Concrete and Symbolic Linearisability Checking of Non-Blocking Concurrent Data Structures

Nicole Cathryn du Toit

Thesis presented in fulfilment of the requirements for the degree of
Master of Science in the Faculty of Science at Stellenbosch University

Supervisor: Dr. C.P. Inggs, Cornelia Petronella Inggs

December 2021

Stellenbosch University https://scholar.sun.ac.za

PLAGIARISM DECLARATION

1. Plagiarism is the use of ideas, material and other intellectual property of another's work and to present it as my own.
2. I agree that plagiarism is a punishable offence because it constitutes theft.
3. Accordingly, all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.
4. I also understand that direct translations are plagiarism.
5. I declare that the work contained in this thesis, except otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this thesis or another thesis.

Student number	Signature
N.C. du Toit	04 October 2021
Initials and surname	Date

Copyright © 2021 Stellenbosch University

All rights reserved

Stellenbosch University https://scholar.sun.ac.za

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Cornelia Inggs, who suggested this area of research and whose expertise and guidance throughout my studies was invaluable in formulating this thesis. Your insightful feedback has pushed me to sharpen my thinking, writing, and researching skills, and has brought my work to a higher level.

Stellenbosch University https://scholar.sun.ac.za

ABSTRACT

Non-blocking concurrent data structures are developed as a more efficient solution to concurrent data structures; in non-blocking concurrent data structures hardware-level atomic instructions are used instead of higher-level, expensive locking mechanisms. Lock-free algorithms, however, are notoriously hard to design and prone to subtle concurrency errors that are difficult to pick up. Linearisability Checking is the standard correctness condition for non-blocking concurrent data structures; a data structure is linearisable if each concurrent execution of the data structure corresponds to the execution of its correct sequential specification.

In this thesis, the focus is on the linearisability checking of non-blocking data structures using a model checker. The approaches for checking linearisability using a model checker can be broadly categorised into linearisation point and automatic linearisability checking. The state-of-the-art strategies were implemented using the Java PathFinder Model Checker as basis. The linearisation point linearisability checking strategy of Vechev et al. was extended to include data structures with operations that act generically on the data structure, and not just on one element in the data structure. An improved version of Doolan et al.'s external automatic checker was implemented and the idea of an external checker was extended to the improved linearisation point checking strategy. The lazy read optimisation, proposed by Long et al., and a hash optimisation, proposed in this thesis, for the automatic checker was implemented and the effectiveness and benefit of the optimisations determined. The performance-limiting factor of the automatic checker was investigated and the claims made by Vechev et al., Liu et al., and Doolan et al. confirmed/falsified.

The concrete checker's usefulness in finding linearisability errors is constrained by the user's ability to hand-craft test cases in which errors are present. A new Symbolic Linearisability Checker was developed, the major novel contribution in this thesis, that integrates linearisability checking into Symbolic PathFinder, a symbolic model checker. The symbolic checker performs linearisability checking on all possible test cases and program paths; it verifies the linearisability of a data structure in general, constrained only by a user-defined number of operations to be executed by each thread.

Finally, extensive evaluations and comparisons of all checkers were performed, on the same model checking framework and hardware, considering their manual input required, resource usage, scalability, and ability to find errors.

Stellenbosch University https://scholar.sun.ac.za

OPSOMMING

Nie-blokkerende gelyklopende data strukture is 'n meer effektiewe oplossing as data strukture wat blokkeringsmeganismes gebruik; in nie-blokkerende data strukture word atomiese hardeware-instruksies gebruik in plaas van duur, hoër vlak, blokkeringsmeganismes. Nie-blokkerende gelyklopende data strukture is egter ingewikkeld, en is geneig om subtiele gelyklopende foute in te hê wat moeilik is om op te spoor. Lineêriseerbaarheid is die standaard korrektheidskondisie vir nie-blokkerende gelyklopende data strukture; 'n data struktuur is lineêriseerbaar as elke uitvoering van die data struktuur ooreenstem met die uitvoering van sy korrekte sekwensiële spesifikasie.

Die tesis fokus op die verifikasie van lineêriseerbaarheid van nie-blokkerende data strukture deur gebruik te maak van 'n modeltoetser. Die metodes vir die verifikosie van lineêriseerbaarheid deur gebruik te maak van 'n modeltoetser kan breedweg gekattegoriseer word in lineêrisasie-punt en outomatiese lineêrisasie toetsing. Die jongste tegnieke is implementeer deur gebruik te maak van die Java PathFinder modeltoetser as basis. Die lineêrisasie-punt lineêriseerbaarheids toetsstrategieë van Vechev et alis uitgebrei om data strukture wat generiese operasies op die data struktuur uitvoer in plaas van op 'n spesifieke element in die data struktuur, in te sluit. 'n Gevorderde weergawe van Doolan et al. se eksterne outomatiese toetser is geïmplementeer en die idee van 'n eksterne implementasie is gebruik om ook 'n eksterne weergawe van die verbeterde lineêrisasie-punt toetsstrategie te implementeer. Die lui-lees optimering wat deur Long et al. voorgestel is, en 'n hutsstrategie optimering wat in die tesis voorgestel word, is vir die outomatiese toetser geïmplementeer en die effektiwiteit en voordele van die optimerings is bepaal. Die faktore wat die effektiwiteit van die outomatiese toetser beperk is ondersoek en die stellings wat deur Vechev et al., Liu et al., en Doolan et al. gemaak is, is verifieer.

Die konkrete toetser se bruikbaarheid vir die vind van lineêriseringsfoute word beperk deur die gebruiker se vermoë om toevoergevalle wat foute sal uitwys, op te stel. 'n Nuwe simboliese lineêrisasie toetser is ontwikkel, 'n groot bydrae van die tesis, wat lineêrisasie toetsing in die simboliese PathFinder, 'n simboliese modeltoetser, integreer. Die simboliese toetser voer toetsing uit op al die moontlike toevoergevalle en al die moontlike paaie; dit verifieer dus die lineêriseerbaarheid van 'n data struktuur vir algemene gevalle, en word slegs beperk deur die gebruikersgespesifiseerde aantal operasies per liggewigproses.

Stellenbosch University https://scholar.sun.ac.za

Breedvoerige evaluering en vergelykings van al die toetsers is op dieselfde modeltoetser raamwerk en hardeware uitgevoer en die volgende is in ag geneem: die toevoer wat hul nodig het, gebruik van bronne, skaleerbaarheid, en hul vermoë om foute te vind.

Stellenbosch University https://scholar.sun.ac.za

TABLE OF CONTENTS

PLAGIARISM DECLARATION **ii**

ACKNOWLEDGEMENTS **iii**

ABSTRACT **vi**

TABLE OF CONTENTS **vii**

LIST OF FIGURES **xiii**

LIST OF TABLES **xiv**

LIST OF APPENDICES **xv**

LIST OF ABBREVIATIONS AND/OR ACRONYMS **xvi**

1 INTRODUCTION **1**

1.1 Related Work 2

1.2 Objectives 4

1.3 Contributions 4

1.4 Thesis Overview 6

2 BACKGROUND **7**

2.1 Model Checking and Java PathFinder (JPF) 7

2.1.1 Model Checking 7

2.1.2 Java PathFinder (JPF) 13

2.1.3 Symbolic PathFinder (SPF) 14

2.1.4 State space handling techniques in JPF and SPF 14

2.2 Linearisability Checking of Non-Blocking Concurrent Data Structures 16

2.2.1 Non-blocking Concurrent Data Structures 16

2.2.2 The Trace Model 19

2.2.3 Linearisability Checking 21

2.2.4 Linearisability Checking with JPF 23

Stellenbosch University https://scholar.sun.ac.za

2.3 Types of Linearisability Checking Strategies 23

3 DESIGN AND IMPLEMENTATION **26**

3.1 Linearisation Point Checking 27

3.1.1 Fixed and Non-fixed Linearisation Points 29

3.1.2 Data structures with generic operations 33

3.1.3 Conclusions for the linearisation point strategy 40

3.2 Automatic Checking 40

3.2.1 Optimisations 43

3.2.2 Conclusions for the automatic strategy 46

3.3 Implementation of the Concrete Checkers 46

3.3.1 Internal Implementation 47

3.3.2 External Implementation 50

3.3.3 Internal and External Comparison 52

3.4 Symbolic Checking 53

3.4.1 Benefits 54

3.4.2 Challenges 56

3.4.3 Implementation of Symbolic Checkers 57

3.4.4 A Hybrid Checker 60

3.5 Completeness and Soundness 61

3.5.1 JPF's state hashing optimisation causes unsoundness with respect to linearisability 62

3.5.2 A strategy to guarantee soundness with respect to linearisability of the input, in JPF 64

3.5.3 Missed Violations of the Concrete Checkers 64

3.5.4 Soundness in SPF 65

3.6 The JPF/SPF Linearisability Checking Extension Framework 66

3.6.1 jpf-linearisable 66

3.6.2 jpf-symb 67

3.7 Design and Implementation Conclusions 68

Stellenbosch University https://scholar.sun.ac.za

4 RESULTS AND ANALYSIS **69**

4.1 Machine Specs and Checker Inputs used for the experiments in this chapter 69

4.2 Efficiency 71

4.2.1 Resource usage of the Concrete Checkers 72

4.2.2 Resource usage of the Symbolic Checkers 81

4.2.3 Optimisation techniques for the Concrete Automatic Checkers 84

4.2.4 Efficiency Conclusions 89

4.3 Scalability 90

4.3.1 Scaling of the Concrete Checkers 90

4.3.2 Scaling of the Symbolic Checker 93

4.3.3 Scalability Conclusions: 98

4.4 Error Finding 98

4.4.1 Error finding of the Concrete Checkers 100

4.4.2 Error finding of the Symbolic Checker 101

4.4.3 Error finding of the Hybrid Checker 102

4.4.4 Error finding efficiency 102

4.4.5 Depth until error 105

4.5 Error Finding Conclusions 107

5 CONCLUSION AND FUTURE WORK **108**

5.1 Conclusion 108

5.1.1 Efficiency 108

5.1.2 Scalability 110

5.1.3 Input 110

5.1.4 Concrete Checkers 111

5.1.5 Symbolic Checkers 111

5.2 Final Comments 112

5.3 Future Work 113

APPENDIX A CONCURRENT HISTORY EXAMPLES **118**

Stellenbosch University https://scholar.sun.ac.za

A.1 Concurrent history generation possibilities for the LockFreeList algorithm on a particular input example 119
A.2 Linearisation-error-containing concurrent history diagrams for the BuggyQueue, SnarkDeque, LockFreeList, and PairSnap algorithms 120
A.3 Non-fixed linearisation point examples for an unsuccessful *add*, an unsuccessful *contains*, and a successful *contains* operation of the LockFreeSet data structure. 122

APPENDIX B JAVA CODE 124

B.1 BuggyQueue Java Class (SUT) 124
B.2 LockFreeSet Java Class (SUT) 125
B.3 Pseudocode for JPF's depth-first-search model checking traversal 127
B.4 Pseudocode for Automatic Linearisability Checking using JPF's listener class 128

APPENDIX C EXPERIMENT RESULTS 129

C.1 Error Finding 129

Stellenbosch University https://scholar.sun.ac.za

LIST OF FIGURES

2.1 **Execution path interleavings example.** A visual depiction of the six sequential orderings which can be derived from a program running two threads, each of which execute two atomic operations. 8

2.2 **Concrete Model Checking Example.** A concrete model checking example 9

2.3 **Model Checking.** A concrete and symbolic model checking comparison example . 11

2.4 **Java Pathfinder (JPF).** A visual illustration of the logical components of JPF Core. 13

2.5 **Symbolic Pathfinder (SPF).** A visual illustration of the logical components of JPF Core and its symbolic extension SPF. 15

2.6 **Example of linearisations derived from a concurrent history** 20

2.7 **Linearisability example with a Queue data structure.** An example of two concurrent history examples of the BuggyQueue data structure where one history is linearisable and the other not. 22

2.8 **Concurrent history and linearisations: Automatic versus Linearisation Point checking strategies** 24

3.1 **On-the-fly Linearisation Point Checking Process.** 28

3.2 **Set data structure: Fixed Linearisation Point Example.** 30

3.3 **Non-Fixed Linearisation Points (key-specific example).** A visual illustration of a concurrent history example for a Set data structure containing non-fixed linearisation points 31

3.4 **Queue data structure: Fixed Linearisation Points.** 34

3.5 **Non-Fixed Linearisation Points (linearisable key-generic example).** A visual illustration of a linearisable concurrent history example for a Queue data structure containing non-fixed linearisation points 36

3.6 **Non-Fixed Linearisation Points (non-linearisable key-generic example).** A visual illustration of a concurrent history example for a Queue data structure containing non-fixed linearisation points 37

3.7 **Automatic Linearisability Checking Process.** A visual illustration of the Automatic Linearisability Checking Process. 41

Stellenbosch University https://scholar.sun.ac.za

3.8 **Automatic Checking Process Example.** 42
3.9 **Automatic checking strategy with the lazy-read optimisation example.** . . 45
3.10 **Symbolic Linearisability Checker.** A duplicate-value concurrent history example where the symbolic checker locates a linearisation error that the concrete checker does not. 55
3.11 **Soundness in JPF.** An example showing two concurrent histories where, when traversed by JPF in order, JPF makes an incorrect cut off of path 2 due to state hashing and misses the linearisation error in the cut-off path. 63
3.12 **An example situation where the concrete checkers incorrectly verify linearisability** 65

4.1 **Unsound Concrete Checkers:** Execution time results (from Table 4.3) for entire search space traversal of the LockFreeSet algorithm in the test suite 75
4.2 **Sound Concrete Checkers:** Execution time results (from Table 4.3) for entire search space traversal of the LockFreeSet algorithm in the test suite 75
4.3 **Concrete checker performance-limiting-factor** execution time experiment results from Table 4.3. 78
4.4 **Concrete, Hybrid and Symbolic checker execution time comparison.** 82
4.5 **Symbolic checker execution time for increasing number of operations executed per thread.** 96
4.6 **Symbolic checker number of end states for increasing number of operations executed per thread.** 96
4.7 **Symbolic checker number of unique concurrent histories generated during execution for increasing number of operations executed per thread.** 97
4.8 **Symbolic checker execution time results graph for increased depth limit.** . 97
4.9 **Internal Concrete Checker Types:** Execution time results (from Table 4.13) for the time taken until the respective linearisation error was found for each of the test suite SUTs. 103
4.10 **External Concrete Checker Types:** Execution time results (from Table 4.13) for the time taken until the respective linearisation error was found for each of the test suite SUTs. 103

Stellenbosch University https://scholar.sun.ac.za

4.11 **Symbolic Checker Types:** Execution time results (from Table 4.13) for the time taken until the respective linearisation error was found for each of the test suite SUTs. 104

A.1 **Diagram depicting the 31 possible concurrent histories for the LockFreeSet SUT.** . . . 119
A.2 **A linearisation error containing concurrent history for the BuggyQueue algorithm** . . . 120
A.3 **A linearisation error containing concurrent history for the LockFreeList algorithm — Bug1** . . . 120
A.4 **A linearisation error containing concurrent history for the LockFreeList algorithm — Bug2** . . . 121
A.5 **A linearisation error containing concurrent history for the SnarkDeque algorithm — Bug1** . . . 121
A.6 **A linearisation error containing concurrent history for the SnarkDeque algorithm — Bug2** . . . 121
A.7 **A linearisation error containing concurrent history for the PairSnap algorithm** . . . 122
A.8 **Illustration of a non-fixed linearization point for an add operation** . . . 122
A.9 **Illustration of a non-fixed linearization point for a contains operation** . . . 123
A.10 **Illustration of a non-fixed linearization point for a true returning contains operation** . . . 123

Stellenbosch University https://scholar.sun.ac.za

LIST OF TABLES

3.1 **Defining the types of linearisability checkers.** . . . 27
3.2 Test suite of SUT algorithms and the operation-sequence test cases used for the concrete, hybrid, and symbolic checkers. . . . 60

