \( H \), \( \text{complete}(H) \) 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:

$$OP_1 <_H OP_2 \quad \text{if \ resp}(OP_1) \ \text{precedes inv}(OP_2) \ \text{in} \ H.$$ 

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

- $\text{complete}(H')$ 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:

$inv_1(INC), inv_2(INC), resp_1(INC, 1), inv_1(DEC), resp_1(DEC, 0), resp_2(INC, 1)$

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

$inv_1(INC), resp_1(INC, 1), inv_1(DEC), resp_1(DEC, 0), inv_2(INC), resp_2(INC, 1)$

The execution on the right has the following trace:

$inv_1(INC), inv_2(INC), resp_2(INC, 1), resp_1(INC, 1)$

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

$inv_i(INC), resp_i(INC, 1), inv_j(INC), resp_j(INC, 1)$

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*). 

\[\square\]
2.3. MUTUAL EXCLUSION

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{inv}_1(\text{INC}) )</td>
<td>( \text{inv}_2(\text{INC}) )</td>
</tr>
<tr>
<td>( a := 0 )</td>
<td>( a := 0 )</td>
</tr>
<tr>
<td>( b := 1 )</td>
<td>( b := 1 )</td>
</tr>
<tr>
<td></td>
<td>( c := 1 )</td>
</tr>
<tr>
<td>( \text{resp}_1(\text{INC}, 1) )</td>
<td>( \text{resp}_2(\text{INC}, 1) )</td>
</tr>
<tr>
<td>( \text{inv}_1(\text{DEC}) )</td>
<td>( \text{resp}_1(\text{INC}, 1) )</td>
</tr>
<tr>
<td>( a := 1 )</td>
<td></td>
</tr>
<tr>
<td>( b := 0 )</td>
<td></td>
</tr>
<tr>
<td>( c := 0 )</td>
<td></td>
</tr>
<tr>
<td>( \text{resp}_1(\text{DEC}, 0) )</td>
<td></td>
</tr>
</tbody>
</table>

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
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, 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.

---

3Though 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.

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.

\[\text{\footnotesize 4} \] 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
2.5. NONBLOCKING SYNCHRONIZATION

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

(a) Using CAS  (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 \cite{Moir:97,AndersonMoir:99,JayantiPetrovic:03,Dohertyetal:04,JayantiPetrovic:05}.

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 \( \text{CAS}(x, A, B) \) and later \( \text{CAS}(x, B, A) \). Now if \( p \) performs \( \text{CAS}(x, A, Z) \) 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 \( \langle A, 0 \rangle \), then \( q \) performs \( \text{CAS}(x, \langle A, 0 \rangle, \langle B, 1 \rangle) \) and \( \text{CAS}(x, \langle B, 1 \rangle, \langle A, 2 \rangle) \). Now \( p \)'s \( \text{CAS}(x, \langle A, 0 \rangle, \langle Z, 1 \rangle) \) 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 \cite{Moir:97}.

### 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
2.6. ALGORITHMS

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. 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 non-blocking 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.
CHAPTER 2. NONBLOCKING DATA STRUCTURES

Type: 
\[ Node = \{ val : T; \text{next} : \text{Node} \} \]

Shared: 
\[ \text{Head} : \text{Node} := \text{null} \]

1: \textbf{operation} \texttt{PUSH}(lv:T) 
2: \hspace{1em} \texttt{n} := \texttt{new}\,(\text{Node}) 
3: \hspace{1em} \text{\texttt{n.val} := lv} 
4: \hspace{1em} \textbf{repeat} 
5: \hspace{2em} \text{ss} := \text{Head} 
6: \hspace{2em} \text{\texttt{n.next} := ss} 
7: \hspace{1em} \textbf{until} \text{CAS}(\text{Head}, ss, n) 
8: \hspace{1em} \textbf{end operation} 

9: \textbf{operation} \texttt{POP}() 
10: \hspace{1em} \textbf{repeat} 
11: \hspace{2em} \text{ss} := \text{Head} 
12: \hspace{2em} \textbf{if} \text{ss} = \text{null} \textbf{then} 
13: \hspace{3em} \textbf{return} \text{empty} 
14: \hspace{2em} \textbf{end if} 
15: \hspace{2em} \text{\texttt{ssnext} := ss.next} 
16: \hspace{2em} \text{lv} := \text{ss.val} 
17: \hspace{2em} \textbf{until} \text{CAS}(\text{Head}, ss, ssnext) 
18: \hspace{2em} \text{\texttt{ss.next} := null} 
19: \hspace{2em} \text{\texttt{ss.val} := null} 
20: \hspace{2em} \textbf{return} \text{lv} 
21: \hspace{1em} \textbf{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 \texttt{val} field and a \texttt{next} field pointing to another node. A shared \texttt{Head} variable points to the first element when the stack is non-empty, and is \texttt{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 \texttt{n} and sets its value. It then takes a “snapshot” of \texttt{Head} and points \texttt{n}'s \texttt{next} field at the snapshot. A CAS operation is used to ensure that \texttt{Head} is updated to point to \texttt{n} only if it has not been modified. If \texttt{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 \texttt{Head} and tests to see if the snapshot is \texttt{null}; if so it returns “empty”. Otherwise it takes a snapshot of this node’s \texttt{next} field and records the value in the \texttt{val} field. As for push, a CAS is used to detect a conflict with another successful operation —

\footnote{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: \( \text{Node} = \{ \text{val} : T; \text{next} : \text{Node} \} \)

Shared: \( \text{Head} : \text{Node} := \text{new}(\text{Node}) \)

Shared: \( \text{Tail} : \text{Node} := \text{Head} \)

1: operation \( \text{ENQUEUE}(lv:T) \)
2: \( n := \text{new}(\text{Node}) \)
3: \( n.val := lv \)
4: loop
5: \( sstail := \text{Tail} \)
6: \( ssnext := sstail.next \)
7: if \( sstail = \text{Tail} \) then
8: if \( ssnext = \text{null} \) then
9: if \( \text{CAS}(sstail.next, ssnext, n) \) then
10: break
11: end if
12: else
13: \( \text{CAS}(\text{Tail}, sstail, ssnext) \)
14: end if
15: end if
16: end loop
17: \( \text{CAS}(\text{Tail}, sstail, n) \)
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 \( \text{Head} \) and \( \text{Tail} \) variables, and both are initialised to a new dummy node; a queue of \( n \) elements is represented with a list of \( n + 1 \) nodes. \( \text{Head} \) always points to the dummy node at the beginning of the list and its \( \text{next} \) field points to the node that holds the first element in the queue; thus the queue is empty iff \( \text{Head}.next = \text{null} \). An enqueue operation contains two distinct atomic updates — “add a new node to the end of the list” and “make \( \text{Tail} \) point to the new node”. Because of this, \( \text{Tail} \) 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 enqueueing 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
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
2.6. ALGORITHMS

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; Béard 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 $M$, representing a system is a model of a temporal logic formula $\phi$, representing a specification of the system, in other words that $M$ satisfies $\phi$, written $M \models \phi$. 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.
CHAPTER 3.  MODEL CHECKING

\[
\begin{align*}
  x &= 0 & x &= 1 & x &= 2 \\
  y &= 2 & y &= 1 & y &= 0 \\
\end{align*}
\]

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 \( \mathbf{AP} \) of atomic propositions by a tuple \( \langle S, \text{Init}, R, L \rangle \), where \( S \) is a set of states, \( \text{Init} \subseteq S \) is a set of initial states, \( R \subseteq S \times S \) is a transition relation, and \( L : S \to 2^{\mathbf{AP}} \) is a labelling function, or interpretation, that assigns all the propositions that are true at each state. For model checking purposes, both \( S \) and \( \mathbf{AP} \) 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 \( \text{var} = \text{num} \) 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.
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.\footnote{For an overview of the development of these three logics see the historical surveys of Goldblatt [2003, 2006 §7.3] and Vardi [2008a,b].}

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

- \( \text{X} \) (“next”) requires a property to hold in the second state of the path;
- \( \text{F} \) (“eventually” or “in the future”) requires a property to hold in some state of the path;
- \( \text{G} \) (“always” or “globally”) requires a property to hold in every state of the path;
- \( \text{U} \) (“until”) is a binary operator that requires the first property to hold in every state until the second holds;
- \( \text{R} \) (“release”) is the logical dual of \( \text{U} \) 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 \( p \in \text{AP} \), then \( p \) is a state formula.
- If \( \varphi \) and \( \psi \) are state formulas, then \( \neg \varphi \), \( \varphi \land \psi \), \( \varphi \lor \psi \) and \( \varphi \rightarrow \psi \) are state formulas.
- If \( \varphi \) is a path formula, then \( \text{A} \varphi \) and \( \text{E} \varphi \) are state formulas.
- If \( \varphi \) is a state formula, then \( \varphi \) is a path formula also.
- If \( \varphi \) and \( \psi \) are path formulas, then \( \neg \varphi \), \( \varphi \land \psi \), \( \varphi \lor \psi \), \( \varphi \rightarrow \psi \), \( \text{X} \varphi \), \( \text{F} \varphi \), \( \text{G} \varphi \), \( \varphi \text{ U} \psi \) and \( \varphi \text{ R} \psi \) 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 \( \langle S, \text{Init}, R, L \rangle \) where \( R \) is total. A path in \( M \) is a sequence of states \( \pi = s_0, s_1, s_2, \ldots \) such that for every \( i \geq 0 \), \( \langle s_i, s_{i+1} \rangle \in R \). This corresponds to an infinite branch in the computation tree of the Kripke structure. The \( i \)-th suffix of a path, \( \pi^i \), is the path that starts from the \( i \)-th state of \( \pi \). For a state formula \( \varphi \), the notation \( M, s \models \varphi \) means that \( \varphi \) holds at state \( s \) in \( M \). For a path formula \( \varphi \), the notation \( M, \pi \models \varphi \) means that \( \varphi \) holds along the
3.2. SPECIFICATIONS

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

$\mathcal{M}, s \models p$ iff $p \in L(s)$
$\mathcal{M}, s \models \neg \varphi$ iff $\mathcal{M}, s \not\models \varphi$
$\mathcal{M}, s \models \varphi \land \varphi'$ iff $\mathcal{M}, s \models \varphi$ and $\mathcal{M}, s \models \varphi'$
$\mathcal{M}, s \models E \psi$ iff $\exists \pi = s, \ldots \bullet \mathcal{M}, \pi \models \psi$
$\mathcal{M}, s \models A \psi$ iff $\forall \pi = s, \ldots \bullet \mathcal{M}, \pi \models \psi$
$\mathcal{M}, \pi \models \varphi$ iff $\mathcal{M}, s \models \varphi$ where $\pi = s, \ldots$
$\mathcal{M}, \pi \models \neg \psi$ iff $\mathcal{M}, \pi \not\models \psi$
$\mathcal{M}, \pi \models \varphi \land \varphi'$ iff $\mathcal{M}, s \models \varphi$ and $\mathcal{M}, \pi \models \psi'$
$\mathcal{M}, \pi \models X \psi$ iff $\exists i \geq 0 \bullet \mathcal{M}, \pi^i \models \psi$
$\mathcal{M}, \pi \models F \psi$ iff $\exists k \geq 0 \bullet \mathcal{M}, \pi^k \models \psi'$ and $\forall 0 \leq j < k \bullet \mathcal{M}, \pi^j \models \psi$
$\mathcal{M}, \pi \models \psi R \psi'$ iff $\forall j \geq 0 \bullet$ if $\mathcal{M}, \pi^j \not\models \psi$ for every $i < j$
then $\mathcal{M}, \pi^j \models \psi'$

The semantics of $\lor$ and $\to$ are given by their standard definitions:

$\varphi \lor \psi \equiv \neg (\neg \varphi \land \neg \psi)$
$\varphi \to \psi \equiv \neg \varphi \lor \psi$

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

$\varphi R \psi \equiv \neg (\neg \varphi U \neg \psi)$
$F \varphi \equiv \text{true} U \varphi$
$G \varphi \equiv \neg F \neg \varphi$
$A \varphi \equiv \neg E \neg \varphi$

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

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

$\text{ACTL}^* = \{ \varphi \mid \varphi \text{ does not contain } E \text{ and is in NNF} \}$
$\text{ECTL}^* = \{ \varphi \mid \varphi \text{ does not contain } A \text{ and is in NNF} \}$

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 $\neg A \varphi = E \neg \varphi$ 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 $A\varphi$, where $\varphi$ 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

$$
\text{LTL} ::= A\varphi \\
\varphi ::= AP | \neg \varphi | \varphi \land \varphi | \varphi \lor \varphi | \varphi \rightarrow \varphi | X\varphi | F\varphi | G\varphi | \varphi U \varphi | \varphi R \varphi
$$

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

$$
\varphi ::= AP | \neg \varphi | \varphi \land \varphi | \varphi \lor \varphi | \varphi \rightarrow \varphi | AX\varphi | EX\varphi | AF\varphi | EF\varphi | AG\varphi | EG\varphi | A[\varphi U \varphi] | E[\varphi U \varphi] | A[\varphi R \varphi] | E[\varphi R \varphi]
$$

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 Dragnicescu, 1988; Vardi, 1998a, 2001; Nain and Vardi, ...]
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 $n^2O(m)$ [Lichtenstein and Pnueli, 1985]. 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 $\text{AG(EF}p\text{)}$, 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 $\text{AFG}q$ 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”.

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. 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 \( A \) over finite words is defined by a tuple \( (\Sigma, Q, \Delta, Q_0, F) \), where \( \Sigma \) 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 \( A \) on \( v \) is a mapping \( \rho : \{0, 1, \ldots, |v|\} \rightarrow Q \) such that \( \rho(0) \in Q_0 \) and, for all \( 0 \leq i < |v|, (\rho(i), v(i), \rho(i + 1)) \in \Delta \). A run is accepting if \( \rho(|v|) \in F \), i.e. it ends at an accepting state, and an automaton \( A \) accepts a word \( v \) if, and only if, there exists an accepting run of \( A \) on \( v \). The language of \( A \), \( L(A) \subseteq \Sigma^* \), consists of all the words accepted by \( A \) [Clarke et al., 1999].

A Büchi automaton has the same components as a finite automaton; a run of a Büchi automaton \( A \) 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 \( M = (S, Init, R, L) \) can be represented by a Büchi automaton \( A_M = (\Sigma, S \cup \{i\}, \Delta, \{i\}, S \cup \{i\}) \) where

- \( \Sigma = 2^{\text{AP}} \),
- \(<s, \alpha, s'> \in \Delta \iff (s, s') \in R \) and \( \alpha = L(s') \), and
- \(<i, \alpha, s > \in \Delta \iff s \in \text{Init} \) 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].
3.3. MODEL CHECKING PARADIGMS

(a) Kripke structure

(b) Büchi automaton

Figure 3.3: Two equivalent system representations

Figure 3.4: A Büchi automaton for $\mathbf{AG}p$
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 $AGp$. The model checking problem $M \models \varphi$ can be solved by the language inclusion problem $L(A_M) \subseteq L(A_\varphi)$, which is equivalent to checking whether $L(A_M) \cap L(A_{\neg \varphi}) = \emptyset$. Complementing Büchi automaton is a complicated procedure, so it is common to construct $A_{\neg \varphi}$ instead of $A_\varphi$ [Clarke et al., 1999].

It is possible to construct the intersection automaton that accepts the language $L(A_M) \cap L(A_{\neg \varphi})$ 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
3.3. MODEL CHECKING PARADIGMS

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 \((a_1 \leftrightarrow b_1) \land (a_2 \leftrightarrow b_2)\).

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
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 \( \text{AG} \varphi \) has the form

\[
I(s_0) \land T(s_0, s_1) \land T(s_1, s_2) \land T(s_2, s_3) \land
(\neg \varphi(s_0) \lor \neg \varphi(s_1) \lor \neg \varphi(s_2) \lor \neg \varphi(s_3))
\]

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 \( \varphi \) 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. 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...
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 Tretler, 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 $M$ satisfies a (concrete) specification $\varphi$, “$M \models \varphi$?”, we construct an abstract model $M_A$ and an abstract specification $\varphi_A$ and ask “$M_A \models \varphi_A$?” 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 \{negative, zero, 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 Saidi, 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.

$$M_A \models \varphi_A \Leftrightarrow M \models \varphi$$

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.

$$M_A \models \varphi_A \Rightarrow M \models \varphi$$
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 often 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, $\text{Lit} = \text{AP} \cup \{ \neg p \mid p \in \text{AP} \}$. The Kripke structures are the same, except that the labelling function $L : S \rightarrow 2^{\text{Lit}}$ is required to satisfy

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

Note that it is possible that neither $p$ nor $\neg p$ at $s$. 
A Galois connection from a partially ordered set \( \langle C, \leq_C \rangle \) to another \( \langle A, \leq_A \rangle \) is a pair of functions \( \alpha : C \to A \) (the abstraction function) and \( \gamma : A \to C \) (the concretisation function) such that

- \( \alpha \) and \( \gamma \) are total and monotonic,
- for all \( c \in C \), \( \gamma(\alpha(c)) \geq_C c \), and
- for all \( a \in A \), \( \alpha(\gamma(a)) \leq_A a \).

Additionally, if \( \leq_A \) is defined by \( \leq_C \) such that

\[ a \leq_A a' \iff \gamma(a) \leq_C \gamma(a') \]

then \( \langle \alpha, \gamma \rangle \) is a Galois insertion.

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

\[ \hat{S}_0 = \{ \alpha(\{s\}) \mid s \in S_0 \} \]

and the labelling function is defined as

\[ p \in \hat{L}(a) \iff \forall s \in \gamma(a) \cdot p \in L(s) \]

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

\[ R^{\forall \exists} = \{ (X,Y) \mid \forall x \in X \cdot \exists y \in Y \cdot x R y \} \]
\[ R^{\exists \exists} = \{ (X,Y) \mid \exists x \in X \cdot \exists y \in Y \cdot x R y \} \]

These relations can be used to define abstract systems that preserve different properties. The Kripke structure \( \langle \hat{S}, \hat{S}_0, \hat{R}^E, \hat{L} \rangle \) preserves the logic ACTL* and the Kripke structure \( \langle \hat{S}, \hat{S}_0, \hat{R}^A, \hat{L} \rangle \) preserves the logic ECTL*, where the transition relations are defined as

\[ a \hat{R}^E b \iff b \in \{ \alpha(\{Y\}) \mid Y \in \min \{ Y' \mid \gamma(a) R^{\forall \exists} Y' \} \} \]
\[ a \hat{R}^A b \iff b \in \{ \alpha(\{Y\}) \mid Y \in \min \{ Y' \mid \gamma(a) R^{\exists \exists} Y' \} \} \]
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 \( \mathcal{M} = \langle \hat{S}, \hat{S}_0, \hat{R}^A, \hat{R}^E, L \rangle \) and has two types of paths — A-paths, defined along \( \hat{R}^A \) transitions, and E-paths, defined along \( \hat{R}^E \) transitions. Finally, we modify the semantics of CTL* from Section 3.2.1 so that

\[
\begin{align*}
\mathcal{M}, s \models E\psi & \text{ iff } \mathcal{M}, \pi \models \psi \text{ for some E-path } \pi \\
\mathcal{M}, s \models A\psi & \text{ iff } \mathcal{M}, \pi \models \psi \text{ for every A-path } \pi
\end{align*}
\]

### 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 [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 [Promela was originally an acronym for PROcess MEta LAnguage; Spin was an acronym of Simple Promela INterpreter.](http://www.spinroot.com) is procedural, with visual similarity with C, and is largely based on Dijkstra’s guarded command language. It can check \( \omega \)-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]
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, 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 (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.

http://sal.csl.sri.com
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 approach for 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
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 \( \mathcal{P} = \{ \text{eq}, p_1, \ldots, p_n \} \) 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 \( \text{eq} \) has arity 2). Then, a concrete configuration \( S^\prime = \langle U^\prime, \iota^\prime \rangle \) has a universe \( U^\prime \) that is a (finite or infinite) set of objects and an interpretation \( \iota^\prime \) over the logical values true (1) and false (0). For each \( k \)-ary predicate \( p \),

\[
\iota^\prime(p) : (U^\prime)^k \rightarrow \{0, 1\}
\]

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

**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 \( \text{node} \) to indicate the type of Node objects, a unary predicate \( x \) to indicate which object is pointed to by \( x \), and a binary predicate \( \text{next} \) to represent the relationship of the next field.

\[
\mathcal{P} = \{ \text{eq}, \text{node}, x, \text{next} \}
\]

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...
4.1. CANONICAL ABSTRACTION

follows, where in each case $1 \leq i, j \leq 3$.

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

$$\iota^\sharp(\text{eq}) = \{\langle u_i, u_j \rangle \mapsto 1 | i = j\} \cup \{\langle u_i, u_j \rangle \mapsto 0 | i \neq j\}$$

$$\iota^\sharp(\text{node}) = \{u_i \mapsto 1\}$$

$$\iota^\sharp(\text{x}) = \{u_1 \mapsto 1, u_2 \mapsto 0, u_3 \mapsto 0\}$$

$$\iota^\sharp(\text{next}) = \{\langle u_1, u_2 \rangle \mapsto 1, \langle u_2, u_3 \rangle \mapsto 1\} \cup \{\langle u_i, u_j \rangle \mapsto 0 | \langle i, j \rangle \not\in \{(1, 2), (2, 3)\}\}$$

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;
- 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 $\text{eq}$ 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 $p$,

$$\iota(p) : U^k \rightarrow \{1, 0, \frac{1}{2}\}$$

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

\footnote{The names $u_i$ have no meaning other than for ease of identification.}

\footnote{We will generally say “objects” of the graph, to avoid confusion with the nodes of linked lists being represented.}
An object \( u \), for which \( \iota(eq)(u, u) \) 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 \( \iota(eq)(u, u) = \frac{1}{2} \)) have a double line,
- unary predicates with an unknown interpretation have an addition to the label (e.g. "\( x = \frac{1}{2} \)"), 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) 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 \( u_1, \ldots, u_k \in U_1 \),

\[
\iota_1(p)(u_1, \ldots, u_k) \sqsubseteq \iota_2(p)(f(u_1), \ldots, f(u_k))
\]

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 \texttt{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

---

3Since 2-valued configurations are trivially 3-valued configurations also, we will assume that configurations are 3-valued unless otherwise noted.
4.1. CANONICAL ABSTRACTION

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:

\[ \iota(\text{eq}) = \begin{cases} 
\langle u_{\{1\}}, u_{\{1\}} \rangle \mapsto 1, & \langle u_{\{2,3\}}, u_{\{2,3\}} \rangle \mapsto \frac{1}{2}, \\
\langle u_{\{1\}}, u_{\{2,3\}} \rangle \mapsto 0, & \langle u_{\{2,3\}}, u_{\{1\}} \rangle \mapsto 0 \end{cases} \]

\[ \iota(\text{node}) = \begin{cases} 
u_{\{1\}} \mapsto 1, & \nu_{\{2,3\}} \mapsto 1 \end{cases} \]

\[ \iota(\text{x}) = \begin{cases} 
u_{\{1\}} \mapsto 1, & \nu_{\{2,3\}} \mapsto 0 \end{cases} \]

\[ \iota(\text{next}) = \begin{cases} 
\langle u_{\{1\}}, u_{\{2,3\}} \rangle \mapsto \frac{1}{2}, & \langle u_{\{2,3\}}, u_{\{2,3\}} \rangle \mapsto \frac{1}{2}, \\
\langle u_{\{1\}}, u_{\{1\}} \rangle \mapsto 0, & \langle u_{\{2,3\}}, u_{\{1\}} \rangle \mapsto 0 \end{cases} \]

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 \( p \), \( \iota'(p)(u_0, \ldots, u_0) = \frac{1}{2} \).

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 \( u_1, \ldots, u_k \in U_2 \),

\[
i_2(p)(u_1, \ldots, u_k) = \begin{cases} 
1 & \text{if } \forall u'_1 \in f^{-1}(u_1), \ldots, u'_k \in f^{-1}(u_k) \bullet \ \ i_1(p)(u'_1, \ldots, u'_k) = 1 \\
0 & \text{if } \forall u'_1 \in f^{-1}(u_1), \ldots, u'_k \in f^{-1}(u_k) \bullet \ \ i_1(p)(u'_1, \ldots, u'_k) = 0 \\
\frac{1}{2} & \text{otherwise}
\end{cases}
\]

**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

\[
i(\text{next})(u_{\{2,3\}}, u_{\{1\}}) = \frac{1}{2}
\]

then the information loss would not be minimal, as both

\[
i^\natural(\text{next})(u_2, u_1) = 0 \\
i^\natural(\text{next})(u_3, u_1) = 0
\]

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 \( A \subseteq P_1 \), 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 \( A \), if it is the canonical abstraction of itself.

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

\[
f = \{ u \mapsto u_A \mid u \in U \land u_A \in U' \land \\
(\forall p \in A \bullet i(p)(u) = 1) \land \\
(\forall p \in A - A \bullet i(p)(u) \neq 1) \}
\]
Example  The abstract configuration in Figure 4.3 is the canonical abstraction of the concrete configuration in Figure 4.2 with $A = P_1$. For both $u_2, u_3 \in U^3$, $t^\sharp (\text{node}) (\_ ) = 1$ and $t^\sharp (x) (\_ ) = 0$ so they are both mapped to $u_{\{2,3\}} \in U$. The object $u_1 \in U^2$ differs, as $t^\sharp (x) (u_1) = 1$, 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 $u_{\{\text{node}, x\}}$ and $u_{\{\text{node}\}}$, 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 $\alpha$ 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 $P$ and set of variables $Var$ inductively as follows:

- The logical literals 0 and 1 are atomic formulas with no free variables.

- For every predicate $p \in P^k$, $p(v_1, \ldots, v_k)$ is an atomic formula with free variables $\{v_1, \ldots, v_k\} \subseteq Var$. 
• If \( \varphi_1 \) and \( \varphi_2 \) are formulas with sets of free variables \( V_1 \subseteq \text{Var} \) and \( V_2 \subseteq \text{Var} \) respectively, then \( \varphi_1 \land \varphi_2, \varphi_1 \lor \varphi_2 \) and \( \neg \varphi_1 \) are formulas with free variables \( V_1 \cup V_2, V_1 \cup V_2 \) and \( V_1 \) respectively.

• If \( \varphi_1 \) is a formula with free variables \( \{v_1, \ldots, v_k\} \subseteq \text{Var} \), 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 \( \varphi_1 \) is a formula with free variables \( V \subseteq \text{Var} \), such that \( v_1, v_2 \in V \) and \( v_3, v_4 \not\in V \), then \((\text{TC} v_1, v_2 \bullet \varphi_1)(v_3, v_4)\) 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 \( S = \langle U, i \rangle \) 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 \( \varphi \), relative to \( S \) and \( Z \), is written \( \mathcal{J}^S_3(Z) \), and is defined inductively below.

Atomic formulas If \( \varphi = l \in \{1, 0\} \), then
\[
[\langle l \rangle]^S_3(Z) = l
\]
If \( \varphi = p(v_1, \ldots, v_k) \), for some \( p \in \mathcal{P}_k \), then
\[
[p(v_1, \ldots, v_k)]^S_3(Z) = i(p)(Z(v_1), \ldots, Z(v_k))
\]

Logical connectives If \( \varphi \) is built from subformulas \( \varphi_1 \) and \( \varphi_2 \) then
\[
[\varphi_1 \land \varphi_2]^S_3(Z) = \min([\varphi_1]^S_3(Z), [\varphi_2]^S_3(Z))
\]
\[
[\varphi_1 \lor \varphi_2]^S_3(Z) = \max([\varphi_1]^S_3(Z), [\varphi_2]^S_3(Z))
\]
\[
[\neg \varphi_1]^S_3(Z) = 1 - [\varphi_1]^S_3(Z)
\]

Quantifiers If \( \varphi \) is built from subformula \( \varphi_1 \) then
\[
[\forall v_1 \bullet \varphi_1]^S_3(Z) = \min_{u \in U} [\varphi_1]^S_3(Z[v_1 \mapsto u])
\]
\[
[\exists v_1 \bullet \varphi_1]^S_3(Z) = \max_{u \in U} [\varphi_1]^S_3(Z[v_1 \mapsto u])
\]

\footnote{The notation \( Z[v_1 \mapsto u] \) is an assignment update, i.e. \( \{v_i \mapsto u' \mid j \neq i \land Z(v_j) = u'\} \cup \{v_i \mapsto u\} \).}
4.2. REFINING ABSTRACTIONS

Transitive closure If $\varphi$ is built from subformula $\varphi_1$ then

$$\left[ (\text{TC } v_1, v_2 \cdot \varphi_1)(v_3, v_4) \right]^S_3(Z) = \max_{n \geq 1; u_1, \ldots, u_{n+1} \in U; Z(v_3) = u_1; Z(v_4) = u_{n+1}} \min_{i=1}^n \left[ \varphi_1 \right]^S_3(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])$$

We say that $S$ and $Z$ potentially satisfy $\varphi$, written $S, Z \models \varphi$, if $\left[ \varphi \right]^S_3(Z) = 1$ or $\left[ \varphi \right]^S_3(Z) = \frac{1}{2}$, 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:

$$\varphi_1 \rightarrow \varphi_2 \equiv \neg \varphi_1 \lor \varphi_2$$

$$p^+(v_3, v_4) \equiv (\text{TC } v_1, v_2 \cdot p(v_1, v_2))(v_3, v_4)$$

$$p^*(v_3, v_4) \equiv eq(v_3, v_4) \lor p^+(v_3, v_4)$$

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 $\varphi$ is a conservative approximation of the information extracted from $S$ via $\varphi$. Alternatively, if we prove a property $\varphi$ 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 $S = \langle U, i \rangle$ and $S' = \langle U', i' \rangle$ be structures, and $f : U \rightarrow U'$ a function that embeds $S$ into $S'$. Then, for every formula $\varphi$ and assignment $Z$,

$$\left[ \varphi \right]^S_3(Z) \subseteq \left[ \varphi \right]^{S'}_3(f \circ Z)$$

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 \( \text{next} \) represent a global variable and a field, respectively. As such, we expect various properties to exist, for example that \( x \) is unique, and that \( \text{next} \) is functional. We can see though that the functional property is lost, as the formula

\[
\forall v_1, v_2, v_3 \cdot \text{next}(v_1, v_2) \land \text{next}(v_1, v_3) \rightarrow \text{eq}(v_2, v_3)
\]

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 \( \text{next} \) is not functional and has the same canonical abstraction \( S \) as in Figure 4.3. Three objects are mapped to the canonical object \( u\{\text{node}\} \) so it is a summary object, but only one is mapped to \( u\{x, \text{node}\} \) so it is a non-summary object. Now

\[
i(\text{next}(u\{x, \text{node}\}, u\{x, \text{node}\})) = i(\text{next}(u\{\text{node}\}, u\{x, \text{node}\})) = 0
\]

because \( \text{next} \) is false for each of the relevant concrete pairings, and

\[
i(\text{next}(u\{x, \text{node}\}, u\{\text{node}\})) = i(\text{next}(u\{\text{node}\}, u\{\text{node}\})) = 1/2
\]

because \( \text{next} \) is true for some of the relevant concrete pairings and false for others. Clearly, this situation will happen whenever \( \text{next} \) 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:

\[
\gamma[F](S) = \{ S^\natural \mid \alpha(S^\natural) = S \land \forall \varphi \in F \cdot S^\natural \models \varphi \}
\]

Table 4.1 gives some example integrity rules that may be used to enforce properties of a predicate \( p \), including uniqueness and functionality.
4.2. 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 $S^2_1$, and Figure 4.3 which we will call $S_1$. The concrete configuration $S^2_1$ represents a list of length 3, and the abstract configuration $S_1$ represents all lists of length 3 or more.\footnote{In Section 4.1.2 we said that lists of length 2 or more embed into $S_1$. Here we are considering a more refined abstraction.} Connectedness and circularity are
uniqueness: $\forall v_1, v_2 \cdot p(v_1) \land p(v_2) \rightarrow eq(v_1, v_2)$

functionality: $\forall v_1, v_2, v_3 \cdot p(v_1, v_2) \land p(v_1, v_3) \rightarrow eq(v_2, v_3)$

inverse functionality: $\forall v_1, v_2, v_3 \cdot p(v_1, v_3) \land p(v_2, v_3) \rightarrow eq(v_1, v_2)$

symmetry: $\forall v_1, v_2 \cdot p(v_1, v_2) \rightarrow p(v_2, v_1)$

Table 4.1: Example integrity rules

often important properties to consider when analysing lists, so since $S^2_1$ 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:

$$\left[ \forall v_2 \cdot \exists v_1 \cdot x(v_1) \land next^+(v_1, v_2) \right]^{S_1^3} = 1$$

$$\left[ \forall v_2 \cdot \exists v_1 \cdot x(v_1) \land next^+(v_1, v_2) \right]^{S_1^2} = \frac{1}{2}$$

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

$$\left[ \forall v_1 \cdot \neg next^+(v_1, v_1) \right]^{S_1^3} = 1$$

$$\left[ \forall v_1 \cdot \neg next^+(v_1, v_1) \right]^{S_1^2} = \frac{1}{2}$$

Indeed, we can see in Figure 4.5 that $S_1$ is also the canonical abstraction of an unconnected acyclic list ($S^2_2$), and a connected cyclic list ($S^3_3$).

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 $S_1$. 

```latex
\[
\left[ \forall v_2 \cdot \exists v_1 \cdot x(v_1) \land next^+(v_1, v_2) \right]^{S_1^3} = 1
\]
\[
\left[ \forall v_2 \cdot \exists v_1 \cdot x(v_1) \land next^+(v_1, v_2) \right]^{S_1^2} = \frac{1}{2}
\]
\[
\left[ \forall v_1 \cdot \neg next^+(v_1, v_1) \right]^{S_1^3} = 1
\]
\[
\left[ \forall v_1 \cdot \neg next^+(v_1, v_1) \right]^{S_1^2} = \frac{1}{2}
\]

Indeed, we can see in Figure 4.5 that $S_1$ is also the canonical abstraction of an unconnected acyclic list ($S^2_2$), and a connected cyclic list ($S^3_3$). □
```
Figure 4.5: Three different lists have the same canonical abstraction
Figure 4.6: Using instrumentation predicates to distinguish three different list structures

to evaluate definitely:

\[
\text{reach}[x, next](v) = \exists u \cdot x(u) \land \text{next}^+(u, v) \\
\text{circ}[next](v) = \text{next}^+(v, v)
\]

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 \(\text{reach}[x, next]\) is true for all nodes, we know that the list is connected. Also, since \(\text{circ}[next]\) is false for all nodes, we know that the list is connected.

---

\(^6\)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 \(\text{reach}[y, next]\) and \(\text{reach}[z, next]\) at the same time.)
4.2. REFINING ABSTRACTIONS

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:

$$[\forall v_2 \cdot \exists v_1 \cdot x(v_1) \land \text{next}^*(v_1, v_2)]^{S'_1}_{S_1} = \frac{1}{2}$$
$$[\forall v_1 \cdot \neg \text{next}^+(v_1, v_1)]^{S'_1}_{S_1} = \frac{1}{2}$$

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

\[
\begin{align*}
[\forall v_2 \cdot \text{reach}\{x,\text{next}\}(v_2)]_{S_3'} &= 1 \\
[\forall v_1 \cdot \neg \text{circ}\{\text{next}\}(v_1)]_{S_3'} &= 1
\end{align*}
\]

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.
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. 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 \( \tau \) between two states \( S_1 \) and \( S_2 \) when each has a state in its concretisation \( (S'_1, S'_2) \), respectively, that are related by the concrete transition \( \tau \), i.e.

\[
S'_1 \overset{\tau}{\longrightarrow} S'_2
\]

\[
\overset{\gamma}{\longrightarrow} \quad \overset{\gamma}{\longrightarrow} \quad \text{implies} \quad S_1 \overset{\tau}{\rightarrow} S_2
\]

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

---

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 \cite{Sagiv2002, §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 \cite{Cousot1979}, 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 \cite{Sagiv2002, 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 \( \Phi \) (the focus formulas), it returns the minimum set of abstract configurations that together represent the same set of concrete...
4.3. ABSTRACT TRANSITIONS

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 \( Z \):

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

\[\text{Sagiv et al. [2002, §6.3.1]}\] give an algorithm for a restricted class of formulas. \[\text{Lev-Ami [2000, §6]}\] 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 \[\text{Sagiv et al. [2002, §6.3.2]}\]. 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: \( \text{Focus}(p(v_1, v_2)) \)
(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:

$$\varphi_1 \triangleright \varphi_2$$

where the head $$\varphi_2$$ is an atomic FOTC formula (a literal or predicate) or the negation of an atomic FOTC formula, and the body $$\varphi_1$$ is an arbitrary FOTC formula. The novel operator $$\triangleright$$ is defined semantically as follows:

$$S, Z \models \varphi_1 \triangleright \varphi_2$$ iff $$[\varphi_1]_S^Z = 1$$ implies $$[\varphi_2]_S^Z = 1$$

In 2-valued configurations, $$\varphi_1 \triangleright \varphi_2$$ has exactly the same meaning as $$\varphi_1 \rightarrow \varphi_2$$. In 3-valued configurations though it has a stronger meaning; if $$[\varphi_1]_S^Z = 1$$ and $$[\varphi_2]_S^Z = \frac{1}{2}$$ then $$\varphi_1 \rightarrow \varphi_2$$ is satisfied but $$\varphi_1 \triangleright \varphi_2$$ 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 $$\varphi_p$$ is its defining formula:

$$\forall v_1 \cdots v_k \bullet \varphi_p \rightarrow p(v_1, \ldots, v_k)$$
$$\forall v_1 \cdots v_k \bullet \neg \varphi_p \rightarrow \neg p(v_1, \ldots, v_k)$$

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

$$\varphi_1 \triangleright a$$ if $$\varphi = \forall v_1 \cdots v_k \bullet \varphi_1 \rightarrow a$$
$$\neg \varphi \triangleright 0$$ otherwise

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

$$a_1 \land \cdots \land a_n \triangleright a_0$$
where each $a_i$ is an atomic formula or the negation of an atomic formula, we also include the following constraints, for all $1 \leq i \leq n$:

$$\neg a_0 \land a_1 \land \cdots \land a_{i-1} \land a_{i+1} \land \cdots \land a_n \lor \neg a_i$$

### 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 $\psi_{pre}$ and the set of update formulas $\Psi$. The precondition $\psi_{pre}$ can be any FOTC formula. The set $\Psi$ contains a formula of the form $p(v_1, \ldots, v_k) = \varphi_p$ for every predicate $p$, where $k$ is the arity of $p$ and $\varphi_p$ 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) $S_1^2$ to $S_2^2$ iff their universes are the same, i.e. $U_1^2 = U_2^2$, there is an assignment that makes the precondition true in the first state $S_1^2$, i.e.

$$\exists Z \bullet [\psi_{pre}]_2^{S_1^2}(Z) \neq 0$$

---

8The 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.
4.3. ABSTRACT TRANSITIONS

and every predicate has the same value in the second state $S^2_2$ as its update formula has in the first, i.e.

$$\forall p \cdot \forall Z \cdot [p(v_1, \ldots, v_k)]^S_2(Z) = [\varphi_p]^S_1(Z)$$

During the construction 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 $\tau$ hold between abstract states (3-valued configurations) $S_1$ and $S_2$ iff

$$U_1 = U_2$$
$$\exists Z \cdot [\psi_{pre}]^S_1(Z) \neq 0$$
$$\forall p \cdot \forall Z \cdot [p(v_1, \ldots, v_k)]^S_2(Z) = [\varphi_p]^S_1(Z)$$

**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 $is\_New(v) = 0$.

**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.

$$x := x\_next$$
CHAPTER 4. CANONICAL ABSTRACTION

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:

\[ C_1 = \{x\} \]
\[ C_2 = \{\text{next}\} \]
\[ I_1 = \{\text{circ}[\text{next}], \text{reach}[x, \text{next}]\} \]

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

\[ x(v_1) \land x(v_2) \models \text{eq}(v_1, v_2) \] (C.1)
\[ \exists v_2 \models \neg \text{eq}(v_1, v_2) \land x(v_2) \models \neg x(v_1) \] (C.2)
\[ \exists v_1 \models \text{next}(v_1, v_2) \land \text{next}(v_1, v_3) \models \text{eq}(v_2, v_3) \] (C.3)
\[ \exists v_3 \models \neg \text{eq}(v_2, v_3) \land \text{next}(v_1, v_3) \models \neg \text{next}(v_1, v_2) \] (C.4)

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

\[ \text{next}^+(v_1, v_1) \models \text{circ}[\text{next}](v_1) \] (C.5)
\[ \neg \text{next}^+(v_1, v_1) \models \neg \text{circ}[\text{next}](v_1) \] (C.6)
\[ \exists v_1 \models x(v_1) \land \text{next}^*(v_1, v_2) \models \text{reach}[x, \text{next}](v_2) \] (C.7)
\[ \forall v_1 \models \neg x(v_1) \lor \neg \text{next}^*(v_1, v_2) \models \neg \text{reach}[x, \text{next}](v_2) \] (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:

\[
U = \{u_1, u_2\}
\]

<table>
<thead>
<tr>
<th></th>
<th>(u_1)</th>
<th>(u_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(\text{circ[next]})</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(\text{reach[x,next]})</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(\text{next}) (u_1)</td>
<td>0</td>
<td>(\frac{1}{2})</td>
</tr>
<tr>
<td>(\text{next}) (u_2)</td>
<td>0</td>
<td>(\frac{1}{2})</td>
</tr>
<tr>
<td>(\text{eq}) (u_1)</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(\text{eq}) (u_2)</td>
<td>0</td>
<td>(\frac{1}{2})</td>
</tr>
</tbody>
</table>

**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

\(x(v_1) \land \text{next}(v_1, v_2)\)

Now \([x(u_2)]_{S_0}^{S_3} = 0\), so

\([x(u_2) \land \text{next}(u_2, v_2)]_{S_0}^{S_3} (Z) = 0\)

for any \(Z\), even though \([\text{next}(u_2, u_2)]_{S_0}^{S_3} = \frac{1}{2}\). Then

\([x(u_1) \land \text{next}(u_1, u_1)]_{S_0}^{S_3} = 0\)

However,

\([x(u_1) \land \text{next}(u_1, u_2)]_{S_0}^{S_3} = \frac{1}{2}\)

when it is required to be a definite value. Since \(u_2\) is a summary node (\([\text{eq}(u_2, u_2)]_{S_0}^{S_3} = \frac{1}{2}\)) 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.

\([\text{next}(u_1, u_2)]_{S_0}^{S_1} = 1\)

\([\text{next}(u_1, u_2)]_{S_0}^{S_2} = 0\)
thus for all $Z$

\[
[x(v_1) \land \text{next}(v_1, v_2)]_{S_1}^3 (Z) \in \{0, 1\}
\]

\[
[x(v_1) \land \text{next}(v_1, v_2)]_{S_2}^3 (Z) = 0
\]

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

\[
U = \{u_1, u_2, u_3\}
\]

\[
\begin{array}{ccc}
  & u_1 & u_2 & u_3 \\
  x & 1 & 0 & 0 \\
  \\
  \text{circ[next]} & 0 & 0 & 0 \\
  \text{reach[x,next]} & 1 & 1 & 1 \\
  \text{next} & 0 & 1 & 0 \\
  & 0 & 7 & 7 \\
  & 0 & 1 & 2 \\
  \text{eq} & 1 & 0 & 0 \\
  & 0 & 2 & 0 \\
  & 0 & 0 & 1/2
\end{array}
\]

Now $[x(v_1) \land \text{next}(v_1, v_2)]_{S_3}^3 (Z) \in \{0, 1\}$ for all $Z$.

**Coerce**

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

\[
[\exists v_1 \bullet \text{next}(v_1, u_2) \land \text{next}(v_1, u_2)]_{S_1}^3 = 1
\]

but

\[
[\text{eq}(u_2, u_2)]_{S_1}^3 = \frac{1}{2}
\]

Coerce replaces $S_1$ with a state $S'_1$ that differs only by making the second node non-summary, i.e. $i(\text{eq}(u_2, u_2)) = 1$. Now all constraints are satisfied, notably

\[
S'_1 \models \exists v_1 \bullet \text{next}(v_1, v_2) \land \text{next}(v_1, v_3) \triangleright \text{eq}(v_2, v_3)
\]
4.3. ABSTRACT TRANSITIONS

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

$$\left[\forall v_1 \bullet \neg x(v_1) \lor \neg \text{next}^*(v_1, u_2)\right]_{S_2} = 1$$

but

$$\left[\neg \text{reach}[x, \text{next}](u_2)\right]_{S_3} = 0$$

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. $\iota(\text{eq}(u_2, u_2)) = 1$.