4.1 **Inputs required for the different linearisation checker implementations.** . . . 70
4.2 Test suite of SUT algorithms and the operation-sequence test cases used for the concrete, hybrid, and symbolic checkers. . . . 71
4.3 **Concrete: Execution Time and Memory for Entire Search Space** . . . 73
4.4 **Concrete: Search Space Statistics for Entire Search Space** . . . 74
4.5 **Concrete, Hybrid, Symbolic: Execution Time and Memory for Entire Search Space.** . . . 83
4.6 **External Unsound** Concrete Automatic: Optimisation results over entire search space . . . 86
4.7 **External Sound** Concrete Automatic: Optimisation results over entire search space 86
4.8 **Internal Unsound** Concrete Automatic: Optimisation results over entire search space . . . 87
4.9 **Internal Sound** Concrete Automatic: Optimisation results over entire search space 87
4.10 **External Concrete scaling results for an increase in the number of operations executed per thread for 1, 2, and 3 executing threads.** . . . 92
4.11 **Data showing the exponential increase in the number of test cases for a single thread execution of a SUT with 1/2/3 methods when the number of operations for the thread to execute is increased** . . . 94
4.12 **External Symbolic scaling results for an increase in the number of operations executed per thread for 1 executing thread.** . . . 95
4.13 **Linearisability error finding ability for all checker types.** . . . 99
4.14 **Depth at which each error was found and max depth till the error.** . . . 106

C.1 **Execution Time and Memory until error is found** . . . 129

Stellenbosch University https://scholar.sun.ac.za

LIST OF APPENDICES

APPENDIX A CONCURRENT HISTORY EXAMPLES

APPENDIX B JAVA CODE

APPENDIX C EXPERIMENT RESULTS

Stellenbosch University https://scholar.sun.ac.za

LIST OF ABBREVIATIONS AND/OR ACRONYMS

JPF	Java PathFinder
SPF	Symbolic PathFinder
Int.	Internal
Ext.	External
Aut.	Automatic
Lin. Point	Linearisation Point

Stellenbosch University https://scholar.sun.ac.za

CHAPTER 1

INTRODUCTION

Non-blocking concurrent data structures are developed as a more efficient solution to concurrent data structures. Concurrent data structures share their information amongst different threads and race conditions can arise when more than one thread simultaneously access the same location in the data structure. Locking mechanisms can be used to provide mutual exclusive access to critical sections and avoid race conditions, but they reduce the efficiency of the program because of the locking overhead which includes waiting for locks to be released and acquiring and releasing them. In non-blocking concurrent data structures hardware-level atomic instructions are used instead of higher-level, expensive locking mechanisms.

Lock-free algorithms are, however, complex to design and prone to subtle concurrency errors that are difficult to pick up. Designing and proving their correctness is notoriously difficult, several published data structures have been shown to contain errors even in situations where manual proofs were attempted [9, 33, 7, 32]. Attempts to derive non-blocking concurrent data structures from sequential specifications have resulted in algorithms that perform very poorly compared to locking data structures [1, 3, 13, 21]. Linearisability checkers can be used to prove the correctness of non-blocking data structures.

Linearisability Checking is the standard correctness condition for non-blocking concurrent data structures and ensures that each concurrent execution of a data structure corresponds to the execution of its correct sequential specification. Herlihy et al. proposed linearisability and defined it as a non-blocking and local property [14, 15]. A property of a concurrent system is said to be non-blocking if each pending invocation event of an operation is never required to wait for another pending invocation event to complete. A property P of a concurrent system is said to be local if the system as a whole satisfies P whenever each individual data structure in the system satisfies P. A non-blocking concurrent data structure is correct with respect to linearisability if each execution of the data structure is linearisable.

A number of verification techniques for checking linearisability have been developed over the years including approaches such as data refinement, reduction, manual proofs, static and runtime analysis, and model checking [15, 39, 22, 8]. For example, Herlihy et al. presented a technique for manually

Stellenbosch University https://scholar.sun.ac.za

proving linearisability and illustrated their strategy using a Queue example[15]. Dongol et al. discussed some foundational strategies which have been used to verify linearisability such as data refinement, shape analysis, and reduction; and compared the advantages and limitations of each of the strategies [8]. Elmas et al. presented a runtime technique for checking the correctness of concurrent programs and Flanagan presented a solution to verifying Commit-Atomicity, which is similar to linearisability, using a model checker. Elmas et al. and Flanagan's solutions used manually specified points in a program and only apply to algorithms where the points can be specified for each operation in the algorithm code [10, 11].

In this thesis the state-of-the-art strategies for the linearisability checking of non-blocking concurrent data structures, using model checking as a basis, were implemented [38, 39]. Model checking is an automated property verification technique which, by definition, systematically and exhaustively explores all possible execution sequences of a program [6].

1.1 RELATED WORK

The approaches for checking linearisability using a model checker can be broadly categorised into linearisation point [38, 39, 22, 37] and automatic [38, 4, 23, 35, 24, 9] linearisability checking. Linearisation points are specified in the operations of a concurrent data structure; the linearisation point of an operation is considered to be the instant, between the operation's invocation and response events, at which the operation takes effect.

Both linearisation point and automatic linearisability checking techniques are based on the execution of a correct sequential version of the data structure alongside the execution of the non-blocking concurrent data structure, and the checking that both versions yield the same final state for each operation run during program execution. However, the former approach requires the user to manually identify the algorithm-specific linearisation points, whilst the latter automatically performs linearisability checking without the manual identification of linearisation points.

Vechev et al. extend Flanagan's work on checking commit-automaticity using the model checker SPIN; they use their `PARAGLIDER` tool and the SPIN model checker [36] for the linearisation point linearisability checking of data structures not only with operations that have specifiable linearisation points, but also those with operations for which linearisation points cannot be determined [38].

Stellenbosch University https://scholar.sun.ac.za

They formalised their extension and define non-fixed linearisation points for situations of unspecified linearisation point operations [39]. Interestingly, linearisability checking with non-fixed linearisation points has also been considered by other techniques such as Liang et al. who proposed a solution based on refinement [22], and Vafeiadis proposed a solution which uses prophecy variables [37]. Since algorithm-specific linearisation points are user-intensive and often difficult to determine, it is desirable to have an automatic solution for linearisability checking that does not require user-specified linearisation points.

Liu et al. proposed an automatic strategy for linearisability checking using a model checker, their strategy is based on refinement and can check linearisability without user-specified linearisation points but is also able to take advantage of linearisation points when they are available [23].

Vechev et al. described an automatic checking strategy in which the concurrent execution information, for each execution path generated by the model checker, is maintained at the model checker's states along that path. At each path end state, the concurrent execution information is used to generate all possible sequential interleavings of the concurrent execution's operations. If at least one sequential interleaving is equivalent to the execution of a correct sequential specification, then the concurrent execution is considered linearisable [38].

Burckhardt et al. [4] developed the first complete and automatic tool for automatic linearisability checking, called Line-Up. Their tool enumerates and checks all sequential behaviour of a program execution; they built their tool on top of the stateless model checker CHESS [26]. Unlike Vechev et al.'s strategy, their tool works on full featured code and not just single data structures.

Doolan et al. presented an optimisation to the automatic checking strategy. The automatic checking strategy described by Vechev et al. uses code instrumentation to include the concurrent execution information at the model checker's states, and for the linearisability checking logic to execute at the end states. Doolan et al.'s strategy instead outputs the concurrent execution information generated by the model checker to some external log. An external automatic linearisability checking tool then uses the logged information to perform the sequential interleaving generation and linearisability checking. The optimisation aims to reduce the size of the model checker's state space by minimising the information stored at its states and allowing the model checker to optimise state space exploration by backtracking mechanisms [9].

Stellenbosch University https://scholar.sun.ac.za

Long et al. proposed an optimisation technique to the automatic checking algorithm which they call lazy read acceleration. The optimisation aims to reduce the number of sequential interleavings generated for each concurrent execution produced by the model checker [24].

Various claims have been made in the literature regarding the performance-limiting factor of the model checker utilising automatic checking tools. Vechev et al. and Liu et al. claimed that the automatic linearisability checking logic does not scale, in time or memory, and that the automatic checking process is the performance limiting factor [39, 23]. Doolan et al. investigated the performance of their unoptimised automatic implementation and report that the performance limiting factor of execution is the model checker's concurrent-execution generation process [9].

1.2 OBJECTIVES

We aim to integrate all state-of-the-art linearisability checking techniques into Java PathFinder Model Checker (JPF), perform reproducibility tests to confirm state-of-the-art results, develop a symbolic linearisability checker by integrating linearisability checking into Symbolic PathFinder (SPF), and compare all implemented checker types on a uniform system with the same model checker and hardware.

1.3 CONTRIBUTIONS

The linearisation point checking and automatic linearisability checking techniques of Vechev et al. have been integrated with the Java PathFinder model checker [38, 39].

We propose a **hash optimisation** for the automatic checking strategy that avoids re-computation of concurrent executions that are generated by the model checker and equivalent with respect to linearisability. We perform experiments to determine the effectiveness and efficiency benefit of this hash optimisation as well as the lazy read optimisation proposed by Long et al. [24]. We evaluate these optimisations on the checkers and compare the benefit of the optimisations for each checker.

We have improved the external automatic checker developed by, Doolan et al [24, 9], by extracting not only the concurrent execution information and linearisability logic from the model checker's search space but also the logging logic; we present a **completely external linearisability checker** that does not require any extra information included in the model checker's search space. We define

Stellenbosch University https://scholar.sun.ac.za

those checkers that do not use the external optimisation as Internal Checkers, and those that do as External Checkers.

In the literature, Vechev et al. proposed the linearisation point strategy for the Set data structure. Their strategy can be generalised to data structures with operations that act on a specific element in the data structure but are incompatible for operations that act generically on the data structure. We **extend their linearisation point strategy to handle generic operations** and use a Queue data structure as an example (Section 3.1.2).

For a linearisability checking tool to be considered sound, it must guarantee linearisability of the concurrent data structure, on the input situation, given that its execution did not find any linearisation errors. We have developed a **solution to guarantee sound linearisability checking in JPF** and explain why this solution is necessary. We have created the linearisability checking tools in such a way that the user can easily configure the tool to either run a sound or unsound linearisability checking execution.

We extend the idea of linearisability checking with a concrete model checker to that of a symbolic model checking setting; a **Symbolic Linearisability Checker**. The Symbolic Linearisability Checker combines symbolic execution [20] with model checking and constraint solving for automated linearisability error detection and test case generation for Java programs [29], [2], [27], [30]. We have chosen to implement the automatic checking strategy for the symbolic domain because it is most logically compatible with SPF's framework as discussed in Section 3.4.3, and to implement it as as an external checker because of the performance benefits of the external optimisation as discussed in Section 4.2.1.2.

We also present a variation of the ordinary symbolic checker which is a **Hybrid (concrete-symbolic) Linearisability Checker**. The purely symbolic checker performs automatic test case generation and symbolic execution to automatically execute all possible operation sequences and traverse all possible program paths for a given number of operations per thread. The hybrid checker uses symbolic execution to traverse all program paths, but does not use the automatic test case generation so checks only one exact sequence of operations per thread.

An **evaluation and comparison for all checkers is performed on the same model checking framework and hardware**; considering their manual input required, resource usage, scalability,

Stellenbosch University https://scholar.sun.ac.za

and ability to find errors. The evaluation also determines the performance-limiting factors of each of the implemented checkers, and confirm/falsify the related claims made by various authors in the literature.

1.4 THESIS OVERVIEW

In Chapter 2 the background for model checking and linearisability checking is explained in detail and an overview of the linearisation point and automatic checking strategies described in the literature, is provided. The model checkers used in this thesis are Java PathFinder (JPF) and its symbolic extension Symbolic PathFinder (SPF); these model checkers are described in Section 2.1 [20, 19, 29, 2, 30].

In Chapter 3 the intricacies of the linearisation point and automatic checking strategies (Sections 3.1 and 3.2) are described, the design and implementation details for our internal concrete checkers (Section 3.3.1) explained, the implementation details of our novel external versions of these strategies given (Section 3.3.2), and a hash optimisation for the automatic linearisability checker (Section 3.2.1.2) proposed. We present a new Symbolic Linearisability Checker and discuss its design as an Automatic Linearisability Checker (Section 3.4) and external implementation, as well as the benefits associated with these categories (Section 3.4.3). We present both a symbolic checker and a hybrid (concrete-symbolic) version called Hybrid Checker. Soundness is discussed in detail and the linearisation checkers were implemented such that they can run in a mode that guarantees soundness or a mode that does not guarantee soundness with respect to linearisability (Section 3.5).

In Chapter 4 we evaluate and compare each of the checkers with respect to their manual input requirements (Table 4.1), efficiency (Section 4.2), scalability (Section 4.3), and error finding capability (Section 4.4).

In Chapter 5 we give our findings and conclusions for this thesis and discuss areas of future work.

Stellenbosch University https://scholar.sun.ac.za

CHAPTER 2

BACKGROUND

In this chapter we an overview of the foundational concepts for linearisability checking of non-blocking concurrent data structures using model checking, is provided. Model checking, symbolic model checking, and the model checking tools used in this thesis are described in Section 2.1. Non-blocking concurrent data structures are defined in Section 2.2.1, the trace model used to represent an execution of the non-blocking data structures is illustrated in Section 2.2.2, linearisability checking is explained in Section 2.2.3, and an overview of the linearisability checking strategies presented in the literature is provided in Section 2.3.

2.1 MODEL CHECKING AND JAVA PATHFINDER (JPF)

Model checking is an automated property verification technique, that systematically and exhaustively explores all possible execution paths of a program. Java PathFinder (JPF) has been chosen as the model checking tool for the work in this thesis.

2.1.1 Model Checking

Model checkers prove that certain properties hold for some input program and when a property does not hold, return the exact execution sequence in which the property is violated. This is in contrast to testing that can only prove the presence of errors in a program, not the absence of them.

The JPF model checker exhaustively explores all possible bytecode interleavings of a concurrent execution and thus is particularly effective in finding subtle errors in complex concurrent systems; it takes into consideration all the branching statements and different thread interleavings for the input program. The exhaustive exploration of all execution paths does, however, cause the model checker's search space to grow exponentially for an increase in the number of program instructions or executing threads.

For example, a program with two threads, each executing two atomic instructions, yields six unique concurrent execution paths. For four, eight and sixteen atomic instructions per thread the number of unique paths is 70, 12,870 and 601,080,390; an obvious exponential trend [16]. Figure 2.1 illustrates the six different execution paths for the example two atomic instructions per thread example.

Stellenbosch University https://scholar.sun.ac.za

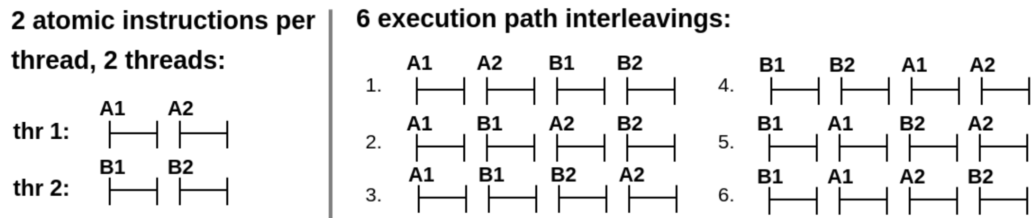


Figure 2.1: **Execution path interleavings example.** A visual depiction of the six sequential orderings which can be derived from a program running two threads, each of which execute two atomic operations.

2.1.1.1 Concrete Model Checking

A concrete, or otherwise known as explicit state, model checker performs model checking using concrete variable values. The values are used to choose a single execution path for each branching statement encountered during execution. Figure 2.2 depicts an example concrete model checking execution.

The following elements are present in the diagram:

(a) **The code fragment** is the program executed by the model checker. The model checker executes the method *thread1Operation()* on thread-1 and the operation *thread2Operation()* on thread-2.

(b) **thread1Operation()** swaps the values in `x` and `y` and includes two branching statements (lines 4 and 8).

(c) **thread2Operation()** assigns the concrete value 1 to the variable `x`.