Update

The precondition of the transition is the formula

$$x(v_1) \land \text{next}(v_1, v_2)$$

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

$$x(v_1) = \exists v_2 \bullet x(v_2) \land \text{next}(v_2, v_1)$$

$$\text{reach}[x, \text{next}] (v_1) = \text{reach}[x, \text{next}] (v_1) \land (\neg x(v_1) \lor \text{circ}[\text{next}](v_1))$$

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

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:

<table>
<thead>
<tr>
<th></th>
<th>$S_5$</th>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$S_5$</th>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$u_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>0</td>
<td>1</td>
<td></td>
<td>$x$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>reach[$x$, next]</td>
<td>0 1</td>
<td></td>
<td>reach[$x$, next]</td>
<td>0 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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. CONCURRENT SYSTEMS

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. 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

---

9Yahav 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 \texttt{tr\_scheduled} to unknown for all threads based on the following update formula:

\[
\text{tr\_scheduled}(v) = \text{is\_thread}(v) \land \frac{1}{2}
\]
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 \( \text{eq} \) 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 \( \text{eq} \), which can be defined as an instrumentation predicate:

\[
\text{sm}(v) = \neg \text{eq}(v, v)
\]

In an abstract state, \( \text{sm}(u) \) is unknown if \( u \) is a summary node, and is false if \( u \) is a non-summary node. However, if \( \text{sm} \) is used in place of \( \text{eq} \), 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 \( \text{sm} \) 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 \( \text{sm} \) in place of \( \text{eq} \).
4.5.2 Partially Disjunctive Analysis

Manevich et al. [2004] describe an approach for storing abstract states in the state space 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 $u_1, \ldots, u_k$

$$i_3(p)(u_1, \ldots, u_k) = \begin{cases} i_1(p)(u_1, \ldots, u_k) & \text{if } i_1(p)(u_1, \ldots, u_k) = i_2(p)(u_1, \ldots, u_k) \\ \frac{1}{2} & \text{otherwise} \end{cases}$$

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. IMPROVEMENTS

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 non-blocking 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 \texttt{location}. Thus, thread $t_1$ is at location ‘idle’ when $t_1.location = \text{idle}$.

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 \texttt{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 \texttt{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 \textit{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 : T; next : Node\}

Shared: Head : Node := null

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

11: operation POP()
12: repeat
13: atomic
14: \( ss := \text{Head} \)
15: if \( ss = \text{null} \) 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: \( ss.next := \text{null} \)
23: \( ss.val := \text{null} \)
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 interleavings.

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 \( i \)th 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
5.2. SPECIFYING LINEARISABILITY

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

\[
\begin{align*}
\text{call}_i(\text{PUSH}(v_1)) & \quad - \quad \text{apply}_i(\text{PUSH}(v_1), \text{ok}) & \quad - \quad \text{return}_i(\text{PUSH}(v_1), \text{ok}) \\
    & \vdots & \vdots & \vdots \\
\text{call}_i(\text{PUSH}(v_n)) & \quad - \quad \text{apply}_i(\text{PUSH}(v_n), \text{ok}) & \quad - \quad \text{return}_i(\text{PUSH}(v_n), \text{ok}) \\
\text{call}_i(\text{POP}) & \quad - \quad \text{apply}_i(\text{POP}, \text{empty}) & \quad - \quad \text{return}_i(\text{POP}, \text{empty}) \\
\text{call}_i(\text{POP}) & \quad - \quad \text{apply}_i(\text{POP}, \text{ok}(v_1)) & \quad - \quad \text{return}_i(\text{POP}, \text{ok}(v_1)) \\
    & \vdots & \vdots & \vdots \\
\text{call}_i(\text{POP}) & \quad - \quad \text{apply}_i(\text{POP}, \text{ok}(v_n)) & \quad - \quad \text{return}_i(\text{POP}, \text{ok}(v_n))
\end{align*}
\]

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 \textit{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 \textit{trace} output of call$_2(\text{PUSH}(a))$ is (inv$_2(\text{PUSH}, a))$. For the final transition of an operation, the output is a
singleton containing the appropriate response. For example, the trace output of return_{\text{POP}}(\text{POP}, \text{empty}) is (\text{resp}_T(\text{POP}, \text{empty})). 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.

\footnote{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 $lp_p(\text{op}, val^*)$, 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 $\tau$ between two implementation states $s_1$ and $s_2$, and a specification state $s'_1$ related to $s_1$, i.e.

\[
\begin{array}{c}
\text{s}_1 \\
\text{R}
\end{array} \xrightarrow{\tau} \begin{array}{c}
\text{s}_2 \\
\text{s}'_1
\end{array}
\]

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

\[ s_1 \overset{\tau}{\longrightarrow} s_2 \]

\[ \downarrow R \]

\[ s'_1 \overset{\tau'}{\longrightarrow} s'_2 \]

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 \( \tau \) between two implementation states \( s_1 \) and \( s_2 \), and a specification state \( s'_2 \) related to \( s_2 \), i.e.

\[ s'_2 \]

\[ \overset{R}{\longrightarrow} \]

\[ s_1 \overset{\tau}{\longrightarrow} s_2 \]

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

\[ s'_1 \overset{\tau'}{\longrightarrow} s'_2 \]

\[ \overset{R}{\longrightarrow} \]

\[ s_1 \overset{\tau}{\longrightarrow} s_2 \]

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 \cite{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 De-queue 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. 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 state space 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 — the complexity of model checking is dependent on

---

2Their 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).

3Colvin 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 \textit{Smith and Derrick} \cite{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:

\[
\begin{array}{c}
  s_1 \\
  \downarrow R \\
  s'_1 \\
  \uparrow \\
  s_2 \\
\end{array}
\]

there could be two specification transition sequences $s'_1 \xrightarrow{\tau'} s'_2$ and $s'_1 \xrightarrow{\tau''} s'_3$ 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 \textit{Smith and Derrick} \cite{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 call$_1$(\textsc{Push}(a)), call$_2$(\textsc{Push}(a)), and call$_2$(\textsc{Push}(b)) would all have the same output to trace: inv(\textsc{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
5.2. SPECIFYING LINEARISABILITY

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 \(\tau\) in both the implementation and specification systems, we construct a transition \(\tau_r\), such that \(s_2 \xrightarrow{\tau} s_1\) 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
5.2. SPECIFYING LINEARISABILITY

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

\[ \text{AGEF} \text{init} \]

where \text{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 Do [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 \text{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. 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 non-blocking 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.

4The pseudocode “when φ τ” means that transition τ is only enabled when formula φ is true.
5.2. SPECIFYING LINEARISABILITY

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 ∨
23:              ¬∃t • t.observedEmptyWontLoop
24:           if CAS(Head, sshead, ssnext) then
25:             ssnext.val := null
26:             sstail := Tail
27:             if sshead = sstail then
28:               CAS(Tail, sstail, ssnext)
29:             end if
30:           end if
31:       end if
32:     end if
33:   end loop
34: 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). 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, (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

---

5The two “IDLE” states in each diagram are the same state repeated for clarity.
a non-empty Dequeue; thus the specification must be able to perform apply\(_i\)(\text{DEQUEUE,ok}(v)) after performing apply\(_i\)(\text{DEQUEUE,empty}).

The modified concurrent specification can be simply argued to be linearisable. There is exactly one apply\(_i\)(\text{ENQUEUE}(v),\text{ok}) step in an (unmodified) Enqueue operation and exactly one apply\(_i\)(\text{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\(_i\)(\text{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 \( \text{doneLP} \) 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 \texttt{repeat} . . . \texttt{until} structure has been rearranged into a \texttt{loop} . . . \texttt{break} structure.

In the Push operation, the auxiliary variable \( \text{doneLP} \) 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 \( \text{doneLP} \) 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).
5.2. SPECIFYING LINEARISABILITY

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: SPECUSH(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(SPECIS_EMPTY())
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 2.8(a) and 2.8(b), with the Dequeue operation modified to allow multiple empty linearisation points, and with the concurrent specification merged in.

The Enqueue operation in Figure 5.7(a) 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 5.7(b) 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
### 5.2. Specifying Linearisability

(a) Enqueue operation

![Figure 5.7: A lock-free queue algorithm with merged specification](image-url)
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
5.2. SPECIFYING LINEARISABILITY

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

\[ t.\text{location} \neq \text{idle} \rightarrow F(t.\text{location} = \text{idle}) \]

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

\[ AG(\forall i \cdot t_i.\text{location} \neq \text{idle} \rightarrow F(t_i.\text{location} = \text{idle})) \]

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:

\[ AG(t_1.\text{location} \neq \text{idle} \rightarrow F(t_1.\text{location} = \text{idle})) \] (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 \( \text{last\_step} \) to record which thread performed the immediately preceding step, then these futures can be specified by the following formulas:
CHAPTER 5. MODEL CHECKING NONBLOCKING ALGORITHMS

- \( F(t_1.\text{location} = \text{idle}) \)
- \( G(t_1.\text{location} \neq \text{idle}) \land GF(\text{last}_step = t_1) \)
- \( G(t_1.\text{location} \neq \text{idle}) \land FG(\text{last}_step \neq t_1) \)

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

\[
F(t_1.\text{location} = \text{idle}) \lor FG(\text{last}_step \neq t_1)
\]

Note that the \( \text{last}_step \) 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 \( \text{lastStep}TOne \) instead. Thus the full wait-free property is expressed in LTL as:

\[
AG(t_1.\text{location} \neq \text{idle} \rightarrow (F(t_1.\text{location} = \text{idle}) \lor FG(\neg \text{lastStep}TOne)))
\]  
(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:

\[
\exists i \cdot t_i.\text{location} \neq \text{idle} \land X(t_i.\text{location} = \text{idle})
\]

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

\[
AGF \text{was}_\text{response}
\]  
(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:

\[
AG(t_1.\text{location} \neq \text{idle} \land t_1.\text{doneLP} \rightarrow F(t_1.\text{location} = \text{idle}))
\] (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:

\[
AG(t_1.\text{location} \neq \text{idle} \land \neg t_1.\text{doneLP} \rightarrow F(t_1.\text{doneLP}) \lor GF(\text{wasLP}))
\] (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:

\[
AG(t_1.\text{location} \neq \text{idle} \land t_1.\text{doneLP} \rightarrow F(t_1.\text{location} = \text{idle}) \lor FG(\neg \text{lastStepTOne}))
\] (LF-3.1)

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

\text{symmetry} we are able to choose just one (the first: } t_1.\text{)}.
5.3. SPECIFYING NONBLOCKING PROPERTIES

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

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

This formula has no direct equivalent in LTL, but we can construct some-
thing 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:

$$\text{AG}(t_1.\text{location} \neq \text{idle} \rightarrow\
[\text{lastStepTOne} \text{U} (t_1.\text{location} = \text{idle} \lor \neg \text{lastStepTOne}])]$$ (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 com-
pletes 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 viol-
ates 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 $p \text{ U} (q \lor \neg p)$ 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 nonblock-
ing 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 [Don-
gol 2006], in a logic [Dongol and Goldson 2006] not directly applicable to
(existing) model checkers. [Dongol 2009] has expressed these in first or-
der 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:

$$\text{AG}(t_1.\text{location} \neq \text{idle} \rightarrow \text{F}(t_1.\text{location} = \text{idle} \lor \neg \text{lastStepTOne}))$$

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 \textit{completed} that is incremented by the final step of each operation, we could define lock-freedom as follows:

$$\forall n \in \mathbb{N} \quad \text{AG}(\text{completed} = n \rightarrow \text{F}(\text{completed} > n))$$

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 state-space 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 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 state-space 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.
CHAPTER 6. BOUNDED VERIFICATION

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. How-
ever, I found Spin to be much more practical for implementing garbage
collection. The procedural language Promela allows all of the garbage col-
lection code to be wrapped in a \texttt{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, spec-
ifying the property as described in Section 5.2. We first discover (known)
bugs in some incorrect algorithms, and then verify to the largest possi-
bles 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 struc-
ture 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 \texttt{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

\[ 48': \text{free}(ss) \]

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-
CHAPTER 6. BOUNDED VERIFICATION

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

Operation \( (d_2) \) 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:

\[
\text{inv}_2(\text{PUSH}, d_1), \text{resp}_2(\text{PUSH}, \text{ok}), \text{inv}_2(\text{POP}), \text{inv}_1(\text{POP}), \\
\text{resp}_2(\text{POP}, d_1), \text{inv}_2(\text{PUSH}, d_2), \text{resp}_2(\text{PUSH}, \text{ok}), \text{resp}_1(\text{POP}, d_1)
\]

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

\[
\text{inv}_2(\text{PUSH}, d_1), \text{resp}_2(\text{PUSH}, \text{ok}), \text{inv}_2(\text{POP}), \text{resp}_2(\text{POP}, d_1), \\
\text{inv}_2(\text{PUSH}, d_2), \text{resp}_2(\text{PUSH}, \text{ok}), \text{inv}_1(\text{POP}), \text{resp}_1(\text{POP}, d_1)
\]

\[
\text{inv}_2(\text{PUSH}, d_1), \text{resp}_2(\text{PUSH}, \text{ok}), \text{inv}_2(\text{POP}), \text{resp}_2(\text{POP}, d_1), \\
\text{inv}_1(\text{POP}), \text{resp}_1(\text{POP}, d_1), \text{inv}_2(\text{PUSH}, d_2), \text{resp}_2(\text{PUSH}, \text{ok})
\]

\[
\text{inv}_2(\text{PUSH}, d_1), \text{resp}_2(\text{PUSH}, \text{ok}), \text{inv}_1(\text{POP}), \text{resp}_1(\text{POP}, d_1), \\
\text{inv}_2(\text{POP}), \text{resp}_2(\text{POP}, d_1), \text{inv}_2(\text{PUSH}, d_2), \text{resp}_2(\text{PUSH}, \text{ok})
\]

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.
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\textsuperscript{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\textsuperscript{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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Bug</th>
<th>$n_t$</th>
<th>$n_s$</th>
<th>$n_d$</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>ABA\textsuperscript{1}</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>[Treiber, 1986]</td>
</tr>
<tr>
<td>Stack</td>
<td>ABA\textsuperscript{1}</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>[Treiber, 1986]</td>
</tr>
<tr>
<td>Queue</td>
<td>ABA\textsuperscript{1}</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>[Michael and Scott, 1996, 1998]</td>
</tr>
<tr>
<td>Queue</td>
<td>ABA\textsuperscript{2}</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>[Michael and Scott, 1996, 1998]</td>
</tr>
<tr>
<td>Queue</td>
<td>“skip”</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>[Shann et al., 2000]</td>
</tr>
<tr>
<td>Queue</td>
<td>ABA\textsuperscript{1}</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>[Ladan-Mozes and Shavit, 2004, 2008]</td>
</tr>
<tr>
<td>Deque</td>
<td>“empty”</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>[Detlefs et al., 2000]</td>
</tr>
<tr>
<td>Deque</td>
<td>ABA\textsuperscript{3}</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>[Detlefs et al., 2000]</td>
</tr>
</tbody>
</table>

Table 6.1: Minimum parameters for counterexamples to linearisability
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 $2 \leq n_t \leq 3$, $1 \leq n_s \leq 4$, and $1 \leq n_d \leq 2$.

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. $4 \leq n_t \leq 5$, $5 \leq n_s \leq 6$, and $3 \leq n_d \leq 4$. 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.
CHAPTER 6. BOUNDED VERIFICATION

Parameters Spin − DCOLLAPSE − DMA

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

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 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., 2004a], and Ta-

---

2The remaining columns are discussed in the next section.
### 6.1. CHECKING LINEARISABILITY

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

Table 6.3: MS Queue Bounded Verification

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

Table 6.4: DGLM Queue Bounded Verification
### Table 6.5: LMS Queue Bounded Verification

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

**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.
6.1. CHECKING LINEARISABILITY

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 \(-\text{DCOLLAPSE}\). 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 \(-\text{DMA}\). 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.
### Table 6.6: Treiber Stack Bounded Verification With Symmetry Reduction

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Spin</th>
<th>DMA</th>
<th>TopSpin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Th Mem</td>
<td>RAM Time</td>
<td>RAM Time</td>
<td>RAM Time</td>
</tr>
<tr>
<td>3 4 3</td>
<td>2,983 59s</td>
<td>1,471 10.4m</td>
<td>607 10s</td>
</tr>
<tr>
<td>4 3 4</td>
<td>3,228 74s</td>
<td>1,396 10.2m</td>
<td>199 4s</td>
</tr>
<tr>
<td>2 6 2</td>
<td>3,256 46s</td>
<td>957 12.6m</td>
<td>1,945 24s</td>
</tr>
<tr>
<td>7 2 1</td>
<td>4,023 119s</td>
<td>1,873 14.4m</td>
<td>10 0s</td>
</tr>
<tr>
<td>6 2 4</td>
<td>4,282 122s</td>
<td>1,547 14.0m</td>
<td>30 1s</td>
</tr>
<tr>
<td>6 2 7</td>
<td>— —</td>
<td>2,896 39.7m</td>
<td>70 2s</td>
</tr>
<tr>
<td>5 3 2</td>
<td>— —</td>
<td>3,153 24.0m</td>
<td>107 3s</td>
</tr>
<tr>
<td>4 3 6</td>
<td>— —</td>
<td>3,390 37.2m</td>
<td>625 14s</td>
</tr>
<tr>
<td>3 4 4</td>
<td>— —</td>
<td>3,596 38.5m</td>
<td>1,856 30s</td>
</tr>
<tr>
<td>3 6 1</td>
<td>— —</td>
<td>3,755 28.3m</td>
<td>1,282 20s</td>
</tr>
<tr>
<td>7 3 3</td>
<td>— —</td>
<td>— —</td>
<td>2,611 96s</td>
</tr>
<tr>
<td>3 5 2</td>
<td>— —</td>
<td>— —</td>
<td>2,672 44s</td>
</tr>
<tr>
<td>2 5 4</td>
<td>— —</td>
<td>1,270 23.8m</td>
<td>3,225 42s</td>
</tr>
<tr>
<td>7 4 1</td>
<td>— —</td>
<td>— —</td>
<td>3,359 110s</td>
</tr>
<tr>
<td>5 3 7</td>
<td>— —</td>
<td>— —</td>
<td>3,389 93s</td>
</tr>
<tr>
<td>5 4 2</td>
<td>— —</td>
<td>— —</td>
<td>3,746 525s</td>
</tr>
<tr>
<td>4 4 3</td>
<td>— —</td>
<td>— —</td>
<td>3,847 79s</td>
</tr>
<tr>
<td>3 4 5</td>
<td>— —</td>
<td>— —</td>
<td>3,849 68s</td>
</tr>
<tr>
<td>6 3 5</td>
<td>— —</td>
<td>— —</td>
<td>3,981 127s</td>
</tr>
</tbody>
</table>
The data values of many algorithms can be modelled using scalar sets. 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 scalar set. 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 scalar set. 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 scalar set. 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. TopSpin does not use scalar sets 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 scalar set 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 scalar sets) 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

\[ \text{EF}(\neg \text{thread}[1]@\text{IDLE} \land \text{G}\neg \text{thread}[1]@\text{IDLE}) \]

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
begin PUSH\((d_1)\)
2: \(n := \texttt{new}(\text{Node})[0]\)
3: \(n.val := lv[d_1]\)
5: \(ss := \text{Head}[null]\)
6: \(n.next := ss[null]\)
7: CAS(\text{Head}[null], ss[null], n[0])
begin \text{POP}(\)
11: \(ss := \text{Head}[0]\)
17: \(\neg\)CAS(\text{Head}[null], ss[0], ssnext[null])
begin PUSH\((d_1)\)
2: \(n := \texttt{new}(\text{Node})[0]\)
3: \(n.val := lv[d_1]\)
5: \(ss := \text{Head}[null]\)
6: \(n.next := ss[null]\)
7: CAS(\text{Head}[null], ss[null], n[0])
11: \(ss := \text{Head}[0]\)
—end infinite loop—

Thread 2
begin \text{PUSH}\((d_1)\)
2: \(n := \texttt{new}(\text{Node})[0]\)
3: \(n.val := lv[d_1]\)
5: \(ss := \text{Head}[null]\)
6: \(n.next := ss[null]\)
7: CAS(\text{Head}[null], ss[null], n[0])
begin \text{POP}(\)
11: \(ss := \text{Head}[0]\)
17: \(\neg\)CAS(\text{Head}[null], ss[0], ssnext[null])
begin \text{PUSH}\((d_1)\)
2: \(n := \texttt{new}(\text{Node})[0]\)
3: \(n.val := lv[d_1]\)
5: \(ss := \text{Head}[null]\)
6: \(n.next := ss[null]\)
7: CAS(\text{Head}[null], ss[null], n[0])
11: \(ss := \text{Head}[0]\)
—end infinite loop—

Figure 6.3: Simplified execution of stack showing wait-freedom counterexample
### 6.2. Checking Nonblocking Properties

#### Table 6.7: Minimum parameters for counterexamples to nonblocking properties

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Property</th>
<th>Th</th>
<th>Mem</th>
<th>Data</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>WF</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>Treiber, 1986</td>
</tr>
<tr>
<td>Queue</td>
<td>WF</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>Michael and Scott, 1996, 1998</td>
</tr>
<tr>
<td>Queue</td>
<td>WF</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>Ladan-Mozes and Shavit, 2004, 2008</td>
</tr>
<tr>
<td>Deque</td>
<td>WF</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>Detlefs et al., 2000</td>
</tr>
<tr>
<td>Queue</td>
<td>WF</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Shann et al., 2000</td>
</tr>
<tr>
<td>Queue</td>
<td>LF</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Shann et al., 2000</td>
</tr>
<tr>
<td>Queue</td>
<td>OF</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Shann et al., 2000</td>
</tr>
</tbody>
</table>

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 \( n_t \geq 2, n_s \geq 1 \) and \( n_d \geq 1 \) 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
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-
### 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.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>DGLM Queue Bounded Verification</th>
<th>Parameters</th>
<th>LMS Queue Bounded Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Th Mem Data Spin</td>
<td>RAM Time</td>
<td>Th Mem Data Spin</td>
<td>RAM Time</td>
</tr>
<tr>
<td>2 5 2</td>
<td>3,143 29s</td>
<td>4 2 7</td>
<td>3,007 63s</td>
</tr>
<tr>
<td>3 3 5</td>
<td>3,815 68s</td>
<td>2 4 3</td>
<td>3,336 50s</td>
</tr>
<tr>
<td>6 2 1</td>
<td>4,237 118s</td>
<td>2 6 1</td>
<td>3,565 56s</td>
</tr>
<tr>
<td>5 2 4</td>
<td>4,378 112s</td>
<td>3 3 2</td>
<td>3,639 77s</td>
</tr>
<tr>
<td>2 4 5</td>
<td>4,443 74s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.10: DGLM Queue Bounded Verification

Table 6.11: LMS Queue Bounded Verification

ation up to sufficient bounds to supply confidence in an algorithm. TopSpin only supports verifying safety properties, but future work may extend it to combine techniques for supporting LTL properties [e.g. Bošnački 2003].
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
CHAPTER 7. C. A. FOR LINEARISABILITY

(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()        push3  
  push3 newNode(n)         push4  
  push4 assign(n.val, lv)   push6  
  push6 assign(ss, Head)   push7  
  push7 assign(n.next, ss) push9  
}
atomic {
  push9  CASfail(Head, ss) push6  
  push9  CASsucc(Head, ss, n) push11 
  push11 specPush()        push18 
  push18 endPush()         idle  
}
atomic {
  idle  beginPop()        pop24  
  pop24 assign(ss, Head)   pop25  
  pop25 isNotNull(ss)      pop35  
  pop25 isNull(ss)         pop26  
  pop26 specPopEmpty()     pop48  
}
atomic {
  pop35 assign(ssnext, ss.next) pop36  
  pop36 assign(lv, ss.val)   pop38  
}
atomic {
  pop38  CASfail(Head, ss) pop24  
  pop38  CASsucc(Head, ss, ssnext) pop40  
  pop40  specPop()        pop48  
  pop48  endPop()         idle  
}

Figure 7.2: Transitions of stack model
Partial order reduction, as discussed in Section 5.1.3. The update actions are detailed in Figure 7.3. Each action has the form

\[ \text{guard} \rightarrow \text{updates} \]

where \( \text{guard} \) is a boolean formula that must be true for the action to be applied, and \( \text{updates} \) 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

\[ \text{endPushError}() \rightarrow \neg \text{doneLP} \rightarrow \]

and transition

\[ \text{push}_{18} \rightarrow \text{endPushError}() \rightarrow \text{lin_fail} \]

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

\[ \text{endPush}() \rightarrow \text{true} \rightarrow \text{lv}, \text{ss}, n := \text{null} \]

and have it trigger a message \( \neg \text{doneLP} \) 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).

---

1Section 7.8 contains a discussion of models with different partial order reduction.
newNode(x) true →
   x := newNode(null, null)

assign(x, y) true →
   x := y

isNull(x) x = null →
isNotNull(x) x ≠ null →

CASfail(loc, old) loc ≠ old →
CASsucc(loc, old, new) loc = old →
   loc := new

beginPush() true →
   doneLP := false; lv ∈ DATA

specPush() ¬ doneLP →
   doneLP := true; spec.Head := newNode(lv, spec.Head)

dupPush() doneLP →
   lv, ss, n := null

beginPop() true →
   doneLP := false

specPopEmpty() ¬ doneLP ∧ spec.Head = null →
   doneLP := true

specPop() ¬ doneLP ∧ spec.Head ≠ null ∧ spec.Head.val = lv →
   doneLP := true; spec.Head := spec.Head.next

dupPop() doneLP →
   lv, ss, ssnext := null

Figure 7.3: Update actions used in stack model
7.2. THREE-VALUED 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:

\[ \text{is\_thread}, \text{is\_node}, \text{is\_data} \]

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:

\[ \text{Head}_I, \text{Head}_S \]

For the threads we define a unary predicate for each location:

\[ \text{at\[idle\]}, \text{at\[push\_3\]}, \text{at\[push\_4\]}, \text{at\[push\_6\]}, \text{at\[push\_7\]}, \text{at\[push\_9\]}, \text{at\[push\_1\_1\]}, \text{at\[push\_1\_8\]}, \text{at\[pop\_2\_4\]}, \text{at\[pop\_2\_5\]}, \text{at\[pop\_3\_6\]}, \text{at\[pop\_4\_0\]}, \text{at\[pop\_4\_8\]} \]

Finally, the fields of the threads and nodes are represented by binary predicates and the boolean variables by unary predicates:

\[ \text{next, val, n, lv, ss, ssnext, doneLP} \]

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:

\[ \forall v_1, v_2 \cdot \text{Head}_I(v_1) \land \text{Head}_I(v_2) \rightarrow \text{eq}(v_1, v_2) \]
\[ \forall v_1, v_2 \cdot \text{Head}_S(v_1) \land \text{Head}_S(v_2) \rightarrow \text{eq}(v_1, v_2) \]
Additionally, the node and thread fields are functional, so we include the following rules:

\[
\forall v_1, v_2, v_3 \cdot \text{next}(v_1, v_2) \land \text{next}(v_1, v_3) \rightarrow \text{eq}(v_2, v_3)
\]

\[
\forall v_1, v_2, v_3 \cdot \text{val}(v_1, v_2) \land \text{val}(v_1, v_3) \rightarrow \text{eq}(v_2, v_3)
\]

\[
\forall v_1, v_2, v_3 \cdot \text{n}(v_1, v_2) \land \text{n}(v_1, v_3) \rightarrow \text{eq}(v_2, v_3)
\]

\[
\forall v_1, v_2, v_3 \cdot \text{l}(v_1, v_2) \land \text{l}(v_1, v_3) \rightarrow \text{eq}(v_2, v_3)
\]

\[
\forall v_1, v_2, v_3 \cdot \text{ss}(v_1, v_2) \land \text{ss}(v_1, v_3) \rightarrow \text{eq}(v_2, v_3)
\]

\[
\forall v_1, v_2, v_3 \cdot \text{ssnext}(v_1, v_2) \land \text{ssnext}(v_1, v_3) \rightarrow \text{eq}(v_2, v_3)
\]

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:

\[
circ(v) \iff \text{next}^+(v, v) \\
\text{reach}_1(v) \iff \exists u \cdot \text{Head}_1(u) \land \text{next}^*(u, v) \\
\text{reach}_S(v) \iff \exists u \cdot \text{Head}_S(u) \land \text{next}^*(u, v)
\]

\section*{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.:

- \text{val} is always non-null, as we only push non-null values into the list.
- At location \text{push}_9, the \text{next} field of a thread’s \text{n} node is null if and only if the thread’s \text{ssnext} field is null.
- Push operations only have non-null values for \text{n} and \text{lv} after they are assigned.
- In a non-empty \text{pop} operation, \text{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:

\[
\text{has[field]}(v) \iff \exists u \cdot \text{field}(v, u)
\]

\section*{Referenced-by-field}

It is possible to distinguish the objects that are pointed to by the various field predicates using the \text{r_by[field]} predicate defined in Section\ref{ref}. For the field predicates \text{next}, \text{ss}, \text{ssnext} and \text{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\ref{ref}.)

This instrumentation predicate would be beneficial for the \text{n} field predicate, as it is necessary to distinguish the nodes that have yet to be added.

\footnote{For brevity, we remove the “parameters” of these predicates, as used in Chapter\ref{ref} 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, \( r \text{\_by} |n| \) 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:

\[
\text{waiting}(v) \iff \exists u \cdot n(u, v) \land \neg \text{doneLP}(u)
\]

**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:

\[
\text{shared}[n](v) \iff \exists u_1, u_2 \cdot n(u_1, v) \land n(u_2, v) \land \neg \text{eq}(u_1, u_2)
\]

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 \( S^1 \) the lists have the same values in the same order, and in \( S^2 \) 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 \( S^1 \) and \( S^2 \) are in the same equivalence class, even though \( S^1 \) is reachable and \( S^2 \) is not; thus \( S^2 \)'s unreachable (and erroneous) successors are treated as being reachable from \( S^1 \). Specifically, abstract state \( S_1 \), representing \( S^1 \),
Figure 7.4: Canonical abstraction of two lists: the property of matching values is lost
specPush$(x) \rightarrow \text{doneLP} \rightarrow$

\[
\text{doneLP} := \text{true}; \ \text{spec}.\text{Head} := \text{newNode}(x, \text{spec}.\text{Head});
\]
\[
\text{spec}(\text{Head}, \text{spec}.\text{Head}) := \text{true}
\]

\begin{align*}
\text{specPop}(x) & \rightarrow \text{doneLP} \land \text{spec}.\text{Head} \neq \text{null} \land \text{spec}.\text{Head}.\text{val} = x \rightarrow \\
\text{doneLP} & := \text{true}; \ \text{spec}(-, \text{spec}.\text{Head}) := \text{false}; \\
\text{spec}.\text{Head} & := \text{spec}.\text{Head}.\text{next}
\end{align*}

Figure 7.5: Stack specification update operations, incorporating the $\text{spec}$ relation

can be concretised by Focus into $S^2_3$, 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:

\[
\text{ipair}(n_1, n_2) \iff \exists n_3, n_4 \bullet \\
\text{Head}^i(n_3) \land \text{Head}^5(n_4) \land \\
\exists i \bullet \text{next}^i(n_3, n_1) \land \text{next}^i(n_4, n_2)
\]

Instead, we introduce an additional core binary predicate that relates nodes in the two lists. We call it $\text{spec}$ 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 $\text{specPush}$ step and unset in the $\text{specPop}$ step as shown in Figure 7.5.

The new core predicate $\text{spec}$ records a one-to-one correspondence between nodes, so we include integrity rules for functionality and inverse functionality:

\[
\forall v_1, v_2, v_3 \bullet \text{spec}(v_1, v_2) \land \text{spec}(v_1, v_3) \rightarrow \text{eq}(v_2, v_3)
\]
\[
\forall v_1, v_2, v_3 \bullet \text{spec}(v_1, v_3) \land \text{spec}(v_2, v_3) \rightarrow \text{eq}(v_1, v_2)
\]
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:

\[ r_{by}[spec](v) \iff \exists u \bullet spec(u, v) \]

### 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:

\[ matching(n_1) \iff \exists n_2, d_1 \bullet spec(n_1, n_2) \land val(n_1, d_1) \land val(n_2, d_1) \]

The predicate records a "triangular" relationship between nodes and data values, as shown in the following diagram:

![Diagram showing the relationship between nodes and data values](image)

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
Figure 7.6: Canonical abstraction of stack lists using the predicate matching: the property of matching values is retained
7.3. PRESERVING LINEARISABILITY INFORMATION

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 $S_5^\ddagger$, 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:

$$\text{commutes}(n_1) \iff \exists n_2, n_3, n_4 \cdot$$

$$\text{next}(n_1, n_2) \land \text{spec}(n_1, n_3) \land \text{next}(n_3, n_4) \land \text{spec}(n_2, n_4)$$

The predicate records a “square” relationship between nodes, as shown in the following diagram:

![Diagram](image-url)
7.3. PRESERVING LINEARISABILITY INFORMATION

Figure 7.7: Canonical abstraction with “crossed” spec predicates: the property of ordered values is lost
In Figure 7.8 we add \texttt{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 \texttt{spec} predicates is recorded. In Figure 7.8a, \texttt{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, \texttt{commutes} is false for all nodes, so the two lists contain the same values but in different orders. Note that \( S^\hat{t}_7 \) has the same diagram as \( S^\hat{t}_5 \), but it is a different and more precise state because \texttt{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 (\texttt{CASsucc} at location \texttt{push\_9}) and the specification Push update (\texttt{specPush} at location \texttt{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. Logically, because of the values of the \texttt{matching} and \texttt{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:

\[
\text{has}_{S\mid \text{spec}}(v_1) \iff \exists v_2 \bullet \text{Head}_{S\mid v_2} \land \text{spec}(v_1, v_2)
\]

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.

\footnote{The data values are not shown, in order to make the diagram clearer.}
7.3. PRESERVING LINEARISABILITY INFORMATION

Figure 7.8: Canonical abstraction using the commutes predicate: the property of ordered values is retained
Figure 7.8: Canonical abstraction using the commutes predicate: the property of ordered values is retained.
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:

\[
\begin{align*}
  r_{by}[\text{fld, threadA}](v) & \iff \exists t \cdot \text{threadA}(v) \land \text{fld}(t, v) \\
  r_{by}[\text{fld, threadB}](v) & \iff \exists t \cdot \text{threadB}(v) \land \text{fld}(t, v)
\end{align*}
\]

Whilst this approach works for models with a fixed number of threads,
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. (UN)BOUNDED THREADS

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. 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:

\[ \exists t_1, t_2, t_3 \cdot \text{is\_thread}(t_1) \land \text{is\_thread}(t_2) \land \text{is\_thread}(t_3) \land \neg \text{eq}(t_1, t_2) \land \neg \text{eq}(t_2, t_3) \land \neg \text{eq}(t_1, t_3) \land \neg \text{eq}(t_2, t_4) \]

and for a bound of 3:

\[ \exists t_1, t_2, t_3, t_4 \cdot \text{is\_thread}(t_1) \land \text{is\_thread}(t_2) \land \text{is\_thread}(t_3) \land \text{is\_thread}(t_4) \land \neg \text{eq}(t_1, t_2) \land \neg \text{eq}(t_2, t_3) \land \neg \text{eq}(t_3, t_4) \land \neg \text{eq}(t_1, t_3) \land \neg \text{eq}(t_2, t_4) \land \neg \text{eq}(t_1, t_4) \]

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:

\[ \exists t_1, t_2, t_3 \cdot \text{is\_thread}(t_1) \land \text{is\_thread}(t_2) \land \text{is\_thread}(t_3) \land \neg \text{eq}(t_1, t_2) \land \neg \text{eq}(t_2, t_3) \land \neg \text{eq}(t_1, t_3) \land \neg \text{eq}(t_2, t_4) \land \neg \text{eq}(t_1, t_4) \land \neg \text{eq}(t_2, t_4) \land \neg \text{eq}(t_1, t_4) \]

and for a bound of 3:

\[ \exists t_1, t_2, t_3, t_4 \cdot \text{is\_thread}(t_1) \land \text{is\_thread}(t_2) \land \text{is\_thread}(t_3) \land \text{is\_thread}(t_4) \land \neg \text{eq}(t_1, t_2) \land \neg \text{eq}(t_2, t_3) \land \neg \text{eq}(t_3, t_4) \land \neg \text{eq}(t_1, t_3) \land \neg \text{eq}(t_2, t_4) \land \neg \text{eq}(t_1, t_4) \land \neg \text{eq}(t_2, t_4) \land \neg \text{eq}(t_1, t_4) \land \neg \text{eq}(t_2, t_4) \land \neg \text{eq}(t_1, t_4) \]

\[ \text{eq}(t_1, t_4) \]

4Or, more generally in the latter case, if the initial state has more than one thread object.
The latter forms ensure that any state with \textit{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 \textit{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 \textit{bound} − 1 thread objects could be focussed to \textit{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 use of geometric instrumentation predicates.

**Snapshots**

In the stack’s Pop operation, the \texttt{ssnext} field is the \texttt{next}-successor of the \texttt{ss} field when it is read. This property is assumed to persist, so it is not checked before the CAS step at location \texttt{pop38} that attempts to set \texttt{Head} to \texttt{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 \texttt{S}_8\natural, both \texttt{ssnext} predicates are the \texttt{next}-successors of the respective \texttt{ss} predicates, but this is not the case in \texttt{S}_9\natural; nevertheless, both states have the same canonical abstraction (\texttt{S}_8). As a consequence, the CAS transition can remove an arbitrary prefix of the list because \texttt{ssnext} can be concretised at any point.

This information can be preserved using a geometric instrumentation predicate on threads that expresses:

“\text{This thread’s fld}_1 \text{ and fld}_2 \text{ fields are non-null and fld}_2 \text{ 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:

\[ \text{succ}[^{\text{fld}_1, \text{fld}_2}(t_1) \Leftrightarrow \exists n_1, n_2 \cdot \text{fld}_1(t_1, n_1) \land \text{fld}_2(t_1, n_2) \land \text{next}(n_1, n_2) \]  

This predicate records a “triangular” relationship between threads and nodes, as shown in the following diagram:

In Figure 7.12 we add \( \text{succ}[\text{ss}, \text{ssnext}] \) 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 \( \text{succ}[\text{ss}, \text{ssnext}] \) 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 \( \text{ssnext} \) will always be concretised to the next-successor of \( \text{ss} \).

A similar issue arises in the Push operation, as it is implicit at the CAS step at location push_{0} that the next field of the \( n \) node still points to the same node as \( \text{ss} \); this can be resolved by including the predicate \( \text{succ}[n, \text{ss}] \).

**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:
Figure 7.12: Canonical abstraction using the $\text{succ}[ss, ssnext]$ predicate: the relationships between fields’ values are retained
Figure 7.12: Canonical abstraction using the \( \text{succ}[ss, ssnext] \) 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:

\[ \text{opval}[\text{field}](v) \leftrightarrow \exists u_1, u_2 \cdot \text{field}(v, u_1) \land \text{val}(u_1, u_2) \land \text{lv}(v, u_2) \]

This predicate records a “triangular” relationship between threads, nodes and data values, as shown in the following diagram:

I included the predicate \text{opval}[n], which records the relationship throughout a push operation between the \text{n} and \text{lv} fields.

A similar need arises in the pop operation, when \text{lv} is assigned to be the \text{ss} node’s \text{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 \text{lv}. To ensure that the relationship between \text{ss} and \text{lv} is retained in the abstraction, I included the predicate \text{opval}[\text{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
at\[\text{idle}\]  
\(tr_{\text{scheduled}} = \frac{1}{2}\)

Figure 7.13: Initial state of the stack model (plain)

Figure 7.14: Initial state of the stack model (with “garbage” nodes)
7.6. ADDITIONAL COMPATIBILITY CONSTRAINTS

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

$$\text{Head}(n_1) \land \text{next}(n_1, n_2) \land \text{ss}(t_1, n_3) \land \text{ssnext}(t_1, n_4)$$

the Focus operation produces a state where the summary node has been split in two (7.15b).

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.
Figure 7.15: Steps of a Pop update transition
7.6. ADDITIONAL COMPATIBILITY CONSTRAINTS

Figure 7.15: Steps of a Pop update transition
are unknown, so $\text{next}^+$, and hence the entire bodies of $\text{circ}$.s compatibility constraints (recall from Section 4.3.2), evaluate to unknown:

$$\exists u \bullet v = u \land \text{next}^+(v, u) \triangleright \text{circ}(v)$$

$$\forall u \bullet v \neq u \lor \neg \text{next}^+(v, u) \triangleright \neg \text{circ}(v)$$

The $\text{Head}_1$ 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 $\text{next}$ predicates, and the state is already canonically abstract. ■

This example may seen fairly benign — a subsequent transition that focuses the $\text{next}$ predicate from $\text{Head}_1$ 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 $\text{circ}$ constraints. The transition is the assignment of $\text{ssnext}$ from location $\text{pop}_{35}$ to $\text{pop}_{38}$; 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 $\text{next}$ predicates pointing at the head node, as created in the previous example; the thread’s $\text{ss}$ field points to somewhere in the body of the list.

Using the focus formula

$$\text{ss}(t_1, n_1) \land \text{next}(n_1, n_2)$$

the Focus operation produces a state where the summary node has been split, and the $\text{next}$-successor of $\text{ss}$ is the head node (7.16b). This state is inconsistent because $\text{circ}$ 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 $\text{circ}$ compatibility constraints, as before. Coerce instead just sharpens the $\text{ss}$ node to be non-summary (7.16c).

---

6 Again, this is only one of several states produced by Focus, as the unknown $\text{ss}$ and $\text{next}$ predicates can be concretised in several ways.
Figure 7.16: Transition to an inconsistent state
(c) After Coerce

(d) After Update and Coerce

Figure 7.16: Transition to an inconsistent state
7.6. ADDITIONAL COMPATIBILITY CONSTRAINTS

The Update step sets $ss_{next}$ 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 $\text{succ}[ss, ss_{next}]$ implies that there is a node reachable from $\text{Head}_i$ whose next field points to $\text{Head}_i$, but $\text{circ}$ implies that $\text{Head}_i$ 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:
  \[
  \neg \text{circ}(n_1) \land n_1 = n_2 \Rightarrow \neg \text{next}(n_1, n_2)
  \]

- For non-circular nodes, **next** is not symmetric:
  \[
  \neg \text{circ}(n_1) \land \text{next}(n_1, n_2) \Rightarrow \neg \text{next}(n_2, n_1)
  \]

- A node in a non-circular list cannot point back to the head:
  \[
  \text{Head}(n_1) \land \neg \text{circ}(n_1) \land \text{reach}_I(n_2) \Rightarrow \neg \text{next}(n_2, n_1)
  \]
  \[
  \neg \text{reach}_S(n_1) \land \neg \text{circ}(n_1) \land \text{reach}_S(n_2) \Rightarrow \neg \text{next}(n_2, n_1)
  \]

- A node in a list cannot point to a node that is not in the list:
  \[
  \text{reach}_I(n_1) \land \neg \text{reach}_I(n_2) \Rightarrow \neg \text{next}(n_1, n_2)
  \]
  \[
  \text{reach}_S(n_1) \land \neg \text{reach}_S(n_2) \Rightarrow \neg \text{next}(n_1, n_2)
  \]

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.