(d) **Single-border rectangles** represent the program states maintained by the model checker. Program states contain variable names and associated values; the variable values are updated during transitions from one state to another.

(e) **State transitions** are represented by arrows which connect each parent state with one or multiple child states. Each transition corresponds to a line in the code fragment. At branching statements (lines 4 and 8), the path choice is made according to the concrete values contained in the parent state's variables.

(f) **Double-bordered rectangles** represent program end states which, like ordinary states,

Stellenbosch University https://scholar.sun.ac.za

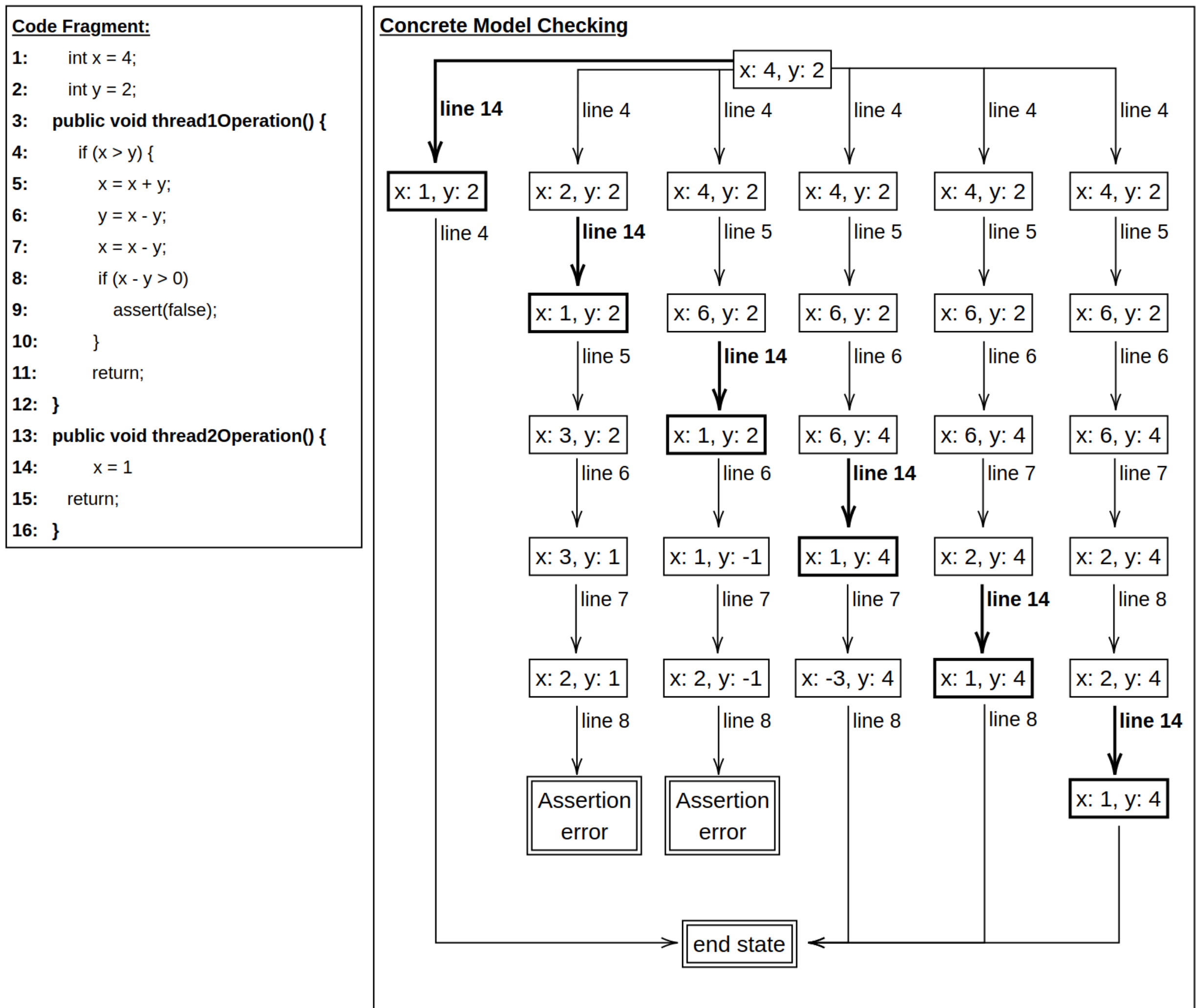


Figure 2.2: **Concrete Model Checking Example.** A diagram showing the search space, including states and transitions between states, for a concrete model checker's traversal of the code fragment algorithm that swaps the values of two variables

Stellenbosch University https://scholar.sun.ac.za

contain variables and their associated values but also represent the end of an execution path.

(g) **Bold arrows** represent transitions containing thread-2's instructions, ordinary arrows represent transitions containing thread-1's instructions.

(h) **Bold state borders** represent states for which the last instruction was executed by thread-2, ordinary state borders represent instructions executed by thread-1.

The model checker must generate all possible execution paths for this code-fragment. To do this it interleaves the operations of all the executing threads. Thread-2 only executes one instruction so the interleaving options are those where this thread-2 instruction is positioned at all possible placements between the instructions of thread-1; resulting in six different execution paths.

The diagram shows that although all execution paths begin with the same start state, the order in which each thread's instructions occur effects the program end states. For example the leftmost path shows that when thread-2's instruction is executed before all thread-1 instructions, it results in an end state where variable `x` is equal to 1 and variable `y` is 2. In the second path from the left, when just the first operation of thread-1's instruction is executed before that of thread-2, the resultant end state is an assertion error. In the fourth path from the left, `x` is -1 and `y` is 4, at the end state.

For simplicity, the example in Figure 2.2 illustrates statement interleavings, instead of bytecode interleavings. JPF actually interleaves the bytecode instructions of the executing threads; and each statement in the code may require multiple bytecode instructions. The model checker verifies that a property holds for the program on some concrete input, by proving that the property holds for all the reachable paths of the program.

2.1.1.2 Symbolic Model Checking

A Symbolic Model Checker combines symbolic execution with model checking to not only explore all execution paths for some concrete input situation, but also to explore all paths for all possible concrete input situations. It does this by using symbolic values in place of concrete values, each symbolic value can represent a range of concrete input values [20]. This checker uses the symbolic values to generate all execution paths for all input value situations and thus explore all reachable sections of the input program; maximising program path coverage.

Figure 2.3 shows side-by-side, the concrete checker's execution of an example single-threaded program (SUT) and the corresponding symbolic checker's execution of the same program (SUT). This program is serial for the sake of simplicity but the discussion can be applied to multi-threaded situations which would add another layer of complexity, thread-interleavings.

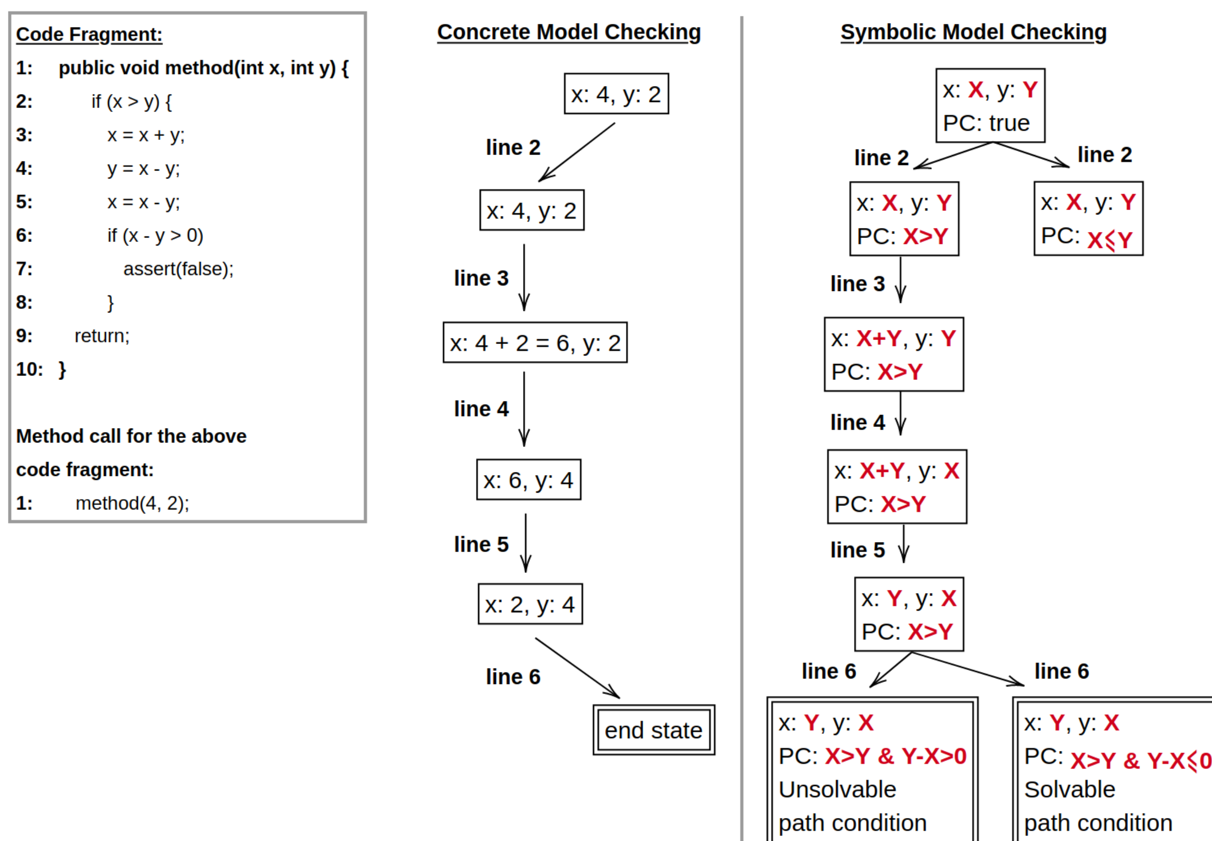


Figure 2.3: **Model Checking.** A diagram showing the search space, including states and transitions between states, for a concrete and for a symbolic model checker's traversal on the same code fragment algorithm that swaps the values of two variables. This example was adapted from Visser et al. [19]

We use the same diagram representations as for the example in Section 2.1.1 but for the symbolic diagram there are a few added complexities.

- The concrete model checker uses concrete values for `x` and `y`, the symbolic checker uses symbolic values instead; the symbolic values `X` and `Y` are assigned to the variables, respectively. Symbolic states thus represent sets of concrete states.

Stellenbosch University https://scholar.sun.ac.za

- For instruction updates that occur during transitions between states, the concrete checker has one child state for each parent state because the concrete variable values necessitate only one particular path taken for any given instruction. Symbolic execution instead updates the variables in terms of the symbolic values. For example, the variable value update of line 3 is updated to: `x` equals 6 and `y` equals 2 for the concrete checker and `x` equals `X+Y` and `y` equals `Y` for the symbolic checker.
- A path condition is included at each of the symbolic model checker's states. The path condition contains symbolic-value constraint rules that correspond to the branching decisions made along the path to that state: `if` and `while` (lines 2 and 6 respectively for example) are branching statements.

  The concrete checkers use the branching statement to create a single child state that represents the path taken for the true or false result of the statement, given concrete variable values. For example the branching condition at line 2 is `x > y` and for concrete values `x` equals 4 and `y` equals 2 the path will take the path of entering the `if` statement.

  The symbolic checker creates a child for each possible result situation and updates the path condition of the child states according to the branching choice made. For example, the boolean branching condition at line 2 is `x > y` and so the symbolic checker creates two children, one with the path condition rule `x > y` (entering the `if` statement) and the other `x <= y` (not entering the `if` statement). The path condition of each child contains rules for the range of concrete variable values that correspond to the branching path decisions made to that end state. Thus all possible branching choices are accounted for.
- For an unsolvable path condition at an end state, there is no possible configuration of variable values which will result in the execution path to that end state; conversely for a solvable path, there is a possible configuration of variable values which will result in the program execution reaching the end state. For example, the leftmost end state in the symbolic diagram example has an unsolvable end state because `X` cannot be both bigger and smaller than `Y` at the same time. The rightmost end state is solvable since there is a set of values which satisfies the constraints of the path condition.

Stellenbosch University https://scholar.sun.ac.za

2.1.2 Java PathFinder (JPF)

JPF is a concrete model checker that runs on the Java Virtual Machine (JVM) and analyses the bytecode of a Java input program. The program is compiled to Java bytecode and then executed on JPF's own custom virtual machine (VM) to find program defects; we call this input program the System Under Test (SUT).

JPF's custom VM allows it to control the program execution of the SUT. Java's VM (JVM) follows only a single execution path for the program when it is run i.e., at every branching statement it selects a single execution path. JPF's VM, on the other hand, is able to identify and explore all possible paths for a branching statement by generating state representations that include all path decision options. Each state is stored and can be restored during backtracking; allowing JPF to exhaustively explore all of the possible execution paths of the SUT, for all branches and thread interleavings, during model checking. JPF's VM can thus traverse a program's execution behaviour by moving forwards and backwards between these states, Java's VM can only move forward. JPF systematically explores all the possible execution paths of the SUT and can be used to verify that none of the execution paths violate a specified property for the SUT. Figure 2.4 depicts the logical components of JPF and the input/output for the tool's execution.

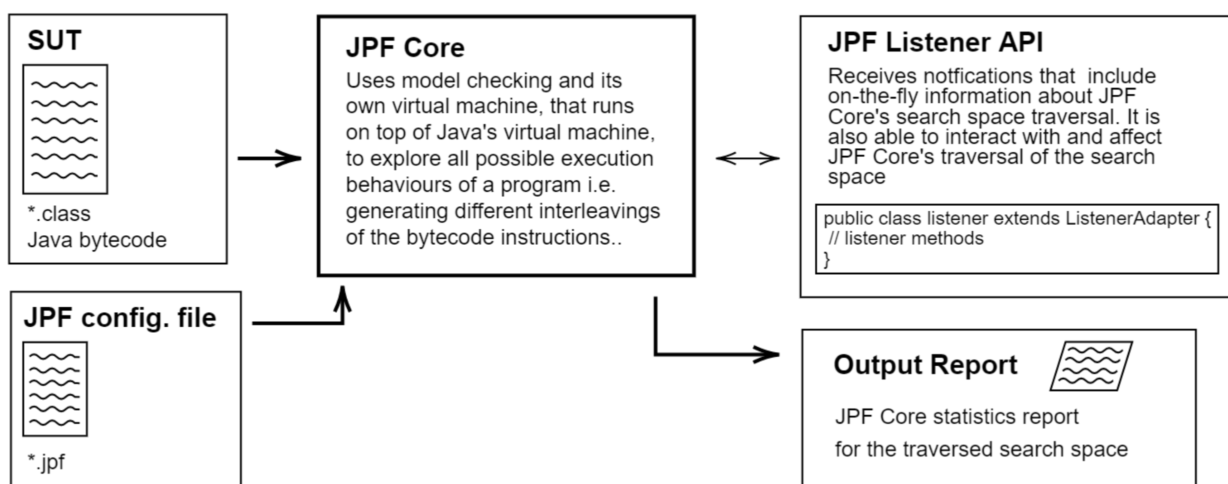


Figure 2.4: **Java Pathfinder (JPF).** A visual illustration of the logical components of JPF Core, which includes: an input configuration file, a Java input program to test, JPF Core itself, JPF Core's Listener class, and an output statistics report.

JPF takes as input a **config file** which it uses to determine the model-checking-execution settings,

Stellenbosch University https://scholar.sun.ac.za

and a **SUT** which it compiles to Java bytecode before performing model checking on it. JPF has been designed to facilitate ongoing research and thus provides extension mechanisms that developers can use to manipulate the search, one such extension mechanism is the listener. **Listeners** do not require any modification to JPF and provide a way to observe, influence, and extend JPF's execution by communicating with JPF during the model checking. The listener receives information about the search space traversal and is able to interact with JPF's VM to alter the model checker's search. JPF produces a **statistics report** on completion of its execution or identification of a program error; the report details information about the search and if relevant, the error encountered.