---

Note that formulating the rule as \( \neg \text{circ}(v) \Rightarrow \neg \text{next}(v, v) \) would remove unknown loops on summary nodes.
7.6. ADDITIONAL COMPATIBILITY CONSTRAINTS

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:

\[
\forall n_2, d_1 \cdot \neg \text{spec}(n_1, n_2) \lor \neg \text{val}(n_1, d_1) \lor \neg \text{val}(n_2, d_1) \rightarrow \neg \text{matching}(n_1)
\]

\[
\exists n_2, d_1 \cdot \text{spec}(n_1, n_2) \land \text{val}(n_1, d_1) \land \text{val}(n_2, d_1) \rightarrow \text{matching}(n_1)
\]

\[
\exists d_1 \cdot \neg \text{matching}(n_1) \land \text{val}(n_1, d_1) \land \text{val}(n_2, d_1) \rightarrow \neg \text{spec}(n_1, n_2)
\]

\[
\exists n_2 \cdot \neg \text{matching}(n_1) \land \text{spec}(n_1, n_2) \land \text{val}(n_2, d_1) \rightarrow \neg \text{val}(n_1, d_1)
\]

\[
\exists n_1 \cdot \neg \text{matching}(n_1) \land \text{spec}(n_1, n_2) \land \text{val}(n_1, d_1) \rightarrow \neg \text{val}(n_2, d_1)
\]

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 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 \( \text{val} \) predicate that is indefinite:

\[
\exists n_1 \bullet \text{matching}(n_1) \land \text{spec}(n_1, n_2) \land \text{val}(n_1, d_1) \Rightarrow \text{val}(n_2, d_1)
\]

\[
\exists n_2 \bullet \text{matching}(n_1) \land \text{spec}(n_1, n_2) \land \text{val}(n_2, d_1) \Rightarrow \text{val}(n_1, d_1)
\]

Note that these compatibility constraints rely on the fact that \( \text{spec} \) and \( \text{val} \) are functional predicates. However, \( \text{val} \) is not inversely functional, so the following constraint rule — to sharpen when it is \( \text{spec} \) 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.

\[
\exists d_1 \bullet \text{matching}(n_1) \land \text{val}(n_1, d_1) \land \text{val}(n_2, d_1) \Rightarrow \text{spec}(n_1, n_2)
\]

\[\blacksquare\]

The same problem occurs for the other geometric predicates, so I defined the following compatibility constraints:

\[
\exists t_1 \bullet \text{succ}[\text{ss, ssnext}](t_1) \land \text{ss}(t_1, n_1) \land \text{ssnext}(t_1, n_2) \Rightarrow \text{next}(n_1, n_2)
\]

\[
\exists n_1 \bullet \text{succ}[\text{ss, ssnext}](t_1) \land \text{ss}(t_1, n_1) \land \text{next}(n_1, n_2) \Rightarrow \text{ssnext}(t_1, n_2)
\]

\[
\exists t_1 \bullet \text{succ}[n, \text{ss}](t_1) \land n(t_1, n_1) \land \text{ss}(t_1, n_2) \Rightarrow \text{next}(n_1, n_2)
\]

\[
\exists n_1 \bullet \text{succ}[n, \text{ss}](t_1) \land n(t_1, n_1) \land \text{next}(n_1, n_2) \Rightarrow \text{ss}(t_1, n_2)
\]

\[
\exists t_1 \bullet \text{opval}[n](t_1) \land n(t_1, n_1) \land \text{lv}(t_1, d_1) \Rightarrow \text{val}(n_1, d_1)
\]

\[
\exists n_1 \bullet \text{opval}[n](t_1) \land n(t_1, n_1) \land \text{val}(n_1, d_1) \Rightarrow \text{lv}(t_1, d_1)
\]

\[
\exists t_1 \bullet \text{opval}[\text{ss}](t_1) \land \text{ss}(t_1, n_1) \land \text{lv}(t_1, d_1) \Rightarrow \text{val}(n_1, d_1)
\]

\[
\exists n_1 \bullet \text{opval}[\text{ss}](t_1) \land \text{ss}(t_1, n_1) \land \text{val}(n_1, d_1) \Rightarrow \text{lv}(t_1, d_1)
\]

\[
\exists n_1, n_2 \bullet \text{commutes}(n_1) \land \text{next}(n_1, n_2) \land \text{spec}(n_1, n_3) \land \text{spec}(n_2, n_4) \Rightarrow \text{next}(n_3, n_4)
\]

\[
\exists n_1, n_3 \bullet \text{commutes}(n_1) \land \text{next}(n_1, n_2) \land \text{spec}(n_1, n_3) \land \text{next}(n_3, n_4) \Rightarrow \text{spec}(n_2, n_4)
\]

\[
\exists n_3, n_4 \bullet \text{commutes}(n_1) \land \text{spec}(n_1, n_3) \land \text{next}(n_3, n_4) \land \text{spec}(n_2, n_4) \Rightarrow \text{next}(n_1, n_2)
\]

These also rely on the functionality of \( \text{spec}, \text{ss}, \text{ssnext}, \text{next}, \text{and} \text{lv} \), and the inverse functionality of \( \text{spec} \). As with \( \text{matching} \), each has one potential constraint that would be incorrect to define, because \( \text{next} \) and \( \text{val} \) are not inversely functional.
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 num-
ber 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. How-
ever, the time taken for each analysis increases exponentially, indicating
that the compatibility constraints used to bound the threads become in-
creasingly 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.\textsuperscript{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\textsubscript{0} with null and non-null head snapshots (ss), and
threads at pop\textsubscript{38} with null and non-null next snapshots (ssnext). For the
location pop\textsubscript{35} there are threads executing the loop for the first time —
these have non-null head snapshots (ss) only. There are also threads ex-
ecuting the loop a subsequent time without the fields being reset — they
additionally have non-null lv values, and null or non-null next snapshots

\textsuperscript{8}Six threads took 40 minutes to complete; seven threads took 30 days. The model with
eight threads was manually killed after 10 months.
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.
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
7.8. STACK VARIATIONS

Table 7.5: Stack analyses with unnecessary referenced-by instrumentation predicates

<table>
<thead>
<tr>
<th>Th.</th>
<th>States (uns.)</th>
<th>States (stored)</th>
<th>States (total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>56</td>
<td>326</td>
<td>904</td>
</tr>
<tr>
<td>2</td>
<td>&gt;20,240</td>
<td>&gt;160,168</td>
<td>&gt;374,000</td>
</tr>
</tbody>
</table>

Table 7.5 shows that for one thread the extra instrumentation predicates quadruple the statespace. For two threads TVLA runs out of memory.
CHAPTER 7. C. A. FOR LINEARISABILITY

Table 7.6: Stack analyses with a pure initial state

<table>
<thead>
<tr>
<th>Th.</th>
<th>States (uns.)</th>
<th>%</th>
<th>States (stored)</th>
<th>%</th>
<th>States (total)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>54</td>
<td>386</td>
<td>320</td>
<td>372</td>
<td>602</td>
<td>288</td>
</tr>
<tr>
<td>2</td>
<td>407</td>
<td>316</td>
<td>4,037</td>
<td>308</td>
<td>9,936</td>
<td>233</td>
</tr>
<tr>
<td>3</td>
<td>1,271</td>
<td>289</td>
<td>18,347</td>
<td>283</td>
<td>49,825</td>
<td>230</td>
</tr>
<tr>
<td>4</td>
<td>2,494</td>
<td>273</td>
<td>49,358</td>
<td>266</td>
<td>137,571</td>
<td>205</td>
</tr>
<tr>
<td>∞</td>
<td>4,847</td>
<td>254</td>
<td>179,246</td>
<td>242</td>
<td>532,515</td>
<td>195</td>
</tr>
</tbody>
</table>

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.
7.8. STACK VARIATIONS

<table>
<thead>
<tr>
<th>Th.</th>
<th>States (uns.)</th>
<th>States (stored)</th>
<th>States (total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>234</td>
<td>2,364</td>
<td>6,769</td>
</tr>
<tr>
<td>3</td>
<td>2,472</td>
<td>34,305</td>
<td>101,470</td>
</tr>
<tr>
<td>4</td>
<td>&gt;10,206</td>
<td>&gt;196,410</td>
<td>&gt;366,000</td>
</tr>
</tbody>
</table>

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 \( \text{bound} + 1 \) thread objects is invalid”) compared to the results from Table 7.1 which used the optimised version (“if there are \( \text{bound} \) 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.
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 7.8 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.

Table 7.8: Stack verification results for bounded threads

<table>
<thead>
<tr>
<th>Th.</th>
<th>Heap Limit (MB)</th>
<th>Ave Time (s)</th>
<th>Ave RAM (MB)</th>
<th>Max RAM (MB)</th>
<th>%</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>800</td>
<td>18</td>
<td>96</td>
<td>108</td>
<td>114</td>
<td>282</td>
</tr>
<tr>
<td>3</td>
<td>800</td>
<td>108</td>
<td>106</td>
<td>172</td>
<td>100</td>
<td>342</td>
</tr>
<tr>
<td>4</td>
<td>800</td>
<td>568</td>
<td>106</td>
<td>256</td>
<td>100</td>
<td>463</td>
</tr>
<tr>
<td>5</td>
<td>2,048</td>
<td>6,532</td>
<td>102</td>
<td>522</td>
<td>104</td>
<td>1,095</td>
</tr>
</tbody>
</table>

(a) Resources

<table>
<thead>
<tr>
<th>Th.</th>
<th>Unsch. Stored States (uns.)</th>
<th>Stored States (stored)</th>
<th>Total States (total)</th>
<th>% Breaches</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>129 +0</td>
<td>1,312 +0</td>
<td>4,569</td>
<td>107</td>
<td>23,302</td>
</tr>
<tr>
<td>3</td>
<td>440 +0</td>
<td>6,498 +5</td>
<td>24,669</td>
<td>114</td>
<td>140,510</td>
</tr>
<tr>
<td>4</td>
<td>915 +0</td>
<td>18,573 +9</td>
<td>72,822</td>
<td>108</td>
<td>443,955</td>
</tr>
<tr>
<td>5</td>
<td>1,398 +0</td>
<td>36,756 +7</td>
<td>140,677</td>
<td>104</td>
<td>881,964</td>
</tr>
</tbody>
</table>

(b) Statespace
7.8. STACK VARIATIONS

Figure 7.18: Transitions of stack model

<table>
<thead>
<tr>
<th>Th.</th>
<th>States (uns.)</th>
<th>%</th>
<th>States (stored)</th>
<th>%</th>
<th>States (total)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45</td>
<td>321</td>
<td>148</td>
<td>172</td>
<td>338</td>
<td>162</td>
</tr>
<tr>
<td>2</td>
<td>1,511</td>
<td>1,171</td>
<td>7,731</td>
<td>589</td>
<td>25,498</td>
<td>597</td>
</tr>
<tr>
<td>3</td>
<td>21,107</td>
<td>4,797</td>
<td>148,191</td>
<td>2,282</td>
<td>567,500</td>
<td>2,618</td>
</tr>
<tr>
<td>4</td>
<td>&gt;28,849</td>
<td>&gt;3,153</td>
<td>&gt;282,441</td>
<td>&gt;1,521</td>
<td>&gt;460,800</td>
<td>&gt;685</td>
</tr>
</tbody>
</table>

Table 7.9: Stack verification results with no partial order reduction
Chapter 7. C. A. for Linearisability

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:

- is_data,
- is_node, next, val, HeadI, TailI, HeadS, TailS,
- is_thread, doneLP, doneELP, lv, n, sshead, sstail, ssnext

For both algorithms we define the following location predicates:

- at[idle], at[enq3], at[enq4], at[enq6], at[enq7], at[enq8],
  at[enq9], at[enq11], at[enq13], at[enq20], at[enq24], at[enq25]

For the original algorithm only we define the following location predicates:

- at[deq30], at[deq31], at[deq33], at[deq34], at[deq35],
  at[deq40], at[deq41], at[deq42], at[deq43], at[deq46],
  at[deq48], at[deq50], at[deq52], at[deq61]
atomic{
    idle    beginEnqueue()       enq3
    enq3    newNode(n)          enq4
    enq4    assign(n.val, lv)   enq6
    enq6    assign(sstail, Tail) enq7
  }
atomic{
    enq7    assign(ssnext, sstail.next) enq8
  }
atomic{
    enq8    isEqual(sstail, Tail)     enq9
    enq8    isNotEqual(sstail, Tail)  enq6
    enq9    isNull(ssnext)            enq11
    enq9    isNotNull(ssnext)         enq20
  }
atomic{
    enq11   CASfail(Tail, sstail)    enq6
    enq20   CASsuc(Tail, sstail, ssnext) enq6
  }
atomic{
    enq11   CASfail(sstail.next, ssnext) enq6
    enq11   CASsuc(sstail.next, ssnext, n) enq13
    enq13   specEnqueue()            enq24
  }
atomic{
    enq24   CASfail(Tail, sstail)    enq25
    enq24   CASsuc(Tail, sstail, n)  enq25
    enq25   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$_{40}$
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$_{46}$ CASsuc(Tail, sstail, ssnext) deq$_{30}$
}
atomic{
    deq$_{48}$ assign(lv, ssnext.val) deq$_{50}$
deq$_{50}$ CASfail(Head, sshead) deq$_{30}$
deq$_{50}$ CASsuc(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 →
    doneLP := false; lv ∈ DATA

specEnqueue() ¬ doneLP →
    doneLP := true;
    spec.Tail.next := newNode(lv, null);
    spec.Tail := spec.Tail.next

delEnqueue() doneLP →
    lv, sstail, ssnext, n := null

beginDequeue() true →
    doneLP, doneELP := false; lv := null

specDequeueEmpty() ¬ doneLP ∧ spec.Head = spec.Tail →
    doneELP := true

specDequeue() ¬ doneLP ∧ spec.Head ≠ spec.Tail ∧ spec.Head.next.val = lv →
    doneLP := true; spec.Head := spec.Head.next

delDequeueEmpty() doneELP ∧ ¬ doneLP →
    lv, sshead, sstail, ssnext := null

delDequeue() doneLP →
    lv, sshead, sstail, ssnext := null

Figure 7.20: Additional update operations used in queue models

For the simplified algorithm only we define the following location predicates:

at[deq67], at[deq69], at[deq70], at[deq71], at[deq76],
at[deq77], at[deq78], at[deq81], at[deq83], at[deq85],
at[deq94], at[deq95], at[deq96], at[deq98]

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:

- \text{has}[\text{val}], \text{has}[\text{next}], \text{reach}_h, \text{reach}_S, \text{circ},
- \text{r_by}[\text{spec}], \text{matching}, \text{commutes}, \text{has}_S[\text{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:

- \text{has}[\text{lv}], \text{has}[\text{n}], \text{waiting}, \text{shared}[\text{n}],
- \text{has}[\text{sshead}], \text{has}[\text{sstail}], \text{has}[\text{ssnext}]

The queues have similar properties that require geometric predicates to preserve sufficient information in the abstract states. In the CAS steps at locations \text{enq}_{20}, \text{enq}_{24}, \text{deq}_{46}, \text{deq}_{50}, \text{deq}_{83} \text{and} \text{deq}_{96}, \text{the old and new values are implicitly assumed to be \text{next}-successors. Thus I defined the predicates:}

- \text{succ}[\text{sshead}, \text{ssnext}], \text{succ}[\text{sstail}, \text{ssnext}], \text{succ}[\text{sstail}, \text{n}]

In the specification actions for enqueues and non-empty dequeues, the value of \( \text{lv} \) is assumed to remain the same as the \( \text{val} \) of the node pointed to by \( \text{n} \) or \( \text{ssnext} \), respectively. Thus, I defined the predicates:

- \text{opval}[\text{n}], \text{opval}[\text{ssnext}]

One property requires a novel predicate not defined so far. After the comparison at locations \text{deq}_{41}/\text{deq}_{95} the equality or otherwise of \( \text{sshead} \) and \( \text{sstail} \) is lost by the abstraction. I defined the following instrumentation predicate, which allows the property to be retained in the canonically abstract states.

\[
\text{same}[\text{sshead}, \text{sstail}](t) \iff \exists n \cdot \text{sshead}(t, n) \land \text{sstail}(t, n)
\]
CHAPTER 7. C. A. FOR LINEARISABILITY

Table 7.10: Queue verification results

<table>
<thead>
<tr>
<th>Deq</th>
<th>Th.</th>
<th>Heap Limit (MB)</th>
<th>Time (s)</th>
<th>Ave RAM (MB)</th>
<th>Max RAM (MB)</th>
<th>Uns. States</th>
<th>Stored States</th>
<th>Total States</th>
<th>Breaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>1</td>
<td>800</td>
<td>1</td>
<td>13</td>
<td>30</td>
<td>26</td>
<td>115</td>
<td>227</td>
<td>345</td>
</tr>
<tr>
<td>O</td>
<td>2</td>
<td>800</td>
<td>393</td>
<td>260</td>
<td>476</td>
<td>3,770</td>
<td>24,271</td>
<td>73,874</td>
<td>849,174</td>
</tr>
<tr>
<td>O</td>
<td>3</td>
<td>2,048</td>
<td>&gt;332k</td>
<td>&gt;25k</td>
<td>&gt;235k</td>
<td>&gt;564k</td>
<td>&gt;25k</td>
<td>&gt;235k</td>
<td>&gt;564k</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>800</td>
<td>1</td>
<td>14</td>
<td>33</td>
<td>26</td>
<td>117</td>
<td>228</td>
<td>361</td>
</tr>
<tr>
<td>S</td>
<td>2</td>
<td>800</td>
<td>83</td>
<td>189</td>
<td>354</td>
<td>1,506</td>
<td>10,746</td>
<td>28,856</td>
<td>147,080</td>
</tr>
<tr>
<td>S</td>
<td>3</td>
<td>2,048</td>
<td>&gt;22k</td>
<td>&gt;230k</td>
<td>&gt;632k</td>
<td>&gt;22k</td>
<td>&gt;230k</td>
<td>&gt;632k</td>
<td></td>
</tr>
</tbody>
</table>

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:

\[
\text{same[sshead, sstail]}(t) \land \text{sshead}(t, n) \supset \text{sstail}(t, n)
\]

\[
\text{same[sshead, sstail]}(t) \land \text{sstail}(t, n) \supset \text{sshead}(t, n)
\]

7.9.2 Results

I analyzed 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\text{c} and Tail\text{c} 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\text{c} at the end, there are fewer ways to distinguish its thread objects — the instrumentation predicates defined using \text{sstail} are all false for the 11 locations deq\text{c}.
7.10. RELATED WORK

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 state spaces 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.

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.

---

10For 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-
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 state-spaces 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 state-space 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 state-space 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 state-space 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 push₉:

- hasₙ is always true
- hasₙssnext is always false
- succₙss is true if hasₙss 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 $A$ (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 push₉ 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 locₓ, then property $ϕ$ holds”.

Thus, I defined instrumentation predicates, called soft invariants, of the form:

$$\text{if}[\text{loc}_x, p](v) \leftrightarrow \text{is\_thread}(v) \land (\text{at}[\text{loc}_x](v) \rightarrow ϕ_p(v))$$
8.1. **OVERVIEW**

Figure 8.1: Collapsing threads: loss of precision
To record the above three properties, I defined the following three soft invariant instrumentation predicates:

$$\text{if} [\text{push}_9, \text{has}[n]](v) \iff$$
$$\text{is\_thread}(v) \land (\text{at}[\text{push}_9] \rightarrow \text{has}[n](v))$$

$$\text{if} [\text{push}_9, \text{has}[\text{ss\_next}]](v) \iff$$
$$\text{is\_thread}(v) \land (\text{at}[\text{push}_9] \rightarrow \neg \text{has}[\text{ss\_next}](v))$$

$$\text{if} [\text{push}_9, \text{succ}[n, \text{ss}]](v) \iff$$
$$\text{is\_thread}(v) \land (\text{at}[\text{push}_9] \rightarrow (\text{has}[\text{ss}](v) \rightarrow \text{succ}[n, \text{ss}](v)))$$

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:

$$\text{if} [\text{push}_9, \text{has}[\text{ss}]](v) \iff \text{is\_thread}(v) \rightarrow (\text{at}[\text{push}_9](v) \rightarrow \text{has}[\text{ss}](v))$$

I do not include the soft invariant predicates in $A$, so when the threads are collapsed (8.3b), $\text{if} [\text{push}_9, \text{has}[\text{ss}]]$ becomes unknown, because the property it is trying to preserve is not actually invariant. Consequently, when a thread at $\text{push}_9$ is concretised, $\text{has}[\text{ss}]$ 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
8.1. OVERVIEW

(a) Three previously canonical thread objects

(b) Collapsed thread object

Figure 8.2: Collapsing threads: properties preserved with soft invariants
Figure 8.3: Collapsing threads: soft invariant for non-invariant property
8.1. OVERVIEW

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 pre-
served in the abstraction by other predicates, and when the property be-
ing 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 invari-
ant 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 gener-
ated by TVLA cannot deduce this, as their bodies all evaluate to unknown:

\[
\begin{align*}
is\_thread(v) & \land (at[push_9](v) \rightarrow has[n](v)) \\
\triangleright & \text{ if}[push_9, has[n]](v) \\
\neg (is\_thread(v) & \land (at[push_9](v) \rightarrow has[n](v))) \\
\triangleright & \neg \text{ if}[push_9, has[n]](v)
\end{align*}
\]
is_thread(v) \land \text{at}[\text{push}_9](v) \to \neg \text{has}[\text{ssnext}](v) \\
\triangleright \text{if}[\text{push}_9, \text{has}[\text{ssnext}]](v) \\
\neg (\text{is_thread}(v) \land \text{at}[\text{push}_9](v) \to \neg \text{has}[\text{ssnext}](v)) \\
\triangleright \neg \text{if}[\text{push}_9, \text{has}[\text{ssnext}]](v) \\
is_thread(v) \land \text{at}[\text{push}_9](v) \to \text{has}[\text{ss}](v) \to \text{succ}[n, \text{ss}](v)) \\
\triangleright \text{if}[\text{push}_9, \text{succ}[n, \text{ss}]](v) \\
\neg (\text{is_thread}(v) \land \text{at}[\text{push}_9](v) \to \text{has}[\text{ss}](v) \to \neg \text{has}[\text{ssnext}](v)) \\
\triangleright \neg \text{if}[\text{push}_9, \text{has}[\text{ssnext}]](v) \\
is_thread(v) \land \text{at}[\text{push}_9](v) \to \neg \text{has}[\text{ssnext}](v) \\
\triangleright \text{if}[\text{push}_9, \text{has}[\text{ssnext}]](v) \\
\neg (\text{is_thread}(v) \land \text{at}[\text{push}_9](v) \to \text{has}[\text{ss}](v) \to \text{succ}[n, \text{ss}](v))) \\
\triangleright \neg \text{if}[\text{push}_9, \text{succ}[n, \text{ss}]](v)

I defined compatibility constraints of the following forms:

\text{if}[\text{push}_9, \text{has}[\text{n}]](v) \land \text{at}[\text{push}_9](v) \triangleright \text{has}[\text{n}](v) \\
\text{if}[\text{push}_9, \text{has}[\text{ssnext}]](v) \land \text{at}[\text{push}_9](v) \triangleright \neg \text{has}[\text{ssnext}](v) \\
\text{if}[\text{push}_9, \text{succ}[\text{n}, \text{ss}]](v) \land \text{at}[\text{push}_9](v) \land \text{has}[\text{ss}](v) \triangleright \text{succ}[\text{n}, \text{ss}](v)

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 \text{is_thread} and \text{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 \( \text{loc}_1 \) and \( \text{loc}_2 \), where \( \text{loc}_1 \neq \text{loc}_2 \):

\[
\text{at}[\text{loc}_1](v) \rightarrow \neg \text{at}[\text{loc}_2](v)
\]

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 \( \text{succ}[n, ss] \) and \( \text{succ}[ss, ssnext] \). 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 \( \text{has}[ssnext] \) true, the predicate \( \text{succ}[ss, ssnext] \) records that \( \text{ssnext} = \text{ss.next} \). In the right-hand thread object, with \( \text{has}[ssnext] \) false, the distinction of the thread objects and the values of the ss predicate record that \( \text{ssnext} = \text{ss.next} \) (i.e. null). Indeed, in every state of the model, a thread at pop\(_{38}\) with \( \text{has}[ssnext] \) false has an ss predicate that points to a node with \( \text{has}[next] \) 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:

\[
\text{has}[ssnext](t) \rightarrow \text{succ}[ss, ssnext](t)\\
\neg \text{has}[ssnext](t) \rightarrow \exists n \cdot ss(t, n) \land \neg \text{has}[next](n)
\]

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:

\[
\text{tonull}[ss](v) \Leftrightarrow \exists u \cdot ss(v, u) \land \neg \text{has}[next](u)
\]
Figure 8.5: Property of null equivalence is lost when threads are collapsed
8.2. STACK MODELS

For the same reasons at push\textsubscript{9}, I also defined an equivalent instrumentation predicate:

\[
\text{tonull}[n](v) \Leftrightarrow \exists u \cdot n(v,u) \land \neg \text{has}[\text{next}](u)
\]

Both of the new instrumentation predicates require additional compatibility constraints:

\[
\exists t \cdot \text{tonull}[\text{ss}](t) \land \text{ss}(t,n) \triangleright \neg \text{has}[\text{next}](n)
\]

\[
\exists t \cdot \text{tonull}[n](t) \land n(t,n) \triangleright \neg \text{has}[\text{next}](n)
\]

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 \texttt{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 \text{tonull}[n] and \text{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;
- \( \neg p \) 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 \text{highlighted} entries are the ones I determined were necessary to record and preserve in the abstraction. For example, if \text{doneLP} can be sharpened to true at push\textsubscript{9} or pop\textsubscript{38}, then the specification step after the CAS will give an error; if \text{doneLP} can be sharpened to true at pop\textsubscript{35}, then it can be true.
Table 8.1: Invariant properties of stack model with restricted interleaving