2.1.3 Symbolic PathFinder (SPF)

Symbolic PathFinder (SPF) is a Symbolic Model Checker that extends JPF [2, 30]. It uses the analysis engine of JPF but where JPF executes the SUT with concrete variable values, SPF executes the SUT using symbolic variable value expressions. Each variable is represented by an expression, in terms of the symbolic values, that allows the variable to represent a range of concrete variable values.

SPF maintains an execution tree, where each state contains a list of program variables, their corresponding symbolic expressions and a path condition. At each choice along the path, a constraint that the input values must satisfy to reach that branch, is added to the path condition. In SPF, each time a constraint is added, the satisfiability of the path condition is checked; if the path condition is not satisfiable then the model checker backtracks to avoid unnecessary computation.

Figure 2.5 depicts the logical components of SPF, extending the component diagram of JPF, and the input/output for the tool's execution. The tool takes as input the SUT and interacts with JPF to perform the model checking. It is able to make use of a listener and once SPF has reached an error or the end of execution it produces a statistics report of the execution.

2.1.4 State space handling techniques in JPF and SPF

JPF, and by extension SPF, uses two main strategies to alleviate state space explosion: State Hashing and Partial Order Reduction (POR) [2].

Stellenbosch University https://scholar.sun.ac.za

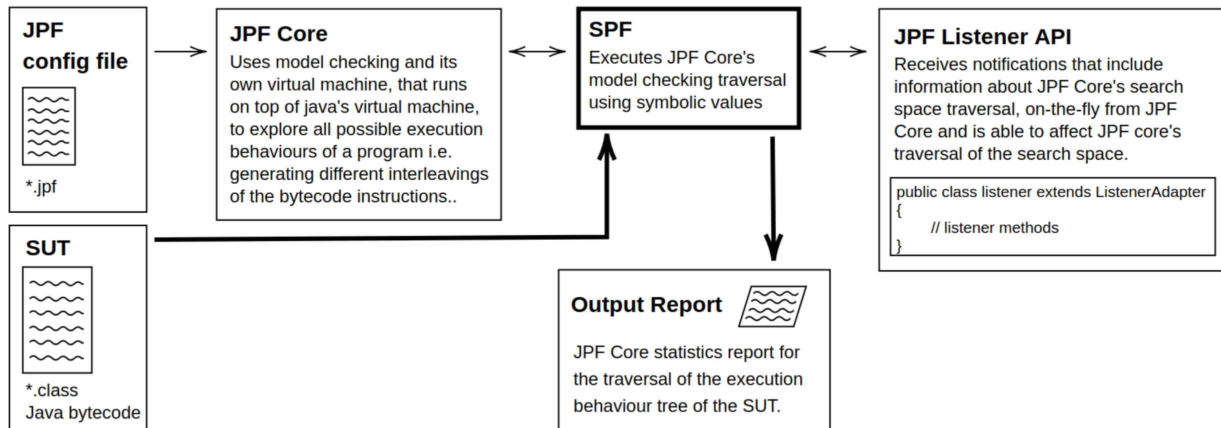


Figure 2.5: **Symbolic Pathfinder (SPF).** A visual illustration of the logical components of JPF Core and its symbolic extension SPF, made up of: an input configuration file, Java input program to test, JPF Core itself, the SPF extension, SPF's Listener class, and an output statistics report.

2.1.4.1 Partial Order Reduction

The group of bytecode operations that are executed for JPF to move from one state to another is called a transition. A transition consists of at least one bytecode instruction that results in the alteration of the current state to form a new child state. For multi-threaded programs, JPF interleaves all the possible bytecode operations of the threads so that all possible execution paths are traversed. To optimise this process, JPF groups together sets of instructions that are allocated to a single thread and that do not affect anything outside of the thread itself, to execute within a single transition; this process is called Partial Order Reduction (POR). POR has been shown to be effective in the alleviation of state space explosion with a more than 70% reduction in state space [18].

2.1.4.2 State Hashing

JPF uses a hashtable to store visited states [16]. The JenkinsStateSet.java class is used by JPF as the hashtable and the hashes are based on Jenkins hashes [17]. Alternative state set implementations can be added to the framework and configured in the jpf.properties file of JPF or the field can be set to empty where no state hashing is to be used.

- extends SerializingStateSet.java which implements StateSet.java

Stellenbosch University https://scholar.sun.ac.za

- calls CFSerializer.java - The default serializer is JPF's CFSerializer. Alternative serializers can be added to the framework and configured in the jpf.properties file of JPF. The serializers in JPF are used to serialize a state into a format that can be used in hashing [5].

Unfortunately, the hashing used by JPF compromises soundness with respect to linearisability. Section 3.5 discusses the details of the interaction between state-hashing and linearisability checking and our proposed soundness-guaranteeing solution for JPF.

2.2 LINEARISABILITY CHECKING OF NON-BLOCKING CONCURRENT DATA STRUCTURES

Non-blocking concurrent data structures are defined in Section 2.2.1, the trace model used to represent concurrent executions of these data structures are defined in Section 2.2.2, linearisability is defined in Section 2.2.3, and two linearisability checking strategies presented in the literature: 1. Linearisation Point Linearisability Checking, and 2. Automatic Linearisability Checking are described in Section 2.3.

Linearisability is used to verify the correctness of non-blocking data structures, it guarantees that every concurrent execution of a data structure correlates to a correct sequential execution of the data structure.

2.2.1 Non-blocking Concurrent Data Structures

A data structure has a type that defines a set of possible values and a set of primitive operations that provide the only means to create and manipulate that data structure; examples of data structures are Sets, Queues, abd Arrays. For concurrent data structures that share their information amongst different threads, race conditions can arise where more than one thread attempts to access a single location in the data structure.

Blocking concurrent data structures avoid race conditions by using locks to synchronise access to resources within the data structure, but they reduce the efficiency of the program because the locks need to be acquired and released and allow only one thread at a time to execute the code in the critical section; i.e. the region of code that contains the statements should be executed in serial to avoid race conditions. This overhead can be considered expensive in performance-critical

Stellenbosch University https://scholar.sun.ac.za

concurrent programs.

Non-blocking concurrent data structures, also known as non-blocking concurrent objects in object oriented environments, instead of locking sections of code, use atomic hardware operations to perform updates to resources. In Java, the compareAndSet operation is the wrapper for the hardware compare-and-swap (CAS) operation; the CAS operation pseudocode is shown in the code fragment below.

CAS operation pseudocode:

```
begin operation CAS(address, oldValue, newValue):
  begin atomic
    if address equals oldValue then:
      address = newValue
      return true
    else
      return false
  end atomic
end operation
```

Making use of fewer and more efficient safety mechanisms allows the non-blocking data structures to benefit from an increase in time efficiency and performance. These data structures are, however, notoriously hard to design, implement and verify which can easily result in subtle concurrency errors that arise due to the exponential increase in the number of interleavings with respect to the number of concurrent processes. As stated by Doolan et al. [9], several published concurrent data structures have been shown to contain errors even in situations where manual proofs were attempted [33, 7].

In the following code fragments we show the *enqueue* operation for the BuggyQueue algorithm (see the test suite algorithms in Section 4.1) with a lock-utilising solution and a non-blocking CAS-atomic-operation-utilising solution, respectively.

Blocking BuggyQueue enqueue operation using a lock:

```
1. public boolean enqueue(int item) {
2.   synchronized(lock) {
3.     Q[REAR%L] = new Node(item, Q[REAR%L].counter+1, false);
```

Stellenbosch University https://scholar.sun.ac.za

```
4.   }
5.   REAR = REAR + 1;
6.   return true;
7. }
```

Listing 2.1: **Non-blocking BuggyQueue enqueue operation using atomic CAS operations:**

```
1.  public boolean enqueue(int item) {
2.    int rear;
3.    Node x;
4.    boolean resultFound = false;
5.    do {
6.      do {
7.        rear = REAR.get();
8.        x = (Node) Q_atomic.get(rear%L);
9.      } while (rear != REAR.get()  rear == FRONT.get()+L);
10.     if (x.isIntNull) {
11.       // CAS operations
12.       if (Q_atomic.compareAndSet(rear%L, x,
              new Node(item, x.counter+1, false))) {
13.         REAR.compareAndSet(rear, rear+1);
14.         resultFound = true;
15.       }
16.     } else {
17.       REAR.compareAndSet(rear, rear+1);
18.     }
19.   } while (!resultFound);
20.   return true;
21. }
```

The lock-utilising *enqueue* operation is simple in comparison to the equivalent operation using CAS-atomic-operations. The non-blocking BuggyQueue algorithm is an example of a situation in which the complex nature of implementing non-blocking algorithms can lead to subtle concurrency errors; see Shann et al.'s article on the BuggyQueue for a description of the subtle concurrency error of this algorithm [33].

Stellenbosch University https://scholar.sun.ac.za

Lock-Free versus Wait-Free

The non-blocking concurrent data structures referred to in this thesis are lock-free but they are not wait-free. For example, a slow thread could keep retrying a CAS indefinitely whilst a faster thread repeatedly performs an update in between the slow thread's read and CAS steps. Friggens defines wait-freedom and lock-freedom as follows [12]:

- Wait-free — A concurrent data structure is considered to be wait-free when every thread can complete its execution within a finite number of its own program steps. Each thread thus acts independently of the number or behaviour of other executing threads. Wait-freedom is considered as ideal concurrent behaviour but in practice wait-free data structures are very difficult to design.
- Lock-free — A concurrent data structure is considered to be lock-free when each of the executing threads is able to complete its execution within a finite number of program steps, that may include program steps in other threads. Lock-free data structures are easier to implement than wait-free data structures and provide a more ideal concurrent behaviour than lock-containing data structures but does not produce the most ideal concurrent behaviour of a completely wait-free execution.

2.2.2 The Trace Model

To formally describe linearisation checking, we define a trace model that represents a concurrent execution and is made up of operations on the non-blocking concurrent data structure.

A **concurrent history** is a finite sequence of operation invocation and response events on a concurrent data structure. Every event must include the data structure's name, the operation name, the arguments, and the id of the thread associated with the event [15]. A response matches an invocation if their data structure names agree and their thread ids agree. An operation, `e`, is a pair consisting of an invocation `inv(e)` and response, `res(e)` event on the data structure. We assume that each thread in a concurrent history sequentially executes a sequence of operations on the data structure. Since the operations of non-blocking concurrent data structures are not executed as atomic units, it is possible for operations in different threads to overlap within a concurrent history.

Stellenbosch University https://scholar.sun.ac.za

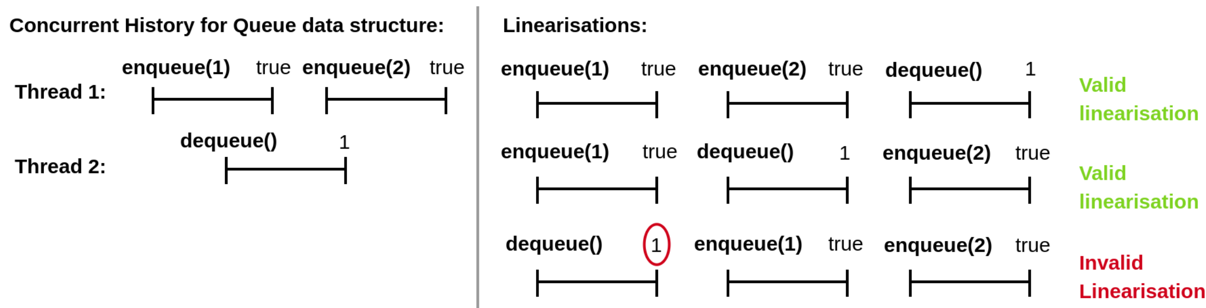


Figure 2.6: **Example of linearisations derived from a concurrent history** A concurrent history for a Queue data structure with the three derivable linearisations of it. Two of the three linearisations are valid. We assume the starting Queue is empty.

A **sequential history** is a special case of a concurrent history where only one thread is executing; each invocation is immediately followed by its matching response and, except the last, each response is immediately followed by an invocation.

A **linearisation** is one possible order in which the operations of a concurrent history can be ordered as a sequential history; considering that overlapping operation calls can take effect in any order, but non-overlapping operation calls must take effect in their real-time order. A concurrent history can have many possible linearisations because the overlapping operations can be ordered in many possible ways; each of the linearisations can be either valid or invalid. A linearisation is valid with respect to a sequential specification if exactly the same sequence of operation invocation and response events can be generated by a correct sequential execution; otherwise it is invalid. Figure 2.6 shows a concurrent history trace on a Queue data structure with two executing threads and the two valid, one invalid linearisation which is derivable from the concurrent history.

On the left of Figure 2.6 is a concurrent history trace of an execution where thread-1 executed an `enqueue(1)` operation and then an `enqueue(2)` operation, and thread-2 executed a `dequeue()` operation which overlapped with both of the thread-1 operations. The right of the figure shows the three linearisations which can be derived from the concurrent history. The overlapping rules allow the thread-2 `dequeue` operation to be interleaved before, between, and after the two `enqueue` operations of thread-1. The figure shows that the top two linearisations are valid; the response values for the operations of each of these two linearisations corresponds to that of a correct sequential specification. The last linearisation, however, is not valid; the response value of the first `dequeue`

Stellenbosch University https://scholar.sun.ac.za

operation in the linearisation incorrectly returns the value 1 (indicated by the red oval) where the Queue was empty and should thus return empty.

2.2.3 Linearisability Checking

Non-blocking concurrent objects are notoriously hard to design without subtle concurrency errors arising. Linearisability is the standard correctness criterion for concurrent data structures.

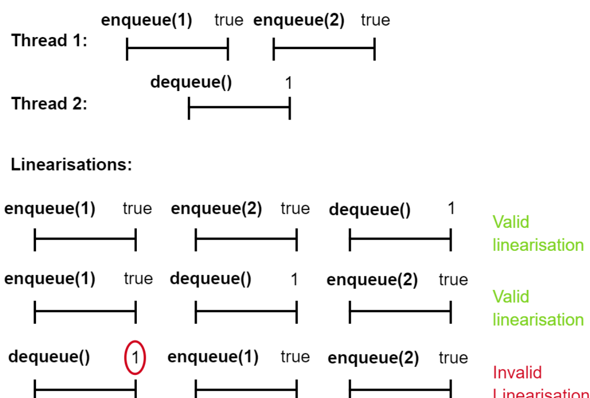
Informally, linearisability assumes that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that a concurrent execution can be represented as a sequential ordered list of events; we call this sequential ordering a linearisation. Linearisations are determined by allowing concurrent (i.e., overlapping) operation calls to take effect in any order, but requiring the real-time order of non-overlapping to be preserved; thus a concurrent execution may have many possible linearisations of its operations. Certain linearisations, however, may not be valid with respect to the correct sequential specification. If none of a concurrent execution's linearisations are valid then the concurrent execution is not considered linearisable because it is not possible to get the same results as the concurrent execution for any of the linearisations (which consider all orders of overlapping operations). Formally,

Definition 1. *A history H induces an irreflexive partial order $<_H$ on operations: $e_0 <_H e_1$ if res* (e_0) *precedes inv*(e_1) *in H. A* ***history*** *H is linearisable if it can be extended (by appending zero or more response events) to some history H' such that:*
complete(H') is equivalent to some valid linearisation S, and $<_H \subseteq <_S$ [15].

Definition 2. *A* ***concurrent history*** *is linearisable if there exists at least one valid linearisation of the history.*

Definition 3. *A* ***concurrent data structure*** *is linearisable if each individual concurrent history reachable during execution of that data structure, is linearisable [15]. If there exists one or more concurrent histories that are not linearisable then by Definition 2 the concurrent data structure is not linearisable.*

Stellenbosch University https://scholar.sun.ac.za



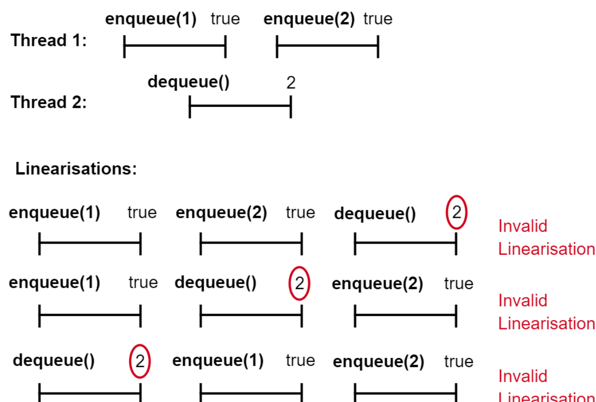


Figure 2.7: **Linearisability example with a Queue data structure.** An example of two concurrent histories for the BuggyQueue data structure where one history is linearisable and the other not. We assume a starting Queue that is empty.

Figure 2.7 shows two different example concurrent histories for the execution of a Queue data structure. Each history has three derivable linearisations, labeled either valid or invalid according to the rules of linearisability, and red ovals are used to indicate invalid response values from the linearisation operations.

The concurrent history on the left has two valid linearisations and one invalid linearisation. Thus by Definition 2, the concurrent history on the left is linearisable because at least one of the linearisations is valid. We say it is possible for thread 2's dequeue, which overlaps with thread 2's enqueue, to return 1. The concurrent history on the right has no valid linearisations. Thus by Definition 2, the concurrent history on the right is not linearisable. We say that it is not possible for thread 2's dequeue operation to return 2 according to the sequential specification of a FIFO queue, even if we consider all the possible orders in which the overlapping operations could have taken effect.

If all concurrent histories, generated by the model checker for the concurrent data structure, are proven linearisable (that is they have at least one valid linearisation as per Definition 2) then the concurrent data structure itself is by Definition 3 also linearisable.

Stellenbosch University https://scholar.sun.ac.za

2.2.4 Linearisability Checking with JPF

JPF requires as input at least the Java problem to test, called the System Under Test (SUT), as well as any input required by the program. In our application, this program is a non-blocking concurrent data structure and our tool performs linearisability checking of this SUT. It is assumed that JPF will run in depth-first-search mode.

In order for a data structure SUT to be verified/checked for linearisability it must be supplied together with a correct sequential version of the data structure SUT, also called the sequential oracle. The sequential oracle will be used during linearisability checking to determine whether a generated linearisation is valid by checking whether it corresponds to a sequential execution of the same sequence of operations as in the linearisation.

To verify that the entire data structure is linearisable it is necessary to prove that each concurrent history execution of the SUT, generated by the JPF model checker, is linearisable.

2.3 TYPES OF LINEARISABILITY CHECKING STRATEGIES

We will describe the two main linearisability checking strategies, that use model checking: 1. Linearisation Point Linearisability Checking, and 2. Automatic Linearisability Checking. Figure 2.8 shows two identical concurrent histories where the left example uses linearisation point checking to check linearisability of the history and the right example uses automatic linearisability checking.

Linearisation points are specified in the operations of a concurrent data structure; the linearisation point of an operation is considered to be the instant, between the operation's invocation and response events, at which the operation takes effect.

The Linearisation Point Checking strategy requires linearisation points to be manually specified by the user at program statements in the SUT. The linearisation points provide a means to order the operations in a concurrent history into a linearisation that exactly reflects the time ordering in which each operation of the concurrent history was executed. The left of Figure 2.8 shows a concurrent history example with linearisation points depicted as orange circles. The orange circles are placed between each operation's invocation and response and corresponds to the point in time at which the operation took place. Chronologically ordering the concurrent history operations, according to

Stellenbosch University https://scholar.sun.ac.za

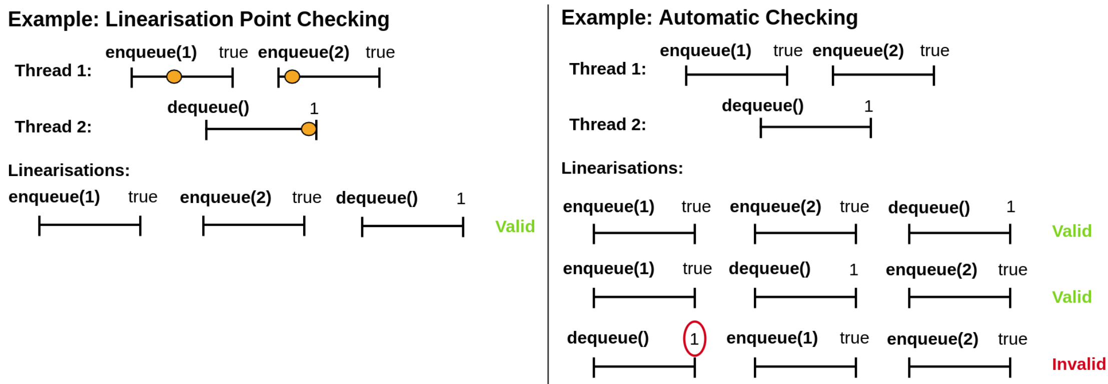


Figure 2.8: **Concurrent history and linearisations: Automatic versus Linearisation Point checking strategies**

the linearisation points, generates the one possible linearisation shown below its concurrent history. To check whether the concurrent history is linearisable the operation results should be compared to the corresponding results of an execution of the same operations by the sequential oracle. It is not necessary for the linearisability checker to wait until the entire history has been generated because the response value of each operation is available at its linearisation point. The linearisation point checker takes advantage of this and makes on-the-fly comparisons between the concurrent history's operation response values and the response values, for the corresponding operations, of the sequential oracle.

Unfortunately, linearisation points are difficult and sometimes impossible to define; the automatic checker provides a solution for checking algorithms where the linearisation points are uncertain. Automatic checking generates the linearisations for a concurrent history automatically. Instead of on-the-fly comparisons, this checker maintains records of the concurrent history along the path and then at end states it uses these records to generate all the possible linearisations of the concurrent history. It then checks each linearisation individually until a valid linearisation is found to verify the linearisability of that path's concurrent history.

The right of Figure 2.8 shows the same concurrent history as for the linearisation point checking example on the left but shows all the possible linearisations generated by the automatic checker,

Stellenbosch University https://scholar.sun.ac.za

which for this example is three. The automatic checker then checks each linearisation to determine whether it is linearisable or not. If at least one is found linearisable then by Definition 2 the concurrent history is linearisable. The design and implementation details for these and other checkers are given in Chapter 3.

Stellenbosch University https://scholar.sun.ac.za

CHAPTER 3

DESIGN AND IMPLEMENTATION

The linearisation point and the automatic linearisability checking strategies are described in detail in Sections 3.1 and 3.2, respectively. An extension to the linearisation point checking strategy in the literature, to include data structures with operations that act generically on a data structure, is presented in Section 3.1.2. Sections 3.2.1.2 and 3.2.1.1 detail two optimisation techniques for the automatic checking strategy: lazy read proposed by Long et al˙ [24], and a hash optimisation proposed in this thesis. A Symbolic Linearisability Checker that uses Symbolic PathFinder (SPF), JPF's symbolic execution extension, to integrate linearisability checking into a symbolic setting is presented in Section 3.4. Lastly, JPF's hash optimisation technique, the reasons why it causes unsoundness with respect to linearisability, and a solution for guaranteeing soundness is presented in Section 3.5.

The two core linearisability checking strategies implemented are Linearisation Point Checking and Automatic Checking:

- *Linearisation Point Checking*
  The linearisation point checking strategy requires manually specified linearisation points in the SUT at specific program statements. The linearisation points are used to determine the order of operation events and compare the response values of each operation in the concurrent execution to the response values of the correct sequential specification of the program. If each operation corresponds to the correct sequential specification then the execution is linearisable.

- *Automatic Checking*
  The Automatic checking strategy does not require manually specified points and therefore does not know the exact order of operation events. It records the trace information for a program execution and at the end of the execution, generates all the derivable linearisations of the trace. For each generated linearisation, it compares the response values of each operation in the execution to the response values of the correct sequential specification of the program. If a linearisation is found where each operation corresponds to the correct sequential specification then the execution, from which the trace was recorded, is linearisable.

Stellenbosch University https://scholar.sun.ac.za

Each of the two linearisability checking strategies have been integrated with JPF to create concrete linearisability checking tools, and the automatic checking strategy has been integrated with SPF to create a symbolic linearisability checking tool. The underlying model checkers must run in depth-first-search mode for the linearisability checkers to run correctly. Internal and external implementations are two different ways to implement the checking strategies in JPF/SPF. The linearisability checking logic can be included in the SUT by way of manual code instrumentation, called the internal implementation, or it can be excluded from it and executed alongside but externally to the SUT, called the external implementation. All of the concrete checkers were implemented for both the internal and external implementation. The symbolic checker was implemented only with the external implementation because of the external performance benefits seen in the concrete checkers; see Section 4.2 of Chapter 4 for performance details. Table 3.1 shows the high-level concrete and symbolic checkers with their corresponding details of internal/external implementations and linearisability comparison strategies.

		Linearisability checking takes place at:	**Internal Imp.**	**External Imp.**
Concrete	**Automatic**	linearisations generated and comparisons made at end states	yes	yes
	Linearisation Point	comparisons made on-the-fly at linearisatoin points	yes	yes
Symbolic	**Automatic**	linearisations generated and comparisons made at end states	no	yes

Table 3.1: **Defining the types of linearisability checkers.** Implementation categories for the two concrete and one symbolic linearisability checker types.

3.1 LINEARISATION POINT CHECKING

Linearisation points are specified in the operations of a concurrent data structure; the linearisation point of an operation is considered to be the instant, between the operation's invocation and response events, at which the operation takes effect. In situations where the exact instant at which an operation comes into effect cannot be specified, a non-fixed linearisation point will be used for that operation during checking.

Informally, fixed and non-fixed linearisation points can be specified as follows [39]: 1. A fixed linearisation point identifies the exact instant at which an operation comes into effect, between

Stellenbosch University https://scholar.sun.ac.za

its invocation and response events. 2. In situations where this exact instant is not evident then non-fixed linearisation points can be used to define the segment or segments of time, between the operation's invocation and response events, for which the operation would return a correct response value if a fixed linearisation point was placed there.

The linearisation point checking strategy performs on-the-fly comparisons between the response values of concurrent history trace operations and the response values of the sequential specification's corresponding operations. The comparisons take place each time a linearisation point is encountered, during the model checker's generation of the trace execution. If the response values of the trace do not correspond exactly to those of the correct sequential specification then by definition 2 the history is not linearisable. If there is at least one trace execution of the SUT which is not linearisable then the SUT is not linearisable by Definition 3. The high-level linearisation point checking process is shown in Figure 3.1.

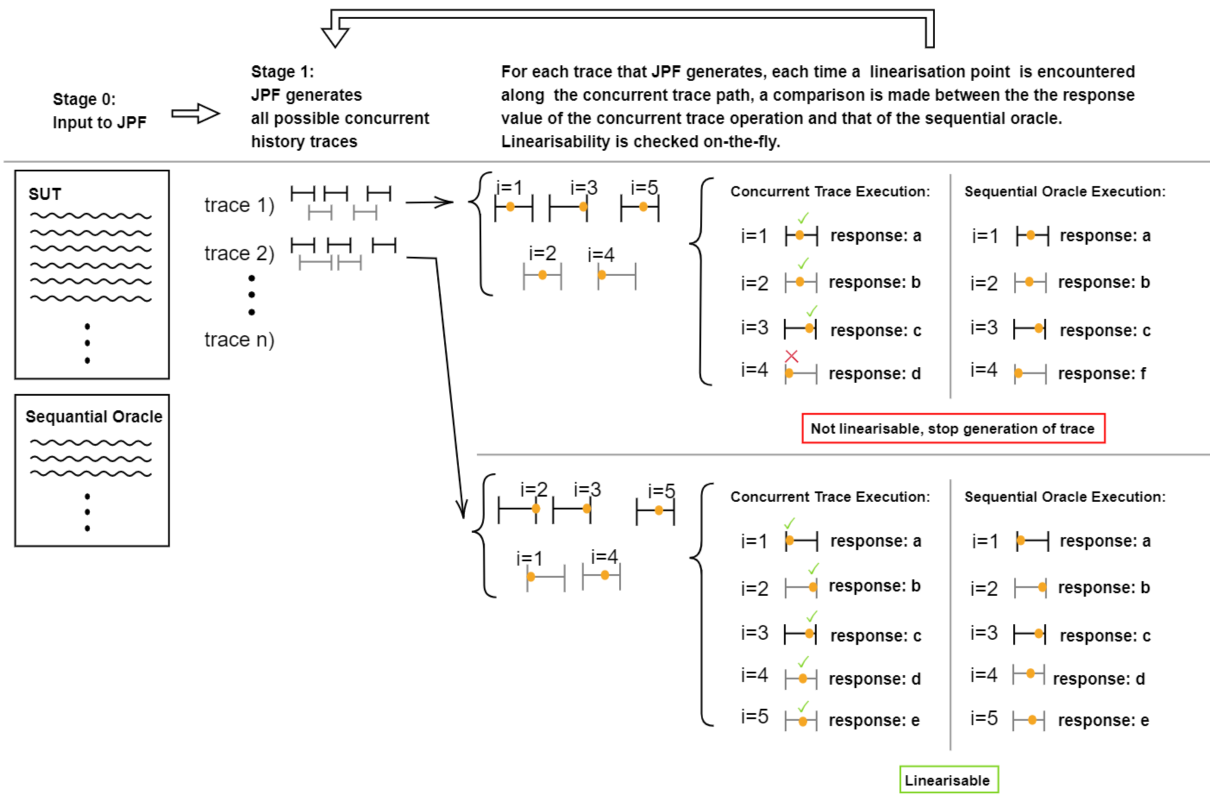


Figure 3.1: **On-the-fly Linearisation Point Checking.** A visual illustration of the high level linearisability checking process for the linearisation point checking strategy.

Stellenbosch University https://scholar.sun.ac.za

JPF takes as input the SUT and the corresponding correct sequential specification (Stage 0). JPF generates all possible execution traces for the input, during the generation of each trace the linearisability checker performs on-the-fly linearisability checking comparisons (Stage 1): each time a linearisation point is encountered the sequential oracle is executed with the equivalent operation and the response values compared. For the trace to be considered linearisable each comparison must match the trace operation's response with that of the sequential oracle; if even a single comparison fails then the trace is not linearisable and the checking process is stopped. In Figure 3.1 the generation of trace 1 is stopped when linearisation point 4 is reached and its response value does not match the response value of the sequential oracle. The history is not linearisable, and it is not necessary to perform the check for any later linearisation points in the history. The history of trace 2 is linearisable since each linearisation point in the history corresponds to a response value equal to that of the sequential oracle. The time and space complexity of the one-the-fly comparisons is linear in the length of the concurrent history.

3.1.1 Fixed and Non-fixed Linearisation Points

The linearisation point checking strategy is made up of two different types of linearisation point logic, called fixed and non-fixed linearisation points. These together make up the linearisation point checking strategy implemented for results in this thesis.

3.1.1.1 Fixed linearisation points

A fixed linearisation point identifies the exact instant at which an operation comes into effect, between its invocation and response events. Fixed linearisation points provide a means to order the operations in a concurrent history into a linearisation that perfectly reflects the time ordering in which each operation of the concurrent history was executed. Fixed linearisation points can only be determined for operations where there is an exact instant at which the operation comes into effect, that is operations with one write operation that acts on the data structure.

For example, a Set data structure has operations `add`, `remove`, and `contains`. When the `add` or `remove` operations return true, a fixed linearisation point corresponds to the exact instant of the write operation on the Set data structure. Figure 3.2 shows a concurrent history trace for the Set data structure where all operations in the trace have a fixed linearisation point.