<table>
<thead>
<tr>
<th></th>
<th>idle</th>
<th>push\textsubscript{9}</th>
<th>pop\textsubscript{35}</th>
<th>pop\textsubscript{38}</th>
</tr>
</thead>
<tbody>
<tr>
<td>doneLP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>has[lv]</td>
<td>0</td>
<td>(1)</td>
<td>(\frac{1}{2})</td>
<td>(1)</td>
</tr>
<tr>
<td>has[n]</td>
<td>0</td>
<td>(1)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>opval[n]</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tonull[n]</td>
<td>0</td>
<td>\neg has[ss]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>succ[n, ss]</td>
<td>0</td>
<td>has[ss]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>has[ss]</td>
<td>0</td>
<td>\frac{1}{2}</td>
<td>1</td>
<td>(1)</td>
</tr>
<tr>
<td>opval[ss]</td>
<td>0</td>
<td>\frac{1}{2}</td>
<td>\frac{1}{2}</td>
<td>1</td>
</tr>
<tr>
<td>has[ssnext]</td>
<td>0</td>
<td>0</td>
<td>\frac{1}{2}</td>
<td>\frac{1}{2}</td>
</tr>
<tr>
<td>tonull[ss]</td>
<td>0</td>
<td>\frac{1}{2}</td>
<td>\frac{1}{2}</td>
<td>\neg has[ssnext]</td>
</tr>
<tr>
<td>succ[ss, ssnext]</td>
<td>0</td>
<td>0</td>
<td>\frac{1}{2}</td>
<td>has[ssnext]</td>
</tr>
</tbody>
</table>

at pop\textsubscript{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\textsubscript{9}, opval[n]] means that additionally defining if[push\textsubscript{9}, has[lv]] and if[push\textsubscript{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:

\[
\text{waiting}(v) \Leftrightarrow \exists t \cdot n(t, v) \land \neg \text{doneLP}(t)
\]

Thus, whenever doneLP is updated for a thread, the value of waiting must
if \( \text{idle, has}[n]\)(v) ⇔ is\_thread(v) \land (\text{at}[\text{idle}](v) \rightarrow \neg \text{has}[n](v)) \\
if \( \text{push}_9, \text{doneLP}\)(v) ⇔ is\_thread(v) \land (\text{at}[\text{push}_9](v) \rightarrow \neg \text{doneLP}(v)) \\
if \( \text{push}_9, \text{opval}[n]\)(v) ⇔ is\_thread(v) \land (\text{at}[\text{push}_9](v) \rightarrow \text{opval}[n](v)) \\
if \( \text{push}_9, \text{succ}[n, ss]\)(v) ⇔ \\
\quad \text{is\_thread}(v) \land (\text{at}[\text{push}_9](v) \rightarrow (\text{has}[ss](v) \rightarrow \text{succ}[n, ss](v))) \\
if \( \text{push}_9, \text{tonull}[n]\)(v) ⇔ \\
\quad \text{is\_thread}(v) \land (\text{at}[\text{push}_9](v) \rightarrow (\neg \text{has}[ss](v) \rightarrow \text{tonull}[n](v))) \\
if \( \text{pop}_{35}, \text{doneLP}\)(v) ⇔ is\_thread(v) \land (\text{at}[\text{pop}_{35}](v) \rightarrow \neg \text{doneLP}(v)) \\
if \( \text{pop}_{35}, \text{has}[n]\)(v) ⇔ is\_thread(v) \land (\text{at}[\text{pop}_{35}](v) \rightarrow \neg \text{has}[n](v)) \\
if \( \text{pop}_{35}, \text{has}[ss]\)(v) ⇔ is\_thread(v) \land (\text{at}[\text{pop}_{35}](v) \rightarrow \text{has}[ss](v)) \\
if \( \text{pop}_{36}, \text{succ}[ss, ssnext]\)(v) ⇔ \\
\quad \text{is\_thread}(v) \land (\text{at}[\text{pop}_{36}](v) \rightarrow (\text{has}[ssnext](v) \rightarrow \text{succ}[ss, ssnext](v))) \\
if \( \text{pop}_{36}, \text{tonull}[ss]\)(v) ⇔ \\
\quad \text{is\_thread}(v) \land (\text{at}[\text{pop}_{36}](v) \rightarrow (\neg \text{has}[ssnext](v) \rightarrow \text{tonull}[ss](v))) \\
if \( \text{pop}_{38}, \text{doneLP}\)(v) ⇔ is\_thread(v) \land (\text{at}[\text{pop}_{38}](v) \rightarrow \neg \text{doneLP}(v)) \\
if \( \text{pop}_{38}, \text{has}[n]\)(v) ⇔ is\_thread(v) \land (\text{at}[\text{pop}_{38}](v) \rightarrow \neg \text{has}[n](v)) \\
if \( \text{pop}_{38}, \text{opval}[ss]\)(v) ⇔ is\_thread(v) \land (\text{at}[\text{pop}_{38}](v) \rightarrow \text{opval}[ss](v)) \\
if \( \text{pop}_{38}, \text{succ}[ss, ssnext]\)(v) ⇔ \\
\quad \text{is\_thread}(v) \land (\text{at}[\text{pop}_{38}](v) \rightarrow (\text{has}[ssnext](v) \rightarrow \text{succ}[ss, ssnext](v))) \\
if \( \text{pop}_{38}, \text{tonull}[ss]\)(v) ⇔ \\
\quad \text{is\_thread}(v) \land (\text{at}[\text{pop}_{38}](v) \rightarrow (\neg \text{has}[ssnext](v) \rightarrow \text{tonull}[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 \( \text{has}[n] \) is not preserved as false by a soft invariant, then it is possible to concretise unreachable states when \( \text{pop} \) 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, \( \text{push}_3 \) and \( \text{push}_4 \) (8.7a). If one of the idle threads begins a push operation then one outcome of performing Focus with the formula

\[
\text{tr\_scheduled}(v_1) \land \text{at}[\text{idle}](v_1) \land n(v_1, v_2)
\]

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 \( \text{has}[n] \) at each of the interleaving locations.
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 $n_{next} = ss$ at push$_9$ and $ss_{next} = ssnext$ 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:

\[
\text{if}[\text{pop}_{36}, \text{succ}[ss, ssnext]](v) \iff \\
\quad \text{is\_thread}(v) \land (\text{at}[\text{pop}_{36}](v) \rightarrow (\text{has}[ssnext](v) \rightarrow \text{succ}[ss, ssnext](v)))
\]

\[
\text{if}[\text{pop}_{36}, \text{tonull}[ss]](v) \iff \\
\quad \text{is\_thread}(v) \land (\text{at}[\text{pop}_{36}](v) \rightarrow (\neg \text{has}[ssnext](v) \rightarrow \text{tonull}[ss](v)))
\]

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 state-space, 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
CHAPTER 8. COLLAPSING THREADS SAFELY

Table 8.2: Stack verification results for restricted interleaving

<table>
<thead>
<tr>
<th>Th.</th>
<th>Heap Limit (MB)</th>
<th>Ave Time (s)</th>
<th>Ave RAM % (MB)</th>
<th>Max RAM % (MB)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>800</td>
<td>25</td>
<td>1.3</td>
<td>115</td>
<td>17.8</td>
</tr>
<tr>
<td>∞</td>
<td>2,048</td>
<td>24</td>
<td>1.2</td>
<td>190</td>
<td>29.3</td>
</tr>
</tbody>
</table>

(a) Resources

<table>
<thead>
<tr>
<th>Th.</th>
<th>Uns. States (uns.) %</th>
<th>Stored States %</th>
<th>Total States %</th>
<th>Breaches %</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>16</td>
<td>0.8</td>
<td>447</td>
<td>0.6</td>
</tr>
</tbody>
</table>

(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 push₁₁, pop₂₅, pop₂₆ and pop₄₀. 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 push₃–push₇, pop₂₄ and pop₃₆ the properties are the same as for push₉ and pop₃₈, from when they are first set. For push₁₈ and pop₄₈, 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
Table 8.3: Invariant properties of stack with full interleaving
Table 8.2: Stack verification results for full interleaving

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-
defined 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.

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 \( \text{loc}_X \):

\[
\begin{align*}
\text{if}[\text{loc}_X, p](v) & \iff \text{is\_thread}(v) \land (\text{at}[\text{loc}_X](v) \rightarrow p(v)) \\
\text{nf}[\text{loc}_X, p](v) & \iff \text{is\_thread}(v) \land (\text{at}[\text{loc}_X](v) \rightarrow \neg p(v))
\end{align*}
\]

I kept the six conditional soft invariants, prefixing them with a ‘c’:

\[
\begin{align*}
c\text{if}[\text{push}_9, \text{tonull}[n]],& \ c\text{if}[\text{push}_9, \text{succ}[n, ss]], \\
c\text{if}[\text{pop}_36, \text{tonull}[ss]],& \ c\text{if}[\text{pop}_36, \text{succ}[ss, ssnext]], \\
c\text{if}[\text{pop}_38, \text{tonull}[ss]],& \ c\text{if}[\text{pop}_38, \text{succ}[ss, ssnext]]
\end{align*}
\]

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.

---

1Assuming the property is correct.


### 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 tonull[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<sub>i</sub>, which is lagging. Other dequeuing threads are about to attempt the CAS step at deq<sub>50</sub> to advance Head<sub>i</sub>. At this location we assume that sshead and sstail differ, given the test at deq<sub>41</sub>, and that

---

### Table 8.5: Stack verification results for full interleaving with extra predicates

<table>
<thead>
<tr>
<th>Th.</th>
<th>Heap Limit (MB)</th>
<th>Ave Time (s)</th>
<th>Ave RAM % (MB)</th>
<th>Max RAM % (MB)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>800</td>
<td>278</td>
<td>197</td>
<td>202</td>
<td>136</td>
</tr>
<tr>
<td>∞</td>
<td>2,048</td>
<td>264</td>
<td>186</td>
<td>375</td>
<td>147</td>
</tr>
</tbody>
</table>

(a) Resources

<table>
<thead>
<tr>
<th>Th.</th>
<th>Unsched. States (uns.)</th>
<th>Stored States % (stored)</th>
<th>Total States % (total)</th>
<th>% Breaches</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>32</td>
<td>726</td>
<td>5,850</td>
<td>104</td>
<td>59</td>
</tr>
</tbody>
</table>

(b) Statespace
### 8.3. QUEUE MODELS

<table>
<thead>
<tr>
<th>State</th>
<th>Transition Description</th>
<th>Enqueue No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td>beginEnqueue()</td>
<td>enq3</td>
</tr>
<tr>
<td>enq3</td>
<td>newNode(n)</td>
<td>enq4</td>
</tr>
<tr>
<td>enq4</td>
<td>assign(n.val, lv)</td>
<td>enq6</td>
</tr>
<tr>
<td>enq6</td>
<td>assign(sstail, Tail)</td>
<td>enq7</td>
</tr>
<tr>
<td>enq7</td>
<td>assign(ssnext, sstail.next)</td>
<td>enq8</td>
</tr>
<tr>
<td>enq8</td>
<td>isEqual(sstail, Tail)</td>
<td>enq9</td>
</tr>
<tr>
<td>enq9</td>
<td>isNull(ssnext)</td>
<td>enq11</td>
</tr>
<tr>
<td>enq11</td>
<td>isEqual(sstail, Tail)</td>
<td>enq6</td>
</tr>
<tr>
<td>enq6</td>
<td>isNull(ssnext)</td>
<td>enq11</td>
</tr>
<tr>
<td>enq11</td>
<td>CASfail(Tail, sstail)</td>
<td>enq6</td>
</tr>
<tr>
<td>enq6</td>
<td>CASfail(Tail, sstail, ssnext)</td>
<td>enq6</td>
</tr>
<tr>
<td>enq13</td>
<td>specEnqueue()</td>
<td>enq24</td>
</tr>
<tr>
<td>enq24</td>
<td>CASfail(Tail, sstail)</td>
<td>enq25</td>
</tr>
<tr>
<td>enq24</td>
<td>CASsucc(Tail, sstail, ssnext)</td>
<td>enq25</td>
</tr>
<tr>
<td>enq25</td>
<td>endEnqueue()</td>
<td>idle</td>
</tr>
</tbody>
</table>

(a) Enqueue operation

Figure 8.8: Transitions of queue models
idle  beginDequeue()  deq

deq assign(sshead, Head)  deq
deq assign(sstail, Tail)  deq

atomic{
  deq assign(ssnext, sshead.next)  deq
  deq isNull(ssnext)  deq
  deq isNotNull(ssnext)  deq
  deq specDequeueEmpty()  deq
}

deq isEqual(sshead, Head)  deq
deq isNotEqual(sshead, Head)  deq

deq isEqual(sshead, sstail)  deq
deq isNotEqual(sshead, sstail)  deq

deq isNull(ssnext)  deq
deq isNotNull(ssnext)  deq

deq endDequeueEmpty()  idle

atomic{
  deq CASfail(Tail, sstail)  deq
  deq CASsucc(Tail, sstail, ssnext)  deq
  deq assign(lv, ssnext.val)  deq
  deq CASfail(Head, sshead)  deq
}

deq specDequeue()  deq

deq endDequeue()  idle

(b) Original Dequeue operation

Figure 8.8: Transitions of queue models
### 8.3. QUEUE MODELS

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td>beginDequeue()</td>
<td>deq67</td>
</tr>
<tr>
<td>deq67</td>
<td>assign(sshead, Head)</td>
<td>deq69</td>
</tr>
<tr>
<td>atomic{</td>
<td>assign(ssnext, sshead.next)</td>
<td>deq70</td>
</tr>
<tr>
<td>deq70</td>
<td>isNull(ssnext)</td>
<td>deq71</td>
</tr>
<tr>
<td>deq70</td>
<td>isNotNull(ssnext)</td>
<td>deq76</td>
</tr>
<tr>
<td>deq71</td>
<td>specDequeueEmpty()</td>
<td>deq76</td>
</tr>
<tr>
<td>}</td>
<td>isEqual(sshead, Head)</td>
<td>deq77</td>
</tr>
<tr>
<td>deq76</td>
<td>isNotEqual(sshead, Head)</td>
<td>deq67</td>
</tr>
<tr>
<td>deq77</td>
<td>isNull(ssnext)</td>
<td>deq78</td>
</tr>
<tr>
<td>deq77</td>
<td>isNotNull(ssnext)</td>
<td>deq81</td>
</tr>
<tr>
<td>deq78</td>
<td>endDequeueEmpty()</td>
<td>idle</td>
</tr>
<tr>
<td>deq81</td>
<td>assign(lv, ssnext.val)</td>
<td>deq83</td>
</tr>
<tr>
<td>deq83</td>
<td>CASfail(Head, sshead)</td>
<td>deq67</td>
</tr>
<tr>
<td>atomic{</td>
<td>CASsucc(Head, sshead, ssnext)</td>
<td>deq85</td>
</tr>
<tr>
<td>deq85</td>
<td>specDequeue()</td>
<td>deq94</td>
</tr>
<tr>
<td>}</td>
<td>assign(sstail, Tail)</td>
<td>deq95</td>
</tr>
<tr>
<td>deq94</td>
<td>isEqual(sshead, sstail)</td>
<td>deq96</td>
</tr>
<tr>
<td>deq95</td>
<td>isNotEqual(sshead, sstail)</td>
<td>deq98</td>
</tr>
<tr>
<td>deq96</td>
<td>CASfail(Tail, sstail)</td>
<td>deq98</td>
</tr>
<tr>
<td>deq96</td>
<td>CASsucc(Tail, sstail, ssnext)</td>
<td>deq98</td>
</tr>
<tr>
<td>deq98</td>
<td>endDequeue()</td>
<td>idle</td>
</tr>
</tbody>
</table>

(c) Simplified Dequeue operation

Figure 8.8: Transitions of queue models
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 ssstail is pointing to a 'garbage' node and sshead is pointing to Head, thus the CAS will succeed and Head will "cross" Tail — not expected behaviour!

The property that is being lost in the above example is that sshead and ssstail never "cross", alternatively that ssstail is reachable from sshead:

$$\forall v_1, v_2 \cdot \text{sshead}(t, v_1) \land \text{ssstail}(t, v_2) \rightarrow \text{next}^*(v_1, v_2)$$
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 ssstail is too. Thus, I defined the following new instrumentation predicate:

\[
\text{ordered}[\text{sshead, ssstail}] (v_1) \iff \exists v_2, v_3 \bullet
\text{sshead}(v_1, v_2) \land \text{ssstail}(v_1, v_3) \land
(\text{reach}_I(v_2) \rightarrow \text{reach}_I(v_3))
\]

Additionally, I defined the following compatibility constraint:

\[
\exists v_1, v_2 \bullet \text{ordered}[\text{sshead, ssstail}] (v_1) \land \text{sshead}(v_1, v_2) \land
\text{ssstail}(v_1, n_3) \land \text{reach}_I(v_2) \triangleright \text{reach}_I(n_3)
\]

With \text{ordered}[\text{sshead, ssstail}] defined, the state in Figure 8.9b would be discarded.

In the simplified dequeue operation (Figure 8.8c), ssstail is not read until after the CAS step at \text{deq}_83, and Head\_I and Tail\_I are actually allowed to cross at this point (see Section 2.6.2). Thus, \text{ordered}[\text{sshead, ssstail}] 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 \text{waiting}, so as with the stack models I defined soft invariants for \text{has}[n] at idle and throughout the dequeue operations.
\begin{table}[h]
\centering
\begin{tabular}{l|ccccccccccc}
& idle & enq& enq& enq& enq& enq& enq& enq& enq& enq& enq \\
\hline
\text{doneLP} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\text{doneELP} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{has[lv]} & 0 & 1 & 1 & (1) & (1) & (1) & (1) & (1) & (1) & 1 & 1 \\
\text{has[n]} & 0 & 0 & 1 & (1) & (1) & (1) & (1) & (1) & (1) & 1 & 1 \\
\text{opval[n]} & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & (1) & 1 \\
\text{has[sshead]} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{has[ssnext]} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{same[sshead, ssnext]} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{has[ssnext]} & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & 0 & (1) & 0 & 0 \\
\text{succ[ssnext, n]} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\text{succ[sshead, ssnext]} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{succ[ssnext, ssnext]} & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \psi_1 & \psi_1 \\
\text{opval[ssnext]} & 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} & \frac{1}{2} \\
\text{ordered[sshead, ssnext]} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{tabular}
\caption{Table 8.6: Invariant properties}
\end{table}
<table>
<thead>
<tr>
<th></th>
<th>(\text{deq}^{0})</th>
<th>(\text{deq}^{1})</th>
<th>(\text{deq}^{2})</th>
<th>(\text{deq}^{4})</th>
<th>(\text{deq}^{5})</th>
<th>(\text{deq}^{6})</th>
<th>(\text{deq}^{7})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{doneLP})</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(\text{doneELP})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
</tr>
<tr>
<td>(\text{has}[lv])</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
</tr>
<tr>
<td>(\text{has}[n])</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(\text{opval}[n])</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(\text{has}[sshead])</td>
<td>(\frac{1}{2})</td>
<td>1</td>
<td>1</td>
<td>(1)</td>
<td>1</td>
<td>1</td>
<td>(1)</td>
</tr>
<tr>
<td>(\text{has}[sstail])</td>
<td>(\frac{1}{2})</td>
<td>1</td>
<td>1</td>
<td>(1)</td>
<td>1</td>
<td>1</td>
<td>(1)</td>
</tr>
<tr>
<td>(\text{same}[sshead, sstail])</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
</tr>
<tr>
<td>(\text{has}[ssnext])</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
</tr>
<tr>
<td>(\text{succ}[sstail, n])</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(\text{succ}[sshead, ssnext])</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
</tr>
<tr>
<td>(\text{succ}[sstail, ssnext])</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
</tr>
<tr>
<td>(\text{opval}[ssnext])</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
</tr>
<tr>
<td>(\text{ordered}[sshead, sstail])</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
<td>(\frac{1}{2})</td>
</tr>
</tbody>
</table>

\[\psi_1 = \text{has}[ssnext]\]
\[\psi_2 = \text{same}[sshead, sstail]\]

(b) Original Dequeue operation

Table 8.6: Invariant properties
### 8.3. QUEUE MODELS

#### Simplified Dequeue operation

<table>
<thead>
<tr>
<th></th>
<th>dequeue0</th>
<th>dequeue1</th>
<th>dequeue2</th>
<th>dequeue3</th>
<th>dequeue4</th>
<th>dequeue5</th>
<th>dequeue6</th>
<th>dequeue7</th>
<th>dequeue8</th>
<th>dequeue9</th>
<th>dequeue10</th>
</tr>
</thead>
<tbody>
<tr>
<td>doneLP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>doneELP</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>has[lv]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>has[n]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>opval[n]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>has[sshead]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>has[sstail]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>same[sshead, sstail]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>has[ssnext]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>succ[sstail, n]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>succ[sshead, ssnext]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>succ[sstail, ssnext]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>opval[ssnext]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \psi_1 = \text{has[ssnext]} \]
\[ \psi_2 = \text{same[sshead, sstail]} \]

(c) Simplified Dequeue operation

Table 8.6: Invariant properties
CHAPTER 8. COLLAPSING THREADS SAFELY

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 enq8 and enq9 there are conditional properties, as succ[sstail, ssnext] is true when has[ssnext] is true; the transitions from enq9 test whether ssnext is null, so succ[sstail, ssnext] is unconditionally true at enq11 and false at enq20.

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, deq70, due to the presence of a conditional property. The second is the disjunction in the properties of doneELP at deq76 and deq77.

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.

\[ \forall v \cdot \text{succ}[\text{sshead}, \text{ssnext}](v) \rightarrow \text{has}[\text{ssnext}](v) \]

There is no such relationship between doneELP and has[ssnext] however. At deq76, if has[ssnext] is false then the thread’s previous transition will instead have been specDequeueEmpty from deq71, which sets doneELP to true. If has[ssnext] is true, the thread’s previous transition will have been isNotNull from deq70 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 \texttt{deq34} and \texttt{deq42} is more subtle, relying on the structure of the list. When \texttt{Head} and \texttt{Tail} are equal, the list may be empty, in which case \texttt{Head} has no next-successor; alternatively the list may have size one and have \texttt{Tail} lagging, in which case \texttt{Head} has a next-successor. Whenever \texttt{Head} and \texttt{Tail} are not equal then the list is non-empty, and \texttt{Head} has a next-successor.

Thus, at \texttt{deq34} after \texttt{ssnext} has been assigned, if \texttt{same[sshead, stail]} is true then \texttt{has[ssnext]} could be either true or false, but if \texttt{same[sshead, stail]} is false then \texttt{has[ssnext]} has to be true. The latter property is essential to preserve because after \texttt{same[sshead, stail]} is checked to be false at \texttt{deq41}, \texttt{ssnext} is assumed to be non-null (e.g. at \texttt{deq48} and \texttt{deq50}) 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 \texttt{enq3–enq6, enq9, enq25, deq30, deq41–deq43, deq50–deq51, deq67, deq77, deq78, deq83, deq95, and deq98}, 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 \texttt{Head} and \texttt{Tail} 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
### Table 8.7: Queue verification results for restricted interleaving

<table>
<thead>
<tr>
<th>Deq</th>
<th>Th.</th>
<th>Heap Limit (MB)</th>
<th>Ave Time (s)</th>
<th>Max Time (s)</th>
<th>Ave RAM (MB)</th>
<th>Max RAM (MB)</th>
<th>Uns. Stored States</th>
<th>Stored States (MB)</th>
<th>Total States (MB)</th>
<th>Breaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>∞</td>
<td>800</td>
<td>81</td>
<td>151</td>
<td>309</td>
<td>12</td>
<td>523</td>
<td>2,994</td>
<td>89,869</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>∞</td>
<td>2,048</td>
<td>79</td>
<td>296</td>
<td>717</td>
<td>12</td>
<td>523</td>
<td>3,001</td>
<td>89,869</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>∞</td>
<td>800</td>
<td>216</td>
<td>181</td>
<td>310</td>
<td>14</td>
<td>591</td>
<td>3,254</td>
<td>248,223</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>∞</td>
<td>2,048</td>
<td>194</td>
<td>363</td>
<td>699</td>
<td>14</td>
<td>591</td>
<td>3,255</td>
<td>248,223</td>
<td></td>
</tr>
</tbody>
</table>

### Table 8.8: Queue verification results for full interleaving

<table>
<thead>
<tr>
<th>Deq</th>
<th>Th.</th>
<th>Heap Limit (MB)</th>
<th>Ave Time (s)</th>
<th>Max Time (s)</th>
<th>Ave RAM (MB)</th>
<th>Max RAM (MB)</th>
<th>Uns. Stored States</th>
<th>Stored States (MB)</th>
<th>Total States (MB)</th>
<th>Breaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>∞</td>
<td>800</td>
<td>3,109</td>
<td>3,838</td>
<td>304</td>
<td>201</td>
<td>727</td>
<td>235</td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>∞</td>
<td>2,048</td>
<td>2,810</td>
<td>3,557</td>
<td>581</td>
<td>196</td>
<td>1,691</td>
<td>236</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>∞</td>
<td>800</td>
<td>2,887</td>
<td>1,337</td>
<td>290</td>
<td>160</td>
<td>731</td>
<td>236</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>∞</td>
<td>2,048</td>
<td>2,658</td>
<td>1,370</td>
<td>515</td>
<td>142</td>
<td>1,537</td>
<td>220</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Resources

<table>
<thead>
<tr>
<th>Deq</th>
<th>Th.</th>
<th>Uns. Stored States (uns.)</th>
<th>Stored States (stored)</th>
<th>Total States (total)</th>
<th>Breaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>∞</td>
<td>96</td>
<td>4,148</td>
<td>24,378</td>
<td>2,375,953</td>
</tr>
<tr>
<td>S</td>
<td>∞</td>
<td>112</td>
<td>4,604</td>
<td>27,422</td>
<td>2,315,936</td>
</tr>
</tbody>
</table>

(b) Statespace

Table 8.7: Queue verification results for restricted interleaving

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 non-blocking 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 linearity, 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 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 $\triangleright$, 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:

$$\text{if}[^{push}_9, \text{has}[n](v) \land \text{at}[^{push}_9](v) \triangleright \text{has}[n](v)$$

is exactly the same, with renaming, for:

$$\text{if}[^{deq}_{31}, \text{has}[ss\text{head}](v) \land \text{at}[^{deq}_{31}](v) \triangleright \text{has}[ss\text{head}](v)$$

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:
APPENDIX A. PROOFS OF C. A. COMPATIBILITY CONSTRAINTS

{ Uniqueness of Head_i }

[A1] \( \forall v_1, v_2 \cdot \text{Head}_i(v_1) \land \text{Head}_i(v_2) \rightarrow v_1 = v_2 \)

{ Functionality of n }

[A2] \( \forall v_1, v_2, v_3 \cdot n(v_1, v_2) \land n(v_1, v_3) \rightarrow v_2 = v_3 \)

{ Functionality of ss }

[A3] \( \forall v_1, v_2, v_3 \cdot \text{ss}(v_1, v_2) \land \text{ss}(v_1, v_3) \rightarrow v_2 = v_3 \)

{ Functionality of sshead }

[A4] \( \forall v_1, v_2, v_3 \cdot \text{sshead}(v_1, v_2) \land \text{sshead}(v_1, v_3) \rightarrow v_2 = v_3 \)

{ Functionality of sshead }

[A5] \( \forall v_1, v_2, v_3 \cdot \text{sstail}(v_1, v_2) \land \text{sstail}(v_1, v_3) \rightarrow v_2 = v_3 \)

{ Functionality of next }

[A6] \( \forall v_1, v_2, v_3 \cdot \text{next}(v_1, v_2) \land \text{next}(v_1, v_3) \rightarrow v_2 = v_3 \)

{ Functionality of spec }

[A7] \( \forall v_1, v_2, v_3 \cdot \text{spec}(v_1, v_2) \land \text{spec}(v_1, v_3) \rightarrow v_2 = v_3 \)

{ Inverse Functionality of spec }

[A8] \( \forall v_1, v_2, v_3 \cdot \text{spec}(v_1, v_3) \land \text{spec}(v_2, v_3) \rightarrow v_1 = v_2 \)

{ Definition of \text{if} }[\text{push}_9] \}

[A9] \( \forall v_1 \cdot \text{if} [\text{push}_9, \text{has}[n]](v_1) \rightarrow \)
\( \quad \text{is_thread}(v_1) \land (\text{at}[\text{push}_9](v_1) \rightarrow \text{has}[n](v_1)) \)

{ Definition of if[push9, succ[n, ss]] }

[A10] \( \forall v_1 \cdot \text{if} [\text{push}_9, \text{succ}[n, ss]](v_1) \rightarrow \)
\( \quad \text{is_thread}(v_1) \land (\text{at}[\text{push}_9](v_1) \rightarrow (\text{has}[ss](v_1) \rightarrow \text{succ}[n, ss](v_1))) \)

{ Definition of tonull[ss] }

[A11] \( \forall v_1 \cdot \text{tonull}[ss](v_1) \rightarrow \exists v_2 \cdot \text{ss}(v_1, v_2) \land \neg \text{has}[next](v_2) \)

{ Definition of same[sshead, sstail] }
A.2 Soft Invariants

A.2.1 Single predicate

if\([\text{push}_9, \text{has}[n]](v) \land \text{at}[\text{push}_9](v) \supset \text{has}[n](v)\)

(1) \[\text{if}[\text{push}_9, \text{has}[n]](t_1) \land \text{at}[\text{push}_9](t_1)\]

(2) \[\text{if}[\text{push}_9, \text{has}[n]](t_1) \] 

\[\text{[1]}, \land \text{E}\]
APPENDIX A. PROOFS OF C. A. COMPATIBILITY CONSTRAINTS

(3) \[ \text{if} [\text{push}_9, \text{has}[n]](t_1) \rightarrow \text{is\_thread}(t_1) \quad [A9] \land \forall E \]
    \[ \land (\text{at}[\text{push}_9](t_1) \rightarrow \text{has}[n](t_1)) \]

(4) \[ \text{is\_thread}(t_1) \land (\text{at}[\text{push}_9](t_1) \rightarrow \text{has}[n](t_1)) \quad (2)(3) \rightarrow E \]

(5) \[ \text{at}[\text{push}_9](t_1) \rightarrow \text{has}[n](t_1) \quad (4) \land E \]

(6) \[ \text{at}[\text{push}_9](t_1) \quad (1) \land E \]

(7) \[ \text{has}[n](t_1) \quad (5)(6) \rightarrow E \]

(8) \[ \text{if} [\text{push}_9, \text{has}[n]](t_1) \land \text{at}[\text{push}_9](t_1) \rightarrow \text{has}[n](t_1) \quad (1)(7) \rightarrow I \]

A.2.2 Conditional property

\[
\text{if} [\text{push}_9, \text{succ}[n, ss]](v) \land \text{at}[\text{push}_9](v) \land \text{has}[ss](v) \Rightarrow \text{succ}[n, ss](v)
\]

(1) \[ \text{if} [\text{push}_9, \text{succ}[n, ss]](t_1) \land \text{at}[\text{push}_9](t_1) \land \text{has}[ss](t_1) \quad (1) \land E \]

(2) \[ \text{if} [\text{push}_9, \text{succ}[n, ss]](t_1) \quad (1) \land E \]

(3) \[ \text{if} [\text{push}_9, \text{succ}[n, ss]](t_1) \rightarrow \text{is\_thread}(t_1) \land \\
    (\text{at}[\text{push}_9](t_1) \rightarrow \text{has}[ss](t_1) \rightarrow \text{succ}[n, ss](t_1))) \quad [A10] \lor E \]

(4) \[ \text{is\_thread}(t_1) \land (\text{at}[\text{push}_9](t_1) \rightarrow \text{has}[ss](t_1) \rightarrow \text{succ}[n, ss](t_1))) \quad (2)(3) \rightarrow E \]

(5) \[ \text{at}[\text{push}_9](t_1) \rightarrow \text{has}[ss](t_1) \rightarrow \text{succ}[n, ss](t_1)) \quad (4) \land E \]

(6) \[ \text{at}[\text{push}_9](t_1) \quad (1) \land E \]

(7) \[ \text{has}[ss](t_1) \rightarrow \text{succ}[n, ss](t_1)) \quad (5)(6) \rightarrow E \]

(8) \[ \text{has}[ss](t_1) \quad (1) \land E \]

(9) \[ \text{succ}[n, ss](t_1) \quad (7)(8) \rightarrow E \]

(10) \[ \text{if} [\text{push}_9, \text{succ}[n, ss]](t_1) \land \text{at}[\text{push}_9](t_1) \land \text{has}[ss](t_1) \rightarrow \\
    \text{succ}[n, ss](t_1) \quad (1)(9) \rightarrow I \]
A.3  Linear

A.3.1  To null

$$\exists t \cdot \text{tonull}[\text{ss}](t) \land \text{ss}(t, n) \triangleright \neg \text{has}[\text{next}](n)$$

1. $$\exists v_1 \cdot \text{tonull}[\text{ss}](v_1) \land \text{ss}(v_1, n_1)$$
2. $$\text{tonull}[\text{ss}](t_1) \land \text{ss}(t_1, n_1)$$
3. $$\text{tonull}[\text{ss}](t_1)$$  \hspace{1cm} (2) \land E
4. $$\text{tonull}[\text{ss}](t_1) \rightarrow \exists v_2 \cdot \text{ss}(t_1, v_2) \land \neg \text{has}[\text{next}](v_2)$$  \hspace{1cm} [A11] \forall E
5. $$\exists v_2 \cdot \text{ss}(t_1, v_2) \land \neg \text{has}[\text{next}](v_2)$$  \hspace{1cm} (3), (4) \rightarrow E
6. $$\text{ss}(t_1, n_2) \land \neg \text{has}[\text{next}](n_2)$$

7. $$\text{ss}(t_1, n_1) \land \text{ss}(t_1, n_2) \rightarrow n_1 = n_2$$  \hspace{1cm} [A3] \forall E
8. $$\text{ss}(t_1, n_1)$$  \hspace{1cm} (2) \land E
9. $$\text{ss}(t_1, n_2)$$  \hspace{1cm} (6) \land E
10. $$\text{ss}(t_1, n_1) \land \text{ss}(t_1, n_2)$$  \hspace{1cm} (8), (9) \land I
11. $$n_1 = n_2$$  \hspace{1cm} (7), (10) \rightarrow E
12. $$\neg \text{has}[\text{next}](n_2)$$  \hspace{1cm} (6) \land E
13. $$\neg \text{has}[\text{next}](n_1)$$  \hspace{1cm} (12) \Rightarrow E
14. $$\neg \text{has}[\text{next}](n_1)$$  \hspace{1cm} (5), (6), (13) \exists E
15. $$\neg \text{has}[\text{next}](n_1)$$  \hspace{1cm} (1), (2), (14) \exists E
16. $$(\exists v_1 \cdot \text{tonull}[\text{ss}](v_1) \land \text{ss}(v_1, n_1)) \rightarrow \neg \text{has}[\text{next}](n_1)$$  \hspace{1cm} (1), (15) \rightarrow I

A.3.2  Same 1

same[sshead, ssTail](t) \land \text{sshead}(t, n) \triangleright \text{ssTail}(t, n)

1. $$\text{same}[\text{sshead, ssTail}](t_1) \land \text{sshead}(t_1, n_1)$$
2. $$\text{same}[\text{sshead, ssTail}](t_1)$$  \hspace{1cm} (1) \land E
APPENDIX A. PROOFS OF C. A. COMPATIBILITY CONSTRAINTS

(3) \(\text{same}[\text{sshead, stail}](t_1) \rightarrow \exists v_2 \cdot \text{sshead}(t_1, v_2) \land \text{stail}(t_1, v_2)\) \[A12\] \forall E

(4) \(\exists v_2 \cdot \text{sshead}(t_1, v_2) \land \text{stail}(t_1, v_2)\) \[2, 3\] \rightarrow E

(5) \(\text{sshead}(t_1, n_2) \land \text{stail}(t_1, n_2)\)

(6) \(\text{sshead}(t_1, n_1) \land \text{sshead}(t_1, n_2) \rightarrow n_1 = n_2\) \[A4\] \forall E

(7) \(\text{sshead}(t_1, n_1)\) \[1\] \land E

(8) \(\text{sshead}(t_1, n_2)\) \[5\] \land E

(9) \(\text{sshead}(t_1, n_1) \land \text{sshead}(t_1, n_2)\) \[7, 8\] \land I

(10) \(n_1 = n_2\) \[6, 9\] \rightarrow E

(11) \(\text{stail}(t_1, n_2)\) \[5\] \land E

(12) \(\text{stail}(t_1, n_1)\) \[10, 11\] \rightarrow E

(13) \(\text{stail}(t_1, n_1)\) \[4, 5, 12\] \exists E

(14) \(\text{same}[\text{sshead, stail}](t_1) \land \text{sshead}(t_1, n_1) \rightarrow \text{stail}(t_1, n_1)\) \[1, 13\] \rightarrow I

A.3.3 Same 2
same[sshead, stail](t) \land stail(t, n) \triangleright sshead(t, n)

(1) \(\text{same}[\text{sshead, stail}](t_1) \land \text{stail}(t_1, n_1)\)

(2) \(\text{same}[\text{sshead, stail}](t_1)\) \[1\] \land E

(3) \(\text{same}[\text{sshead, stail}](t_1) \rightarrow \exists v_2 \cdot \text{sshead}(t_1, v_2) \land \text{stail}(t_1, v_2)\) \[A12\] \forall E

(4) \(\exists v_2 \cdot \text{sshead}(t_1, v_2) \land \text{stail}(t_1, v_2)\) \[2, 3\] \rightarrow E

(5) \(\text{sshead}(t_1, n_2) \land \text{stail}(t_1, n_2)\)

(6) \(\text{stail}(t_1, n_1) \land \text{stail}(t_1, n_2) \rightarrow n_1 = n_2\) \[A5\] \forall E
A.3. LINEAR

(7) \text{sstail}(t_1, n_1)

(8) \text{sstail}(t_1, n_2)

(9) \text{sstail}(t_1, n_1) \land \text{sstail}(t_1, n_2)

(10) n_1 = n_2

(11) \text{sshead}(t_1, n_2)

(12) \text{sshead}(t_1, n_1)

(13) \text{sshead}(t_1, n_1)

(14) \text{same}[\text{sshead, sstail}](t_1) \land \text{sstail}(t_1, n_1)

\rightarrow \text{sshead}(t_1, n_1)

A.3.4 Ordered

\exists t_1, n_1 \cdot \text{ordered}[\text{sshead, sstail}](t_1) \land \text{sshead}(t_1, n_1)

\land \text{sstail}(t_1, n_2) \land \text{reach}_1(n_1) \triangleright \text{reach}_1(n_2)

(1) \exists v_1, v_2 \cdot \text{ordered}[\text{sshead, sstail}](v_1) \land \text{sshead}(v_1, v_2)

\land \text{sstail}(v_1, n_2) \land \text{reach}_1(v_2)

(2) \text{ordered}[\text{sshead, sstail}](t_1) \land \text{sshead}(t_1, n_1)

\land \text{sstail}(t_1, n_2) \land \text{reach}_1(n_1)

(3) \text{ordered}[\text{sshead, sstail}](t_1)

(4) \text{ordered}[\text{sshead, sstail}](t_1) \rightarrow \exists v_2, v_3 \cdot \text{sshead}(t_1, v_2)

\land \text{sstail}(t_1, v_3) \land (\text{reach}_1(v_2) \rightarrow \text{reach}_1(v_3))

(5) \exists v_2, v_3 \cdot \text{sshead}(t_1, v_2) \land \text{sstail}(t_1, v_3)

\land (\text{reach}_1(v_2) \rightarrow \text{reach}_1(v_3))

(6) \text{sshead}(t_1, n_3) \land \text{sstail}(t_1, n_4)

\land (\text{reach}_1(n_3) \rightarrow \text{reach}_1(n_4))
APPENDIX A. PROOFS OF C. A. COMPATIBILITY CONSTRAINTS

| (7) | \(\text{sshead}(t_1, n_1) \land \text{sshead}(t_1, n_3) \rightarrow n_1 = n_3\) [A4], \(\forall \text{E}\) |
| (8) | \(\text{sshead}(t_1, n_1)\) (2) \(\land\) E |
| (9) | \(\text{sshead}(t_1, n_3)\) (6) \(\land\) E |
| (10) | \(\text{sshead}(t_1, n_1) \land \text{sshead}(t_1, n_3)\) (8) \(\land\) I |
| (11) | \(n_1 = n_3\) (7) \(\land\) I |
| (12) | \(\text{sstail}(t_1, n_2) \land \text{sstail}(t_1, n_4) \rightarrow n_2 = n_4\) [A5], \(\forall \text{E}\) |
| (13) | \(\text{sstail}(t_1, n_2)\) (2) \(\land\) E |
| (14) | \(\text{sstail}(t_1, n_4)\) (6) \(\land\) E |
| (15) | \(\text{sstail}(t_1, n_2) \land \text{sstail}(t_1, n_4)\) (13) \(\land\) I |
| (16) | \(n_2 = n_4\) (12) \(\land\) I |
| (17) | \(\text{reach}_I(n_3) \rightarrow \text{reach}_I(n_4)\) (6) \(\land\) E |
| (18) | \(\text{reach}_I(n_1)\) (2) \(\land\) E |
| (19) | \(\text{reach}_I(n_3)\) (11) \(\land\) I |
| (20) | \(\text{reach}_I(n_4)\) (17) \(\land\) I |
| (21) | \(\text{reach}_I(n_2)\) (16) \(\land\) I |
| (22) | \(\text{reach}_I(n_2)\) (5) \(\land\) I |
| (23) | \(\text{reach}_I(n_2)\) (6) \(\land\) I |
| (24) | \(\exists v_1, v_2 \cdot \text{ordered}[\text{sshead}, \text{sstail}](v_1) \land \text{sshead}(v_1, v_2) \land \text{sstail}(v_1, n_2) \land \text{reach}_I(v_2)) \rightarrow \text{reach}_I(n_2)\) (1) \(\exists\) E |

A.4 Geometric

A.4.1 Triangle 1

\(\exists t_1 \cdot \text{succ}[n, ss](t_1) \land n(t_1, n_1) \land \text{ss}(t_1, n_2) \triangleright \text{next}(n_1, n_2)\)

\(\exists v_1 \cdot \text{succ}[n, ss](v_1) \land n(v_1, n_1) \land \text{ss}(v_1, n_2)\) (1)
A.4. GEOMETRIC

(2) \text{succ}[n, ss](t_1) \land n(t_1, n_1) \land ss(t_1, n_2)

(3) \text{succ}[n, ss](t_1)

(4) \text{succ}[n, ss](t_1) \rightarrow \exists v_2, v_3 \cdot n(t_1, v_2) \land \text{next}(v_2, v_3) \land ss(t_1, v_3)

(5) \exists v_2, v_3 \cdot n(t_1, v_2) \land \text{next}(v_2, v_3) \land ss(t_1, v_3)

(6) n(t_1, n_3) \land \text{next}(n_3, n_4) \land ss(t_1, n_4)

(7) n(t_1, n_1) \land n(t_1, n_3) \rightarrow n_1 = n_3

(8) n(t_1, n_1)

(9) n(t_1, n_3)

(10) n(t_1, n_1) \land n(t_1, n_3)

(11) n_1 = n_3

(12) ss(t_1, n_2) \land ss(t_1, n_4) \rightarrow n_2 = n_4

(13) ss(t_1, n_2)

(14) ss(t_1, n_4)

(15) ss(t_1, n_2) \land ss(t_1, n_4)

(16) n_2 = n_4

(17) \text{next}(n_3, n_4)

(18) \text{next}(n_1, n_2)

(19) \text{next}(n_1, n_2)

(20) \text{next}(n_1, n_2)

(21) (\exists v_1 \cdot \text{succ}[n, ss](v_1) \land n(v_1, n_1) \land ss(v_1, n_2)) \rightarrow \text{next}(n_1, n_2)

A.4.2 Triangle 2

\exists n_1 \cdot \text{succ}[n, ss](t_1) \land n(t_1, n_1) \land \text{next}(n_1, n_2) \triangleright ss(t_1, n_2)
APPENDIX A. PROOFS OF C. A. COMPATIBILITY CONSTRAINTS

(1) \( \exists v_1 \cdot \text{succ}[n, ss](t_1) \land n(t_1, v_1) \land \text{next}(v_1, n_2) \)

(2) \( \text{succ}[n, ss](t_1) \land n(t_1, n_1) \land \text{next}(n_1, n_2) \)

(3) \( \text{succ}[n, ss](t_1) \quad [2] \land \text{E} \)

(4) \( \text{succ}[n, ss](t_1) \rightarrow \exists v_2, v_3 \cdot n(t_1, v_2) \quad [A13], \forall \text{E} \)

\( \land \text{next}(v_2, v_3) \land \text{ss}(t_1, v_3) \)

(5) \( \exists v_2, v_3 \cdot n(t_1, v_2) \land \text{next}(v_2, v_3) \land \text{ss}(t_1, v_3) \quad [3] [4] \rightarrow \text{E} \)

(6) \( n(t_1, n_3) \land \text{next}(n_3, n_4) \land \text{ss}(t_1, n_4) \)

(7) \( n(t_1, n_1) \land n(t_1, n_3) \rightarrow n_1 = n_3 \quad [A2], \forall \text{E} \)

(8) \( n(t_1, n_1) \quad [2] \land \text{E} \)

(9) \( n(t_1, n_3) \quad [6] \land \text{E} \)

(10) \( n(t_1, n_1) \land n(t_1, n_3) \quad [8] [9] \land \text{I} \)

(11) \( n_1 = n_3 \quad [7] [10] \rightarrow \text{E} \)

(12) \( \text{next}(n_1, n_2) \land \text{next}(n_1, n_4) \rightarrow n_2 = n_4 \quad [A6], \forall \text{E} \)

(13) \( \text{next}(n_1, n_2) \quad [2] \land \text{E} \)

(14) \( \text{next}(n_3, n_4) \quad [6] \land \text{E} \)