Stellenbosch University https://scholar.sun.ac.za

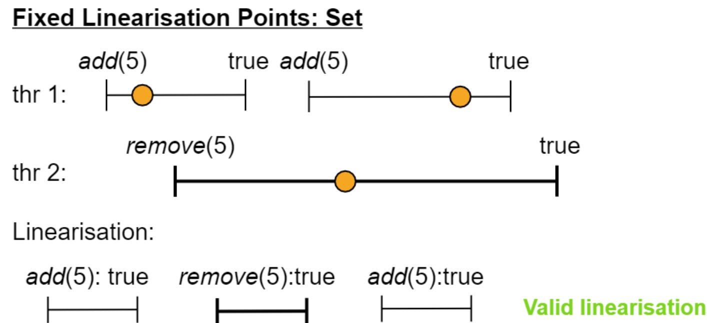


Figure 3.2: **Set data structure: Fixed Linearisation Point Example.** Illustration of a concurrent history where each operations in the history contains a fixed linearisation point, and the corresponding linearisation.

The fixed linearisation points allow the ordering of the concurrent history operations into a linearisation that perfectly reflects the time ordering in which each operation was executed. The linearisation, as shown at the bottom of the figure and we see that each operation in the linearisation corresponds to that of a correct sequential execution; so it is linearisable.

When the `add` or `remove` operation of the Set data structure returns false or the `contains` operation returns true/false, the data structure is not altered and thus an exact instant at which the operation comes into effect cannot be specified; this situation generalises to other data structures and operation types.

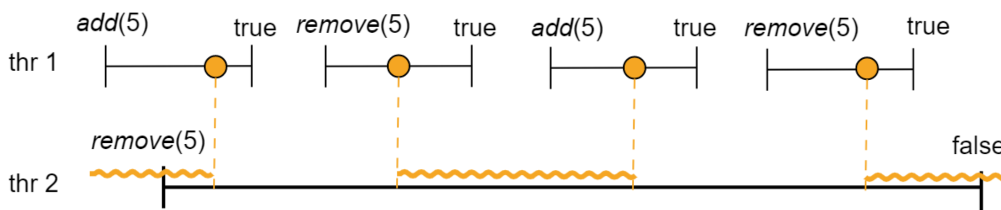
3.1.1.2 Non-fixed linearisation points

In situations where this exact instant is not evident, non-fixed linearisation points can be used to determine the segment or segments of time, between the operation's invocation and response events, for which the effect of the operation would produce the response value that was returned by the operation.

A non-fixed linearisation point, for some operation, uses information from other threads to determine which segments of time, between the operation's invocation and response, the operation would return the response of the executed operation. The markers used to identify the start or end of a non-fixed linearisation point segment are the fixed linearisation points in overlapping operations of other threads; where the overlapping operations are acting on the same data structure element as

Stellenbosch University https://scholar.sun.ac.za

the current operation.



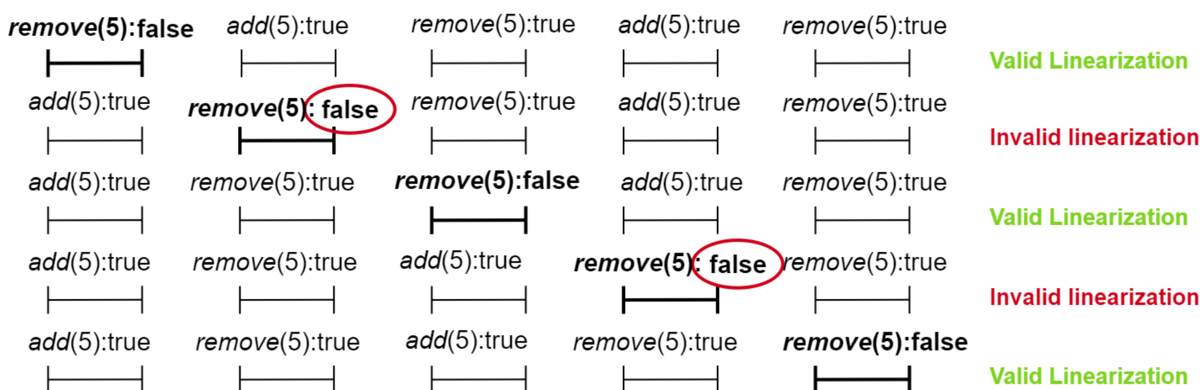


Figure 3.3: **Non-Fixed Linearisation Points (key-specific example).** A visual illustration of a concurrent history for a Set data structure where thread one executes four operations, each with a fixed linearisation point, and thread two executes one operation with non-fixed linearisation point segments. The three valid and two invalid linearisations for this concurrent history are shown below the concurrent history trace.

Figure 3.3 shows how the fixed linearisation points of operations in other threads can be used to identify the non-fixed linearisation points in the current thread's operation. In the figure:

1. The concurrent history shows that thread-1 executed four operations, each with a fixed linearisation point identified by the **filled orange circles**. Thread-2 executes an unsuccessful `remove` operation that does not contain a fixed linearisation point, and overlaps with all four operations in thread-1.

2. Each of the operations act on the element in the data structure with the **value of five** i.e. all the operations are called with argument five. We assume the values for each element in the data structure are unique.

3. The **vertical dashed lines** identify the start and end of non-fixed linearisation point segments

Stellenbosch University https://scholar.sun.ac.za

of the thread-2 operation. The start and end markers are identified by considering the four fixed linearisation points in thread one.

4. The **wavy orange horizontal lines** just above thread two's operation trace and are the non-fixed linearisation point segments.

To determine the non-fixed linearisation point segments, the effect of each overlapping fixed linearisation point must be considered; providing a means to determine in which segment of time the operation would return its response value.

We know that for a Set, after a successful `add` of a value, a `remove` of that value should be successful, and that after a successful `remove` of a value, a `remove` of that value should be unsuccessful. Therefore we can infer that the `remove` operation of thread-2 will return *false* after a successful `remove` and before a successful `add` operation by thread-1. Thus the non-fixed linearisation point for thread-2's operation is determined to fall in three segments: 1. before the first fixed linearisation point of thread-1, 2. after the second but before the third linearisation points of thread-1, and 3. after the fourth linearisation point of thread-1. For completeness, the five derivable linearisations of the concurrent history of Figure 3.3 are shown below the history trace. The five linearisations show the five possible interleavings of thread-2's operation between the ordered fixed-linearisation point operations of thread-1.

The linearisations which correspond to thread-2's operation interleaved in non-fixed linearisation point segments (first, third, and fourth) are shown as valid linearisations, but those corresponding to interleaving positions not in non-fixed linearisation point segments (second and third) are invalid linearisations. The invalid linearisations fail because thread-2's remove operation returns *false* which does not correspond to the sequential oracle for that interleaving position. This illustrates the non-fixed linearisation point's correctness in identifying a segment in time for which, if the operation had to come into effect, the history would be linearisable. See Appendix A.3 for the non-fixed linearisation point diagrams for an unsuccessful `add`, an unsuccessful `contains`, and a successful `contains` operation.

Linearisation point checking, when both fixed and non-fixed linearisation points are present, uses the following method:

- Assumptions:

Stellenbosch University https://scholar.sun.ac.za

1. A key may be of any data type

2. The data structure initially contains no elements

3. Key values within the data structure are unique

4. A key is known to be contained in the data structure for the segment of time after a successful operation that adds the key to the data structure and before a successful operation that removes the key from the data structure.

5. A key is known to be absent from the data structure for the segment of time after a successful operation that removes the key from the data structure and before a successful operation that adds the key to the data structure.

- When a fixed linearisation point is encountered during execution of an operation, the equivalent operation should be executed on the sequential oracle and the response values of the two operations compared.

- When an operation's response is encountered but no fixed linearisation point has occurred for that operation, the fixed linearisation points of overlapping operations in other threads should be considered; only those overlapping operation's with fixed linearisation points that act on the same key value as the current thread's operation should be included.

  If there is at least one non-fixed linearisation point segment for the operation, between the operation's invocation and response events, then there exists a linearisation interleaving for this operation which yields a valid linearisation. Thus, provided that the concurrent history up until this operation was proven linearizable, then the concurrent history remains linearizable.

  If there does not exist a non-fixed linearisation point segment for the operation then there does not exist an interleaving which yields a valid linearisation and thus the history is not linearisable; by extension then the data structure is not linearisable (Definition 3).

3.1.2 Data structures with generic operations

The linearisation point checking strategy described up until this point is that proposed by Vechev et al. for their Set data structure; their strategy generalises to data structures with operations that act on a specific value in the data structure [39]. We extend their strategy to include data

Stellenbosch University https://scholar.sun.ac.za

structures with generic operations. Generic operations do not have arguments, they do not specify the exact value in the data structure on which they will act. An example of such a data structure is a Queue, the Queue has a `dequeue` operation which does not define the value on which it acts but instead dequeues whichever value is at the start of the queue.

3.1.2.1 Generic operations: Fixed linearisation points

Generic operations that contain fixed linearisation points can be handled in the same way as the operations that act on specific values, because the operations take effect at the linearisation points and as soon as an operation takes effect, the specific value is known. Figure 3.4 shows an example of a concurrent history which executes a generic element operation with a fixed linearisation point. The fixed linearisation points still orders the operations of the concurrent history into a linearisation that perfectly reflects the time ordering in which each operation of the concurrent history was executed.

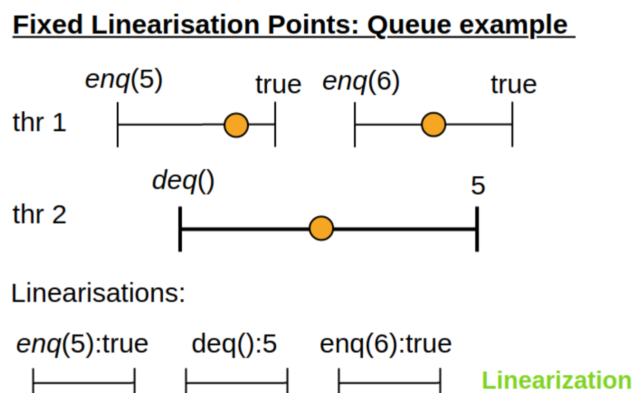


Figure 3.4: **Queue data structure: Fixed Linearisation Points.** An example concurrent history, for a Queue data structure, containing operations that each execute a successful write operation and thus each contain a fixed linearisation point, and the linearisation corresponding to the fixed linearisation point ordering in the concurrent history.

3.1.2.2 Generic operations: Non-Fixed linearisation point segments

For generic operations that do not contain fixed linearisation points, non-fixed linearisation point segments cannot be determined by considering the overlapping operations with fixed linearisation points, pertaining to only one specific value; all overlapping operations with fixed linearisation

Stellenbosch University https://scholar.sun.ac.za

points regardless of value, should be considered.

Let LP be the set of linearisation points that should be considered and let S be the set of all keys contained in the data structure during the time between the operation's invocation and response events, then

1. If there are operations with fixed linearisation points in other threads that overlap with the current operation on a value $v \in S$, only those overlapping fixed linearisation points should be added to LP.

2. If there are no operations with fixed linearisation points that overlap with the current operation on $v \in S$ then the last fixed linearisation point for an operation on value $v \in S$ should be added to LP.

Using LP, the non-fixed linearisation point segments of the operation can be specified: If there is at least one instant in time, between the operation's invocation and response, where non-fixed linearisation point segments for all values in S overlap then, provided that the concurrent history up until this operation was proven linearisable, then the concurrent history remains linearisable.

Figures 3.5 and 3.6 show a linearisable and a non-linearisable concurrent history, respectively. All thread-1 operations contain fixed linearisation points and all thread-2 operations contain non-fixed linearisation points. The following elements are used in the figure:

1. The concurrent history in the figures show that thread-1 executed both specific and generic operations that each contain a fixed linearisation point; identified by the **filled circles**. The `enqueue` operations are specific and the `dequeue` operations are generic.

   Thread-2 executed a generic operation, `dequeue`, which does not contain a fixed linearisation point and has the response value of *empty*. The last 4/3 operations of thread one overlap with the operation of thread two, respectively for Figures 3.5 and 3.6.

2. The **vertical dashed lines** identify the start and end of non-fixed linearisation point segments for thread two's operation, they are identified by considering the fixed linearisation points in the operations of thread one.

3. The **wavy orange, purple, and green horizontal lines** just above thread two's operation trace each correspond to a different value and are the non-fixed linearisation point segments

Stellenbosch University https://scholar.sun.ac.za

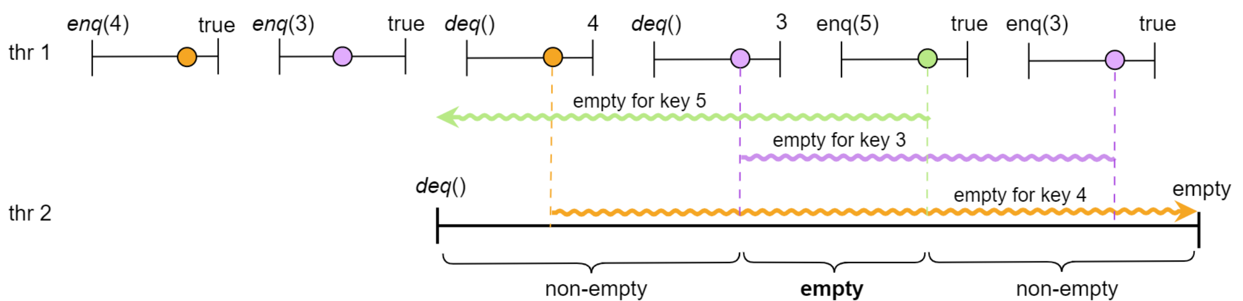
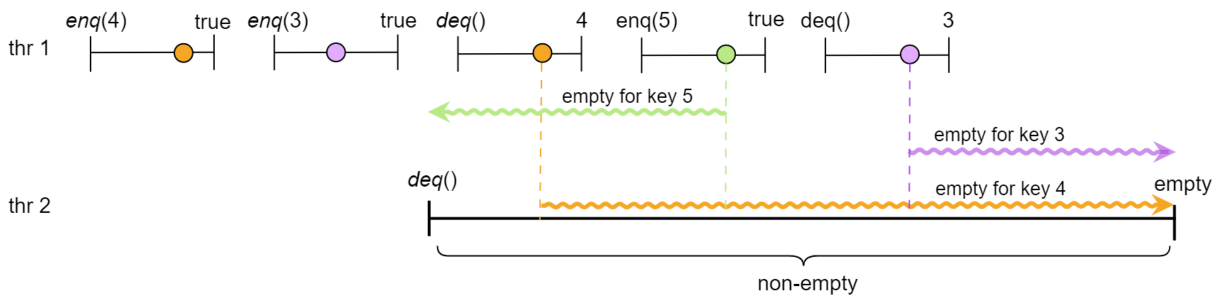


Figure 3.5: **Non-Fixed Linearisation Points (linearisable key-generic example).** A visual illustration of a linearisable concurrent history for a Queue data structure where thread one executes four operations, each with a fixed linearisation point, and thread two executes one operation with non-fixed linearisation point segments. The one valid and four invalid linearisations for this concurrent history are shown below the concurrent history trace.

Stellenbosch University https://scholar.sun.ac.za

Non-Fixed Linearisation Points: Non-linearisable Queue Concurrent History with unsuccessful dequeue



Linearisations:

enq(4):true *enq*(3):true ***deq*(): empty** deq():4 enq(5):true deq():3 — Invalid Linearisation

enq(4):true *enq*(3):true deq():4 ***deq*(): empty** enq(5):true deq():3 — Invalid Linearisation

enq(4):true *enq*(3):true deq():4 enq(5):true ***deq*(): empty** deq():3 — Invalid Linearisation

enq(4):true *enq*(3):true deq():4 enq(5):true deq():3 ***deq*(): empty** — Invalid Linearisation

Figure 3.6: **Non-Fixed Linearisation Points (non-linearisable key-generic example).** A visual illustration of a non-linearisable concurrent history for a Queue data structure where thread one executes four operations, each with a fixed linearisation point, and thread two executes one operation with non-fixed linearisation point segments. The four invalid linearisations for this concurrent history are shown below the concurrent history trace.

Stellenbosch University https://scholar.sun.ac.za

for those respective values.

In order to determine the non-fixed linearisation point segments, the effect of each fixed linearisation point in LP must be considered; providing a means to determine in which segment of time the operation would return its response value.