(15) \( \text{next}(n_1, n_4) \quad (11), (14) \rightarrow \text{E} \)

(16) \( \text{next}(n_1, n_2) \land \text{next}(n_1, n_4) \quad (13), (14) \land \text{I} \)

(17) \( n_2 = n_4 \quad (12), (16) \rightarrow \text{E} \)

(18) \( \text{ss}(t_1, n_4) \quad [6] \land \text{E} \)

(19) \( \text{ss}(t_1, n_2) \quad (17), (18) \rightarrow \text{E} \)

(20) \( \text{ss}(t_1, n_2) \quad [5] [6] [19] \exists \text{E} \)

(21) \( \text{ss}(t_1, n_2) \quad (1) [21] \exists \text{E} \)

(22) \( \exists v_1 \cdot \text{succ}[n, ss](t_1) \land n(t_1, v_1) \land \text{next}(v_1, n_2) \quad (1) [21] \rightarrow \text{I} \)

\( \rightarrow \text{ss}(t_1, n_2) \)
A.4.3 Square 1

\[ \exists n_1, n_2 \cdot \text{commutes}(n_1) \land \]
\[ \text{next}(n_1, n_2) \land \text{spec}(n_1, n_3) \land \text{spec}(n_2, n_4) \Rightarrow \text{next}(n_3, n_4) \]

(1) \[ \exists v_1, v_2 \cdot \text{commutes}(v_1) \land \text{next}(v_1, v_2) \]
\[ \land \text{spec}(v_1, n_3) \land \text{spec}(v_2, n_4) \]

(2) \[ \text{commutes}(n_1) \land \text{next}(n_1, n_2) \land \text{spec}(n_1, n_3) \]
\[ \land \text{spec}(n_2, n_4) \]

(3) \[ \text{commutes}(n_1) \]

(4) \[ \text{commutes}(n_1) \rightarrow \exists v_2, v_3, v_4 \cdot \text{next}(n_1, v_2) \]
\[ \land \text{spec}(n_1, v_3) \land \text{next}(v_3, v_4) \land \text{spec}(v_2, v_4) \]

(5) \[ \exists v_2, v_3, v_4 \cdot \text{next}(n_1, v_2) \land \text{spec}(n_1, v_3) \]
\[ \land \text{next}(v_3, v_4) \land \text{spec}(v_2, v_4) \]

(6) \[ \text{next}(n_1, n_5) \land \text{spec}(n_1, n_6) \land \text{next}(n_6, n_7) \]
\[ \land \text{spec}(n_5, n_7) \]

(7) \[ \text{next}(n_1, n_2) \land \text{next}(n_1, n_5) \rightarrow n_2 = n_5 \]

(8) \[ \text{next}(n_1, n_2) \]

(9) \[ \text{next}(n_1, n_5) \]

(10) \[ \text{next}(n_1, n_2) \land \text{next}(n_1, n_5) \]

(11) \[ n_2 = n_5 \]

(12) \[ \text{spec}(n_1, n_3) \land \text{spec}(n_1, n_6) \rightarrow n_3 = n_6 \]

(13) \[ \text{spec}(n_1, n_3) \]

(14) \[ \text{spec}(n_1, n_6) \]

(15) \[ \text{spec}(n_1, n_3) \land \text{spec}(n_1, n_6) \]

(16) \[ n_3 = n_6 \]
APPENDIX A. PROOFS OF C. A. COMPATIBILITY CONSTRAINTS

(17) \[ \text{spec}(n_2, n_4) \land \text{spec}(n_2, n_7) \rightarrow n_4 = n_7 \quad \text{[A7]} \quad \forall E \]

(18) \[ \text{spec}(n_2, n_4) \quad \text{(2)} \land E \]

(19) \[ \text{spec}(n_5, n_7) \quad \text{(6)} \land E \]

(20) \[ \text{spec}(n_2, n_7) \quad \text{(11)} \land \text{[19]} = E \]

(21) \[ \text{spec}(n_2, n_4) \land \text{spec}(n_2, n_7) \quad \text{(18)} \land \text{(20)} \land I \]

(22) \[ n_4 = n_7 \quad \text{(17)} \land \text{(21)} \rightarrow E \]

(23) \[ \text{next}(n_6, n_7) \quad \text{(6)} \land E \]

(24) \[ \text{next}(n_3, n_4) \quad \text{(16)} \land \text{(22)} \land \text{(23)} = E \]

(25) \[ \text{next}(n_3, n_4) \quad \text{(5)} \land \text{(6)} \land \text{(24)} \exists E \]

(26) \[ \text{next}(n_3, n_4) \quad \text{(1)} \land \text{(2)} \land \text{(25)} \exists E \]

(27) \[ (\exists v_1, v_2 \cdot \text{commutes}(v_1) \land \text{next}(v_1, v_2) \land \text{spec}(v_1, n_3) \land \text{spec}(v_2, n_4)) \rightarrow \text{next}(n_3, n_4) \]

A.4.4 Square 2

\[ \exists n_1, n_3 \cdot \text{commutes}(n_1) \land \text{next}(n_1, n_2) \land \text{spec}(n_1, n_3) \land \text{next}(n_3, n_4) \supset \text{spec}(n_2, n_4) \]

(1) \[ \exists v_1, v_2 \cdot \text{commutes}(v_1) \land \text{next}(v_1, n_2) \land \text{spec}(v_1, v_2) \land \text{next}(v_2, n_4) \]

(2) \[ \text{commutes}(n_1) \land \text{next}(n_1, n_2) \land \text{spec}(n_1, n_3) \land \text{next}(n_3, n_4) \]

(3) \[ \text{commutes}(n_1) \quad \text{(2)} \land E \]

(4) \[ \text{commutes}(n_1) \rightarrow \exists v_2, v_3, v_4 \cdot \text{next}(n_1, v_2) \land \text{spec}(n_1, v_3) \land \text{next}(v_3, v_4) \land \text{spec}(v_2, v_4) \quad \text{[A14]} \land E \]
A.4. GEOMETRIC

(5) \[ \exists v_2, v_3, v_4 \cdot \text{next}(n_1, v_2) \land \text{spec}(n_1, v_3) \land \text{next}(v_3, v_4) \land \text{spec}(v_2, v_4) \]

\[ \text{next}(n_1, n_5) \land \text{spec}(n_1, n_6) \land \text{next}(n_6, n_7) \land \text{spec}(n_5, n_7) \]

(7) \[ \text{next}(n_1, n_2) \land \text{next}(n_1, n_5) \rightarrow n_2 = n_5 \]

(8) \[ \text{next}(n_1, n_2) \]

(9) \[ \text{next}(n_1, n_5) \]

(10) \[ \text{next}(n_1, n_2) \land \text{next}(n_1, n_5) \]

(11) \[ n_2 = n_5 \]

(12) \[ \text{spec}(n_1, n_3) \land \text{spec}(n_1, n_6) \rightarrow n_3 = n_6 \]

(13) \[ \text{spec}(n_1, n_3) \]

(14) \[ \text{spec}(n_1, n_6) \]

(15) \[ \text{spec}(n_1, n_3) \land \text{spec}(n_1, n_6) \]

(16) \[ n_3 = n_6 \]

(17) \[ \text{next}(n_3, n_4) \land \text{next}(n_3, n_7) \rightarrow n_4 = n_7 \]

(18) \[ \text{next}(n_3, n_4) \]

(19) \[ \text{next}(n_6, n_7) \]

(20) \[ \text{next}(n_5, n_7) \]

(21) \[ \text{next}(n_5, n_4) \land \text{next}(n_5, n_7) \]

(22) \[ n_4 = n_7 \]

(23) \[ \text{spec}(n_5, n_7) \]

(24) \[ \text{spec}(n_2, n_4) \]

(25) \[ \text{spec}(n_2, n_4) \]

(26) \[ \text{spec}(n_2, n_4) \]
(27) \((\exists v_1, v_2 \cdot \text{commutes}(v_1) \land \text{next}(v_1, n_2) \land \text{spec}(v_1, v_2) \land \text{next}(v_2, n_4)) \rightarrow \text{spec}(n_2, n_4)\) 

\[1\), \[26\] \rightarrow I

A.4.5 Square 3

\(\exists n_3, n_4 \cdot \text{commutes}(n_1) \land \text{spec}(n_1, n_3) \land \text{next}(n_3, n_4) \land \text{spec}(n_2, n_4) \land \text{next}(n_1, n_2)\)

(1) \(\exists v_1, v_2 \cdot \text{commutes}(n_1) \land \text{spec}(n_1, v_1) \land \text{next}(v_1, v_2) \land \text{spec}(n_2, v_2)\)

(2) \(\text{commutes}(n_1) \land \text{spec}(n_1, n_3) \land \text{next}(n_3, n_4) \land \text{spec}(n_2, n_4)\)

(3) \(\text{commutes}(n_1)\) \hspace{1cm} \[2\], \& E

(4) \(\text{commutes}(n_1) \rightarrow \exists v_2, v_3, v_4 \cdot \text{next}(v_1, v_2) \land \text{spec}(n_1, v_3) \land \text{next}(v_3, v_4) \land \text{spec}(v_2, v_4)\) \hspace{1cm} \[A14\], \& E

(5) \(\exists v_2, v_3, v_4 \cdot \text{next}(v_1, v_2) \land \text{spec}(n_1, v_3) \land \text{next}(v_3, v_4) \land \text{spec}(v_2, v_4)\)

(6) \(\text{next}(n_1, n_5) \land \text{spec}(n_1, n_6) \land \text{next}(n_6, n_7) \land \text{spec}(n_5, n_7)\)

(7) \(\text{spec}(n_1, n_3) \land \text{spec}(n_1, n_6) \rightarrow n_3 = n_6\) \hspace{1cm} \[A7\], \& E

(8) \(\text{spec}(n_1, n_3)\) \hspace{1cm} \[2\], \& E

(9) \(\text{spec}(n_1, n_6)\) \hspace{1cm} \[6\], \& E

(10) \(\text{spec}(n_1, n_3) \land \text{spec}(n_1, n_6)\) \hspace{1cm} \[8\), \[9\] \& I

(11) \(n_3 = n_6\) \hspace{1cm} \[7\), \[10\] \rightarrow E

(12) \(\text{next}(n_3, n_4) \land \text{next}(n_3, n_7) \rightarrow n_4 = n_7\) \hspace{1cm} \[A6\], \& E

(13) \(\text{next}(n_3, n_4)\) \hspace{1cm} \[2\], \& E
(14) \( \text{next}(n_6, n_7) \)

(15) \( \text{next}(n_5, n_7) \)

(16) \( \text{next}(n_5, n_4) \land \text{next}(n_3, n_7) \)

(17) \( n_4 = n_7 \)

(18) \( \text{spec}(n_2, n_4) \land \text{spec}(n_5, n_4) \to n_2 = n_5 \) [A8], \( \forall \text{E} \)

(19) \( \text{spec}(n_2, n_4) \) [2], \( \land \text{E} \)

(20) \( \text{spec}(n_5, n_7) \) [6], \( \land \text{E} \)

(21) \( \text{spec}(n_5, n_4) \) [17], [20], \( \text{E} \)

(22) \( \text{spec}(n_2, n_4) \land \text{spec}(n_5, n_4) \) [19], [21], \( \land \text{I} \)

(23) \( n_2 = n_5 \) [18], [22], \( \land \text{E} \)

(24) \( \text{next}(n_1, n_5) \) [6], \( \land \text{E} \)

(25) \( \text{next}(n_1, n_2) \) [23], [24], \( \text{E} \)

(26) \( \text{next}(n_1, n_2) \) [5], [6], [25], \( \exists \text{E} \)

(27) \( \text{next}(n_1, n_2) \) [1], [2], [26], \( \exists \text{E} \)

(28) \( \exists v_1, v_2 \bullet \text{commutes}(n_1) \land \text{spec}(n_1, v_1) \land \text{next}(v_1, v_2) \land \text{spec}(n_2, v_2) \to \text{next}(n_1, n_2) \) [1], [27], \( \to \text{I} \)

### 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

\( \neg \text{circ}(n_1) \land n_1 = n_2 \not\Rightarrow \neg \text{next}(n_1, n_2) \)
APPENDIX A. PROOFS OF C. A. COMPATIBILITY CONSTRAINTS

(1) \( \neg \text{circ}(n_1) \land n_1 = n_2 \)

(2) \( n_1 = n_2 \) \hspace{1cm} (1) \& E

(3) \( \text{next}(n_1, n_2) \)

(4) \( \text{next}(n_1, n_1) \) \hspace{1cm} (2), (3) = E

(5) \( \text{next}^+(n_1, n_1) \)

(6) \( \text{next}^+(n_1, n_1) \rightarrow \text{circ}(n_1) \) \hspace{1cm} [A17] \& E

(7) \( \text{circ}(n_1) \)

(8) \( \neg \text{circ}(n_1) \)

(9) \( \bot \) \hspace{1cm} (7), (8) \bot I

(10) \( \neg \text{next}(n_1, n_2) \)

(11) \( \neg \text{circ}(n_1) \land n_1 = n_2 \rightarrow \neg \text{next}(n_1, n_2) \) \hspace{1cm} (1), (10) \rightarrow I

A.5.2 No loop back

\( \neg \text{circ}(n_1) \land \text{next}(n_1, n_2) \supset \neg \text{next}(n_2, n_1) \)

(1) \( \neg \text{circ}(n_1) \land \text{next}(n_1, n_2) \)

(2) \( \text{next}(n_1, n_2) \) \hspace{1cm} (1) \& E

(3) \( \text{next}(n_2, n_1) \)

(4) \( \text{next}^+(n_1) \) \hspace{1cm} (2), (3) TC

(5) \( \text{next}^+(n_1) \rightarrow \text{circ}(n_1) \) \hspace{1cm} [A17] \& E

(6) \( \text{circ}(n_1) \)

(7) \( \neg \text{circ}(n_1) \)

(8) \( \bot \) \hspace{1cm} (6), (7) \bot I

(9) \( \neg \text{next}(n_2, n_1) \)

(10) \( \neg \text{circ}(n_1) \land \text{next}(n_1, n_2) \rightarrow \neg \text{next}(n_2, n_1) \) \hspace{1cm} (1), (9) \rightarrow I
A.5. REACHABILITY

A.5.3 No loop to head

\[ \text{Head}_i(n_1) \land \neg \text{circ}(n_1) \land \text{reach}_i(n_2) \implies \neg \text{next}(n_2, n_1) \]

1. \[ \text{Head}_i(n_1) \land \neg \text{circ}(n_1) \land \text{reach}_i(n_2) \]
2. \[ \text{reach}_i(n_2) \]
3. \[ \text{reach}_i(n_2) \implies \exists v_2 : \text{Head}_i(v_2) \land \text{next}^*(v_2, n_2) \]
4. \[ \exists v_2 : \text{Head}_i(v_2) \land \text{next}^*(v_2, n_2) \]
5. \[ \text{Head}_i(n_3) \land \text{next}^*(n_3, n_2) \]
6. \[ \text{Head}_i(n_1) \land \text{Head}_i(n_3) \implies n_1 = n_3 \]
7. \[ \text{Head}_i(n_1) \]
8. \[ \text{Head}_i(n_3) \]
9. \[ \text{Head}_i(n_1) \land \text{Head}_i(n_3) \]
10. \[ n_1 = n_3 \]
11. \[ \text{next}^*(n_3, n_2) \]
12. \[ \text{next}^*(n_1, n_2) \]
13. \[ \text{next}(n_2, n_1) \]
14. \[ \text{next}^+(n_1, n_1) \]
15. \[ \text{next}^+(n_1, n_1) \implies \text{circ}(n_1) \]
16. \[ \text{circ}(n_1) \]
17. \[ \neg \text{circ}(n_1) \]
18. \[ \bot \]
19. \[ \neg \text{next}(n_2, n_1) \]
20. \[ \neg \text{next}(n_2, n_1) \]
21. \[ \text{Head}_i(n_1) \land \neg \text{circ}(n_1) \land \text{reach}_i(n_2) \implies \neg \text{next}(n_2, n_1) \]
A.5.4 Unreachable

\[ \text{reach}_I(n_1) \land \neg \text{reach}_I(n_2) \Rightarrow \neg \text{next}(n_1, n_2) \]

(1) \[ \text{reach}_I(n_1) \land \neg \text{reach}_I(n_2) \]
(2) \[ \text{reach}_I(n_1) \] [1] \( \land \) E
(3) \[ \text{reach}_I(n_1) \rightarrow \exists v_2 \cdot \text{Head}_I(v_2) \land \text{next}^*(v_2, n_1) \] [A18] \( \forall \) E
(4) \[ \exists v_2 \cdot \text{Head}_I(v_2) \land \text{next}^*(v_2, n_1) \] [2], [3] \( \Rightarrow \) E
(5) \[ \text{next}(n_1, n_2) \]
(6) \[ \exists v_2 \cdot \text{Head}_I(v_2) \land \text{next}^*(v_2, n_2) \] [4], [5] TC
(7) \[ (\exists v_2 \cdot \text{Head}_I(v_2) \land \text{next}^*(v_2, n_2)) \rightarrow \text{reach}_I(n_2) \] [A19] \( \forall \) E
(8) \[ \text{reach}_I(n_2) \]
(9) \[ \neg \text{reach}_I(n_2) \] [1] \( \land \) E
(10) \[ \bot \] [8], [9] \( \bot \) I
(11) \[ \neg \text{next}(n_1, n_2) \] [5], [10] \( \neg \) I
(12) \[ \text{reach}_I(n_1) \land \neg \text{reach}_I(n_2) \rightarrow \neg \text{next}(n_1, n_2) \] [1], [11] \( \rightarrow \) I

A.5.5 Chain

\[ \exists n_1 \cdot \text{reach}_I(n_1) \land \text{reach}_I(n_2) \land \neg \text{circ}(n_2) \land \neg \text{eq}(n_1, n_3) \Rightarrow \neg \text{next}(n_3, n_2) \]

(1) \[ \exists v_1 \cdot \text{reach}_I(v_1) \land \text{reach}_I(n_2) \land \neg \text{circ}(n_2) \land \neg \text{eq}(n_1, n_3) \land \text{next}(v_1, n_2) \land \text{reach}_I(n_3) \land v_1 \neq n_3 \]
(2) \[ \text{reach}_I(n_1) \land \text{reach}_I(n_2) \land \neg \text{circ}(n_2) \land \neg \text{eq}(n_1, n_3) \land \text{next}(n_1, n_2) \land \text{reach}_I(n_3) \land n_1 \neq n_3 \]
(3) \[ \text{reach}_I(n_1) \] [2] \( \land \) E
(4) \[ \text{reach}_I(n_2) \] [2] \( \land \) E
A.5. REACHABILITY

(5) \[ \text{reach}(n_3) \] \hspace{2cm} (2) \land E

(6) \[ \text{reach}(n_1) \rightarrow \exists v_2 \cdot \text{Head}_1(v_2) \land \text{next}^*(v_2, n_1) \] \hspace{2cm} [A18] \land E

(7) \[ \text{reach}(n_2) \rightarrow \exists v_2 \cdot \text{Head}_1(v_2) \land \text{next}^*(v_2, n_2) \] \hspace{2cm} [A18] \land E

(8) \[ \text{reach}(n_3) \rightarrow \exists v_2 \cdot \text{Head}_1(v_2) \land \text{next}^*(v_2, n_3) \] \hspace{2cm} [A18] \land E

(9) \[ \exists v_2 \cdot \text{Head}_1(v_2) \land \text{next}^*(v_2, n_1) \] \hspace{2cm} (3), (6) \rightarrow E

(10) \[ \exists v_2 \cdot \text{Head}_1(v_2) \land \text{next}^*(v_2, n_2) \] \hspace{2cm} (4), (7) \rightarrow E

(11) \[ \exists v_2 \cdot \text{Head}_1(v_2) \land \text{next}^*(v_2, n_3) \] \hspace{2cm} (5), (8) \rightarrow E

(12) \[ \text{next}(n_1, n_2) \] \hspace{2cm} (2) \land E

(13) \[ n_1 \neq n_3 \] \hspace{2cm} (2) \land E

\[ \ldots \]

(14) \[ \text{next}^+(n_3, n_1) \lor \text{next}^*(n_2, n_3) \] \hspace{2cm} [A1], [A6], (9)–(13), TC

(15) \[ \text{next}(n_3, n_2) \]

(16) \[ \neg \text{circ}(n_2) \] \hspace{2cm} (2) \land E

(17) \[ \text{next}^+(n_2, n_2) \rightarrow \text{circ}(n_2) \] \hspace{2cm} [A17] \land E

(18) \[ \text{next}^+(n_3, n_1) \]

\[ \ldots \]

(19) \[ \text{next}^*(n_2, n_1) \] \hspace{2cm} (15), (18), [A6], TC

(20) \[ \text{next}^+(n_2, n_2) \] \hspace{2cm} (12), (19), TC

(21) \[ \text{circ}(n_2) \] \hspace{2cm} (17), (20) \rightarrow E

(22) \[ \bot \] \hspace{2cm} (16), (21), \bot I

(23) \[ \text{next}^*(n_2, n_3) \]

(24) \[ \text{next}^+(n_2, n_2) \] \hspace{2cm} (15), (23), TC

(25) \[ \text{circ}(n_2) \] \hspace{2cm} (17), (24) \rightarrow E

(26) \[ \bot \] \hspace{2cm} (16), (25), \bot I
APPENDIX A. PROOFS OF C. A. COMPATIBILITY CONSTRAINTS

(27) \[ \bot \]

(28) \[ \neg \text{next}(n_3, n_2) \]

(29) \[ \neg \text{next}(n_3, n_2) \]

(30) \[ \exists v_1 \cdot \text{reach}_1(v_1) \land \text{reach}_1(n_2) \land \neg \text{circ}(n_2) \land \]
\[ \text{next}(v_1, n_2) \land \text{reach}_1(n_3) \land v_1 \neq n_3 \]
\[ \rightarrow \neg \text{next}(n_3, n_2) \]
Bibliography


Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In John White, Richard Lipton, and Patricia C.


Igor Bogudlov, Tal Lev-Ami, Thomas Reps, and Mooly Sagiv. Revamping TVLA: Making parametric shape analysis competitive. In Werner


Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen


Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a lazy concurrent list-based set algorithm. In Thomas Ball


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


Lindsay Groves and Robert Colvin. Derivation of a scalable lock-free stack algorithm. In Bernhard K. Aichernig, Eerke A. Boiten, John Derrick, and


Leslie Lamport. “Sometime” is sometimes “Not Never”. In Paul Abrahams, Richard Lipton, and Stephen Bourne, editors, *Proceedings of the 7th


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


BIBLIOGRAPHY


Graeme Smith and John Derrick. Model checking downwards simulations. In John Derrick and Eerke A. Boiten, editors, *Proceedings of the
BIBLIOGRAPHY


Moshe Y. Vardi. From Church and Prior to PSL. In Grumberg and Veith [2008], pages 150–171. (p. 29).


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).