We assume unique values for all elements in the Queue, for reasons discussed in Section 3.5.3. Directly after a successful `dequeue` operation the Queue will not contain the value removed from the Queue and directly after a successful `enqueue` operation the Queue will contain the key value. Thus the non-fixed linearisation point segments of thread-2's operation either begin after a successful `dequeue`, by thread-1 on the specific value, or after a successful `enqueue`, by thread-1 on the specific value. For thread-2's `dequeue` operation to be linearisable there must be a segment of time, between the operation's invocation and response events, where the non-fixed linearisation segments overlap for every key in S; if there does not exist a segment of time where this is true then the `dequeue` operation's response value is not linearisable.

The example in Figure 3.5 shows a situation where the non-fixed linearisation point segments, for each value in S, overlap (between the fourth and fifth fixed linearisation points of thread-1); thus thread-2's operation, when interleaved between the fourth and fifth fixed linearisation points of thread-1, corresponds to a valid linearisation. The five possible interleavings, for which only the interleaving just mentioned provides a valid linearisation, are shown below the concurrent history.

For the example in Figure 3.6, however, there does not exist a segment of time between thread two's operation invocation and response events where the non-fixed linearisation segments for each value in S overlap; thus the history is not linearisable. The four linearisations shown below the concurrent history of this figure show that for all possible interleavings of thread-2's operation into a linearisation, none produce a valid linearisation.

Thus if there is at least one instant in time, between the operation's invocation and response, where non-fixed linearisation point segments for all values in S overlap then, provided that the concurrent history up until this operation was proven linearizable, then the concurrent history remains linearizable.

Stellenbosch University https://scholar.sun.ac.za

3.1.2.3 Generic operations: linearisability checking procedure

Linearisation point checking of data structures with generic operations, when both fixed and non-fixed linearisation points are present, uses the following method:

- The assumptions from Section 3.1.1.2 also apply here.
- When a fixed linearisation point is encountered for an executing operation, then the equivalent operation should be executed on the sequential oracle and the response values of the two operations compared.

  Let LP be the set of linearisation points that should be considered and let S be the set of all keys contained in the data structure during the time between the operation's invocation and response events, then

  1. If there are operations with fixed linearisation points in other threads that overlap with the current operation on a value $v \in S$, only those overlapping fixed linearisation points should be added to LP.
  2. If there are no operations with fixed linearisation points that overlap with the current operation on $v \in S$ then the last fixed linearisation point for an operation on value $v \in S$ should be added to LP.

- If an operation's response is encountered but no fixed linearisation point has occurred for that operation, then the fixed linearisation points in LP should be considered.

  If there is at least one segment of time, between the operation's invocation and response, where the non-fixed linearisation points for all values in S overlap, then there exists a linearisation interleaving for this operation which yields a valid linearisation; provided that the concurrent history up until this operation was proven linearizable, then the concurrent history remains linearizable.

  If there does not exist a non-fixed linearisation point segment for which this is true then there does not exist an interleaving which yields a valid linearisation and thus the history is not linearisable; by extension then the data structure is not linearisable (Definition 3).

Stellenbosch University https://scholar.sun.ac.za

3.1.3 Conclusions for the linearisation point strategy

We have discussed the linearisation point checking strategy, the first of the two strategies implemented in this thesis: Linearisation Point Checking, and Automatic Checking, and investigated our extension of this strategy to include data structures with generic operations.

The defining of linearisation points for on-the-fly linearisability checking requires the user to have an in-depth understanding of the SUT data structure in order for the manually specified linearisation points to be determined correctly.

In the next section we investigate the automatic linearisability checking strategy and its ability to check linearisability without specified linearisation points, and describe two optimisations to it.

3.2 AUTOMATIC CHECKING

The automatic strategy does not require user-specified linearisation points, like the linearisation point strategy does, but automatically generates all possible sequential orderings of the concurrent history to determine its linearisability. The automatic strategy collects and stores trace information along each generated concurrent history path, and at each path end state is uses the stored trace information to generate all possible linearisations for the respective concurrent history. Each linearisation is checked for linearisability and if at least one linearisation of the history is valid then the history is linearisable; if no linearisations are valid then the history is not linearisable. The high-level automatic checking process is shown in Figure 3.7. In the figure:

1. **Stage 0:** JPF takes as input the system under test (SUT) and correct sequential specification (sequential oracle).

2. **Stage 1:** JPF generates all concurrent history executions for the SUT data structure.

   For each of the concurrent histories generated, the linearisability checker collects and stores the concurrent history trace information up until JPF's end state for that concurrent history path.

   - **Stage 1.1:** For each concurrent history generated, at the end state of that history's path the stored trace information is used to generate all possible linearisations of the history.

Stellenbosch University https://scholar.sun.ac.za

- **Stage 1.2:** Each linearisation is compared to the sequential oracle and checked as valid or invalid. For the first valid linearisation found, the concurrent history for that path is proven linearisable and the checking for that path stops. If no valid linearisation is found then history along that path is not linearisable.

  Stage 1 is then executed for the next generated concurrent history until all of the possible concurrent histories have been generated and checked.

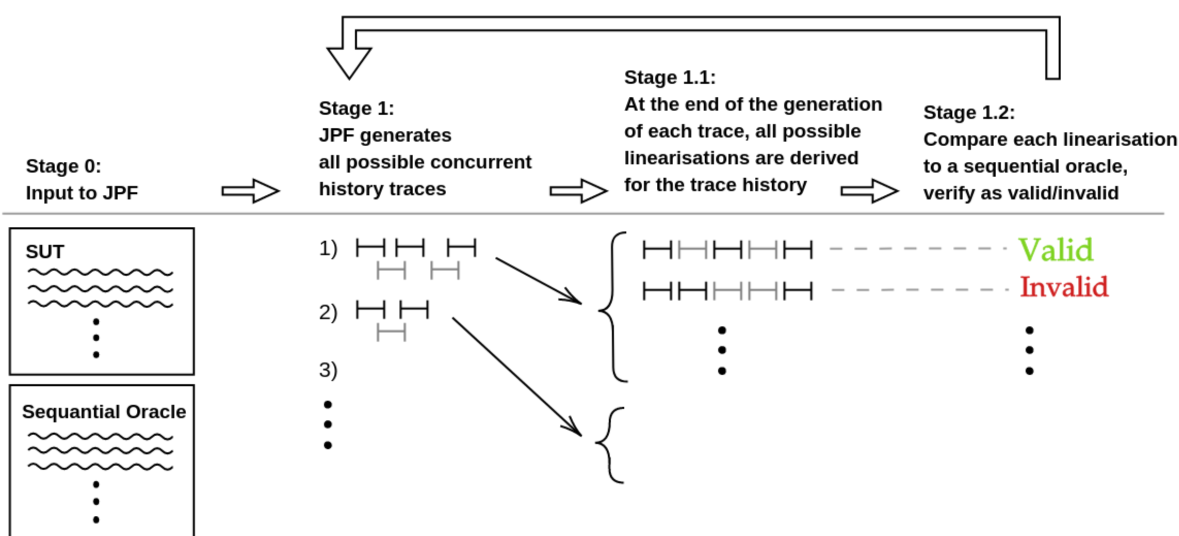


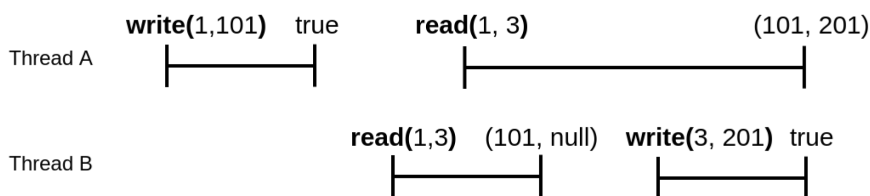
Figure 3.7: **Automatic Linearisability Checking Process.** A visual illustration of the high level linearisability checking process for automatic vanilla linearisability checking tools. 0. JPF takes as input the system under test (SUT) and correct sequential specification (sequential oracle), 1. JPF generates all possible concurrent history executions, 1.1. For each of these concurrent histories, all the linearisations derivable from the concurrent history are generated and 1.2. Each of these linearisations are tested to determine whether they are linearizations. Stage one is then redone for the next generated concurrent history.

Figure 3.8 shows an example concurrent history and the three possible linearisations for that history. Each linearisation is labelled as either an "Invalid" or "Valid" linearisation according to its result compared to the correct sequential specification.

The worst-case time and space complexity for the automatic checking of a history trace is exponential in the length of the concurrent history since, for each additional operation in the concurrent history, there is an exponential increase in the number of generated linearisations. In the next section we will describe two different optimisation techniques which aim to reduce the amount of

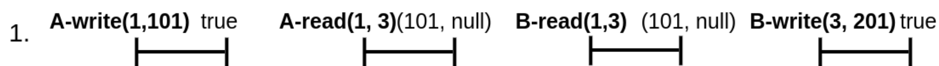
Stellenbosch University https://scholar.sun.ac.za

Concurrent History:

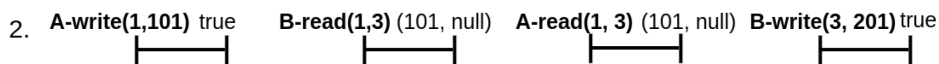


3 Linearisations, 1 of which is a valid

Invalid linearisation:



Invalid linearisation:



Valid linearisation:

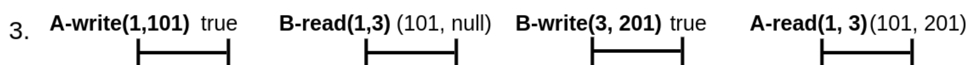


Figure 3.8: **Automatic Checking Process Example:** An example linearisable concurrent history with operations *read(param1, param2)* and *write(param1, param2)*. The two operation parameters for write is 1. the value to write and 2. the memory location at which to execute the write operation; the response of the write operation is the boolean value true if the write was successful and false if it was not. The two operation parameters for the read are the two values to read; the response returns a tuple of the 2 corresponding memory locations at which the values were found. The checking process generates the three linearisations for the history, one of which is a valid linearisation.

Stellenbosch University https://scholar.sun.ac.za

computation necessary for the automatic linearisability checking strategy.

3.2.1 Optimisations

Two optimisations were implemented: lazy read and a hashing optimisation. Lazy read was proposed by Long et alto reduce the number of linearisations generated in checking of each given concurrent history (Section 3.2.1.1) [24]. The hashing optimisation is proposed here to only check linearisability of concurrent histories that have not been checked yet. It is based on the fact that JPF generates many executions that reduce down to the same concurrent history trace.

3.2.1.1 Lazy Read optimisation

The lazy read optimisation aims to reduce the number of linearisations generated during the automatic checking of a given concurrent history. It is based on the fact that only those operations that execute successful write instructions cause changes to the concurrent data structure, and thus it is only necessary to generate linearisations for the enumeration of these write-containing operations. The read operations can then be included at all possible positions for these write-enumerations to achieve the same linearisation checking as the ordinary automatic strategy but with fewer generated linearisations, which should alleviate the exponential worst-case time and space complexity of the automatic checking process. The optimisation changes the way that linearisations are generated from a given concurrent history, outlined below:

1. As for the ordinary automatic checking strategy, the lazy read optimisation generates the linearisations at path end states. The optimisation uses the stored concurrent history trace information to generate linearisations using only those operations in the trace which contain successful write instructions. The resultant enumerations are called the base linearisations.

2. For each base linearisation the read operations, i.e. those that during execution did not alter the state of the data structure, are interleaved at all possible execution positions that satisfy the happens-before relation of overlapping concurrent operations from the history trace. This allows more than one instance of the same read operation to be added to a *base linearisation* and the order of consecutive read operations; allowable because the reads do not alter the state of the data structure so multiple instances in a linearisation do not affect the response results of other operations. The base witnesses with the included read operations are termed

Stellenbosch University https://scholar.sun.ac.za

extended linearisations.

3. Each extended linearisation is then checked to see whether they contain a linearisation. To contain a linearisation, each write-containing operation response must correlate to that of the sequential oracle and there must be at least one instance of each read operation where its response correlates to that of the sequential oracle. Thus the extended linearisation represents many ordinary linearisations and if one of the ordinary linearisations within it are valid then the extended witness is valid.

Figure 3.9 shows the concurrent history example for the ordinary automatic checker, the three linearisations generated by the ordinary automatic checking strategy, and then the base and extended linearisations generated by the lazy read optimisation for the automatic checking strategy. Lazy read, for example, is shown as effective in reducing what would, for ordinary automatic checking, be three generated linearisations to one extended linearisation that represents all three of the ordinary linearisations.

The lazy read optimisation will show more benefit for test cases with fewer write operations because the number of linearisations is determined by the number of write operation interleavings; unlike the un-optimised automatic strategy which creates linearisations for the number of read and write operation interleavings.

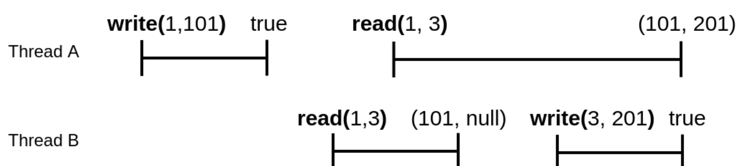
3.2.1.2 Hash optimisation

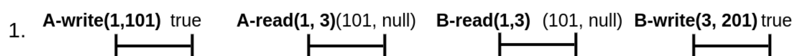
The model checker generates all possible bytecode interleavings for the execution of the SUT, but multiple bytecode interleavings reduce to the same concurrent history trace. This optimisation hashes the history trace information on-the-fly so that when it reaches an end state and determines that the history has been re-encountered, it does not check linearisability for that path again.

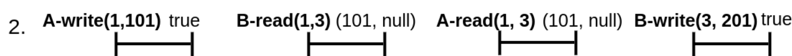
The **hash function** takes as input the operation invocation/response event and it's executing thread information. It returns a byte array that contains the bit-format hash of the input event. Thus each unique history is represented as a unique sequence of hashed invocation/response events which make up the concurrent history trace.

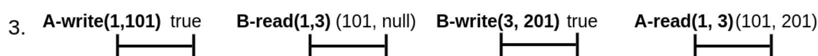
A special kind of **hashtable** is maintained in the listener where operations are incrementally hashed, as execution through the concurrent history progresses. The hashtable form an abstracted

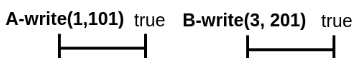
Stellenbosch University https://scholar.sun.ac.za











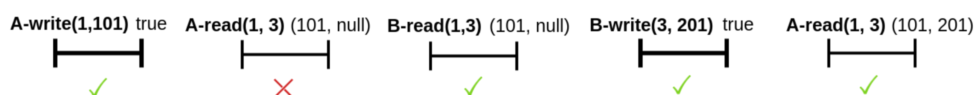


Figure 3.9: **Automatic checking strategy with the lazy-read optimisation example.** The two operation parameters for write is 1. the value to write and 2. the memory location at which to execute the write operation; the response of the write operation is the boolean value true if the write was successful and false if it was not. The two operation parameters for the read are the two values to read; the response returns a tuple of the 2 corresponding memory locations at which the values were found. The figure shows the three linearisations generated by ordinary automatic checking and it shows the base linearisation and extended linearisation of the lazy-read process.

Stellenbosch University https://scholar.sun.ac.za

tree structure where states denote positions in the concurrent history and concurrent history event hashes denote the edges connecting any one parent state to its many possible children states; the hash tree corresponds to a minimised abstraction of JPF's execution tree.

For JPF backtracking operations that revert previously hashed concurrent history operation events, the position in the hash tree is backtracked accordingly to align with JPF's execution; in this way the hash tree shadows JPF's execution. Each time a new concurrent history event is executed by JPF, provided that the position in the hash tree already corresponds to those operation events already executed in the concurrent history, the next increment, child state, of the hashtable is updated with the newly explored event included as the event leading to the child state.

Thus the hash optimisation guarantees that each unique concurrent history traversed during JPF's execution will only be checked for linearisability once, irrespective of JPF generating multiple bytecode interleaving paths that reduce to the same concurrent history trace.

3.2.2 Conclusions for the automatic strategy

Both the linearisation point and the automatic linearisability checking strategies have now been described. Details of their integration into JPF and the two possible implementation approaches are provided in the next section.

3.3 IMPLEMENTATION OF THE CONCRETE CHECKERS

Linearisability checking can be implemented either internally or externaally to the SUT. The internal implementation requires manual code instrumentation to be added to the SUT, the instrumentation includes all linearisability checking logic into the SUT so that it executes along with JPF's execution of the SUT; all linearisability checking state information is part of the state space. The external implementation does not require manual code instrumentation but makes use of JPF's listener API to perform all linearisability checking logic alongside, but external to JPF's execution of the SUT. The listener is able to receive on-the-fly notifications about the model checker's search, which the external implementation uses to gain access to the concurrent history trace information and perform linearisability checking. In this case the linearisability checking state information is recorded separately from the state space.

Stellenbosch University https://scholar.sun.ac.za

3.3.1 Internal Implementation

The internal implementations of the checking strategies include all of the respective linearisability checking logic, including sequential oracle execution, in the SUT via manual code instrumentation. The model checker generates all possible concurrent history executions of the SUT, and the linearisability checking thus takes place along with the execution of each of the generated histories since the histories contain the checking logic.

An example of manual code instrumentation

We give an example of manual code instrumentation by way of fixed linearisation point placement in the SUT. The code fragments below shows the `enqueue` operation of the BuggyQueue algorithm of Section 2.2.1 but with linearisation point code instrumentation; the code instrumentation lines are shown in blue on lines 12, 13, 15, 16, 19, and 22.

Listing 3.1: **Non-blocking BuggyQueue enqueue operation using atomic CAS operations:**

```
1.  public boolean enqueue(int item) {
2.    int rear;
3.    Node x;
4.    boolean resultFound = false;
5.    do {
6.      do {
7.        rear = REAR.get();
8.        x = (Node) Q_atomic.get(rear%L);
9.      } while (rear != REAR.get()  rear == FRONT.get()+L);
10.     if (x.isIntNull) {
11.       // CAS operations
12.       if (Q_atomic.compareAndSet(rear%L, x,
              new Node(item, x.counter+1, false))) {
13.         REAR.compareAndSet(rear, rear+1);
14.         resultFound = true;
15.       }
16.     } else {
17.       REAR.compareAndSet(rear, rear+1);
18.     }
```

Stellenbosch University https://scholar.sun.ac.za

```
19.     } while (!resultFound);
20.     return true;
21. }
```

Listing 3.2: **Non-blocking BuggyQueue enqueue operation using atomic CAS operations and linearisation point instrumentation added in blue code (lines 12, 13, 15, 16, 19, and 22)**:

```
1.  public boolean enqueue(int item) {
2.      int rear;
3.      Node x;
4.      boolean resultFound = false;
5.      do {
6.         do {
7.            rear = REAR.get();
8.            x = (Node) Q_atomic.get(rear%L);
9.         } while (rear != REAR.get()   rear == FRONT.get()+L);
10.        if (x.isIntNull) {
11.           // CAS operations
12.           boolean updated = false;
13.           synchronized(lock)
14.              if (Q_atomic.compareAndSet(
                    rear%L, x, new Node(item, x.counter+1, false))) {
15.                 linPoint(item);
16.                 updated = true;
17.              }
18.           }
19.           if (updated) {
20.              REAR.compareAndSet(rear, rear+1);
21.              resultFound = true;
22.           }
23.        } else {
24.           REAR.compareAndSet(rear, rear+1);
28.        }
```

Stellenbosch University https://scholar.sun.ac.za

```
29.     } while (!resultFound);
30.     return true;
31. }
```

The code instrumentation in this example executes along with the SUT's `enqueue` operation. Each time the instrumented linearisation point method line is executed, the linearisability checking logic performs the respective linearisability comparisons. Manual code instrumentation for other purposes is included in the SUT similarly to this example.

Notice that the linearisation point instrumentation uses a synchronized section to encapsulate the CAS operation and the linearisation point together, the reason for this is that the linearisation point should be associated with the exact instant at which the CAS operation occurs; if the synchronized section is not used other thread's bytecode instructions could execute between the CAS and the linearisation point operations. The use of this synchronized section does not not compromise the non-blocking character of the algorithm because it does not effect any of the code logic.

3.3.1.1 Internal Implementation of the Linearisation Point Strategy

The linearisation point code instrumentation is used, throughout the model checker's traversal, to perform on-the-fly linearisability comparisons to the sequential oracle. In order for the user to correctly place the linearisation points and other linearisability checking logic, the user is required to have an in-depth understanding of the SUT algorithm as well as the linearisation point strategy. The user should implement instrumentation for the following situations:

- At the *fixed linearisation point* of a SUT operation, the equivalent operation is to be executed on the sequential oracle and the two response values compared to verify the linearisability of the operation. All other executing threads should be notified of the fixed linearisation point occurrence, the operation type and the data-structure value for which it occurred. These notifications allow the other threads to determining non-fixed linearisation points in their own operations.
- For operations which reach the end of their execution but for which no fixed linearisation point was encountered, all *non-fixed linearisation point segments* for the operation must be determined by considering the fixed linearisation point notifications from other threads. The

Stellenbosch University https://scholar.sun.ac.za

non-fixed linearisation points should then be used to verify the linearisability of the data structure according to the process described in Sections 3.1.1.2 and 3.1.2.2.

3.3.1.2 Internal Implementation of the Automatic Strategy

Specifically for the automatic checking strategy, code instrumentation should gather concurrent history trace information, generate the possible linearisations at path end states, and then check each linearisation to determine the linearisability of the history trace.

The code instrumentation includes the following logic:

- Operation invocation and response event information is kept in a data structure along each path so that at program end states, the trace information can be used for linearisability checking
- At the end of the SUT, the linearisation generation and checking logic is included so that it will execute only at the end of the program execution. To ensure this a Thread.join() operation is used at the end of the main method to ensure all executing threads complete, the automatic linearisability checking logic is placed directly after the Tread.join().

3.3.2 External Implementation

The external implementation does not require manual code instrumentation but uses of JPF's listener API to perform all linearisability checking logic alongside, but external to JPF's execution of the SUT.

The listener API includes methods that can be called to receive on-the-fly notifications about the model checker's search and perform operations which alter the model checker's traversal of the search space. The external implementation uses the listener API to gain access to the model checker's search information and thus the data structure operation invocation and response events(i.e. the concurrent history trace information). When a linearisation error is encountered, the listener API allows interaction with the model checker to halt model checking execution, if configured to do so. We will now describe the external checker's use of these API calls for the linearisation point and the automatic checking strategies.

Stellenbosch University https://scholar.sun.ac.za

3.3.2.1 External Implementation of the Linearisation Point Strategy

The following API calls are used for the external linearisation point checker implementation:

1. `void methodEntered(VM vm, ThreadInfo currentThread, MethodInfo enteredMethod)`. This method executes in the listener each time JPF executes an operation invocation event, the time-ordered invocation information of each executed operation is used to keep a record of all the operation invocations pertaining to the concurrent history trace.

   We implement fixed linearisation points as method calls so that each time an invocation event happens for this linearisation point method, the listener executes the relevant linearisability checking logic. This logic includes the execution of the sequential oracle from the listener and the comparison of the operation response values.

2. `void methodExited(VM vm, ThreadInfo currentThread, MethodInfo exitedMethod)`. This method executes in the listener each time JPF executes an operation response event, the time-ordered response information of each executed operation is used to keep a record of all the operation invocations pertaining to the concurrent history trace.

   In the situation where a response event occurs for an operation that did not encounter a fixed linearisation point, then the concurrent trace records are used to determine the non-fixed linearisation points of this method by considering the fixed linearisation points of operations in other threads. The linearisability of the operation is checked according to the determined non-fixed linearisation points.

3. `void stateBacktracked(Search search)`. This method executes in the listener each time that JPF performs a state-backtrack operation. A result of the backtrack operation is that JPF may backtrack concurrent history trace events which are contained in the listener's trace records. This method is used by the listener to revert any backtracked history trace events and sequential oracle executions to align with that of state which JPF has backtracked to.

3.3.2.2 External Implementation of the Automatic Strategy

The following API calls are used for the external automatic checker implementation:

1. `void methodEntered(VM vm, ThreadInfo currentThread, MethodInfo enteredMethod)`.

Stellenbosch University https://scholar.sun.ac.za

This method executes the same function as the first paragraph describing this method in Section 3.3.2.1; it does not execute the functionality of the second paragraph, that is the linearisation point logic.

2. `void instructionExecuted(VM vm, ThreadInfo thread, Instruction nextInsn, Instruction executedInsn)`. This method executes the same function as the first paragraph describing this method in Section 3.3.2.1; it does not execute the functionality of the second paragraph, that is the linearisation point logic.

3. `void stateAdvanced(Search search)`. This method executes in the listener each time JPF advances from some parent state to a child state. When this method executes for an advance to an end-state, the listener uses the concurrent history trace records for the path till that end state to perform linearisability checking of the trace history.

4. `void stateBacktracked(Search search)`. This method executes the same function as described in Section 3.3.2.1.

3.3.3 Internal and External Comparison

The internal and external implementations have three main differences: manual user requirements, performance, and mechanisms for handling backtracking. We will now investigate each of these differences.

The internal implementations require more user involvement than the external checkers. The internal implementations require a user with advanced knowledge to manually add the correct code instrumentation for linearisability checking to the SUT; for algorithms of the automatic strategy as well as algorithms with correctly placed linearisation points for the linearisation point strategy. The external checker's listener performs linearisability checking that generalises to all configured algorithms for the automatic checking strategy, and to all algorithms with correctly placed fixed linearisation points for the linearisation point strategy; without any further involvement from the user.

Adding code instrumentation to the SUT translates to additional bytecode instructions, which results in more interleavings and thus longer paths and a larger state space. The external checker does not add any extra instrumentation to the SUT, apart from fixed linearisation point markers

Stellenbosch University https://scholar.sun.ac.za

for the linearisation point strategy, and thus the model checker's search space size is not impacted by the linearisability checking; **we thus expect an improvement in performance for the external implementation over the internal implementation.**

For the internal implementation, the backtracking of any concurrent history records or sequential oracle executions are handled by JPF's VM which backtracks these linearisability logic elements along with its backtrack of the SUT. The external implementations, however, keep these linearisability logic elements in the listener which is not included in JPF's VM state space. The external checker thus implements mechanisms, for each backtrack of JPF, that revert the concurrent history records and sequential oracle execution to that which aligns to the state and position in the concurrent history trace which JPF backtracked to. For each advance of JPF the concurrent history/sequential oracle information in the listener is updated and for each backtrack the information is reverted; ensuring the linearisability checking of each generated history tracked happens correctly.

3.4 SYMBOLIC CHECKING

The concrete model checkers, the internal as well as the external implementation of both the linearisation point and the automatic checkers, were integrated into JPF. In this section we describe the integration of the automatic strategy using the external implementation approach, into Symbolic PathFinder (SPF).

Symbolic PathFinder (SPF) is a symbolic execution extension to JPF. It maintains the core utilities of JPF, such as an exhaustive analysis of different thread interleavings, listener class utilities, state matching, and partial order reduction techniques, but also adds the benefits of symbolic execution and automatic test case generation offered by SPF. Symbolic symbols instead of concrete input values are used to represent a range of arbitrary concrete inputs. The model checker uses the symbols to generate all execution paths for all input value situations and thus explores all reachable sections of the input program; maximising program path coverage. SPF also provides an automatic test case generation mechanism which can be used to generate all possible test cases for a given number of executing operations.

A Symbolic Linearisability Checker can check linearisability of a data structure over a range of input values and test cases. This symbolic linearisability checking tool has both benefits and challenges,

Stellenbosch University https://scholar.sun.ac.za

which we will now discuss.

3.4.1 Benefits

A single run of the Symbolic Linearisability Checker is equivalent to multiple runs of the Concrete Linearisability Checker. The symbolic checker's use of symbolic execution and automatic test case generation produces a much broader and more robust linearisability check than concrete checking. There are three main reasons:

1. Program coverage is maximised, because all reachable sections of the SUT are executed by way of symbolic execution. The symbolic checker explores and checks the linearisability of all possible execution paths of the SUT for all input values of a given domain. This is possible since the inputs are not bound to concrete values, as is the case with a concrete linearisability checker; the inputs are expressed as symbolic values that represent a range of concrete values for a given execution path.

2. Symbolic values can represent multiple domains, whilst the concrete checker verifies the program over one domain at a time. The symbolic execution can check the behaviour of code across input from potentially unbounded data domains. For example, the symbolic values used throughout verification could represent domains of integer, double, and float.

3. The automatic test case generation results in the symbolic checker performing linearisability checks for multiple test case inputs, i.e. ordered sequence of operations for each executing thread, where the concrete checkers check linearisability for just one of these test cases.

The Symbolic Linearisability Checker eliminates the need for hand-crafted test cases. The Symbolic checker requires a single integer value for each executing thread: the number of operations to be executed on that thread, for example: `Thread 1: 2 operations` and `Thread 2: 1 operation`. This is in contrast to the concrete checkers which require input of an exact sequence of operations and parameter values for each executing thread, for example: `Thread 1: enqueue(1), enqueue(2)` and `Thread 2: dequeue()`.

The single test case of the concrete checker is on a higher level than ordinary testing – because JPF will execute all the unique interleavings of the bytecode instructions for each thread and thus systematically explore all the possible execution sequences reachable for the SUT on the specified

Stellenbosch University https://scholar.sun.ac.za

input, which eliminates the need for the programmer to manually create test cases for every possible execution sequence – but it is still possible that the user misses a possible input configuration that would have resulted in the finding of a linearisation error.

The hand-crafting of test cases resembles testing, it is possible that the user does not create a test case for the exact situation in which the error arises. The single input value per thread for the symbolic checker is used to generate and check all possible operation sequences and parameter values for the number bound. The symbolic checker thus provides the most general and robust linearisability check for the number bound.

The symbolic checker does not have the 'missed violation' problem of the concrete checker. The symbolic linearisability checker finds linearisability errors for data structures that can hold multiple instances of the same value. As explained in Section 3.5.3, the concrete checkers incorrectly determine that such a data structure is linearisable for certain test cases. Figure 3.10 depicts two concurrent histories for an existing error in the BuggyQueue algorithm. The histories labeled "Concrete" and "Symbolic" were recorded by a concrete and a symbolic linearisability checker of the BuggyQueue algorithm, respectively.

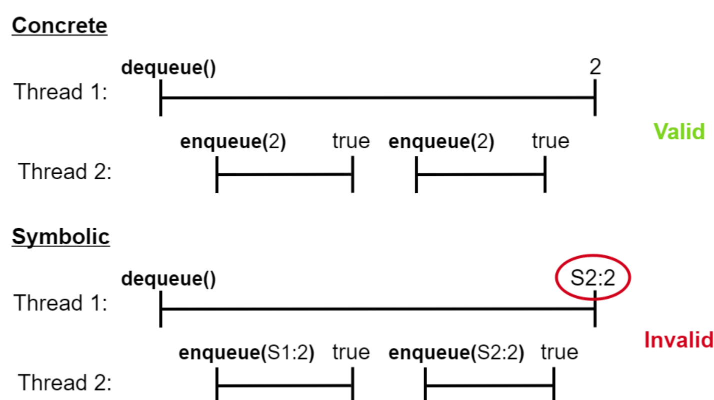


Figure 3.10: **Symbolic Linearisability Checker.** A duplicate-value concurrent history example for the BuggyQueue data structure where the concrete checker incorrectly verifies its linearisability and the symbolic checker correctly locates the linearisation error present and determines the concurrent history as not linearisable.

Both histories contain the error that thread-1's `dequeue` operation returns the value enqueued to the stack second, not first. The concurrent histories both show thread-1 execute a `dequeue` operation and thread-2 execute two `enqueue` operations that overlap with the thread-1 operation.

Stellenbosch University https://scholar.sun.ac.za

Both enqueue operations put different instances of the value 2 into the queue, but for the symbolic history, the symbols `S1` and `S2` have been assigned to the concrete input instances.

The concrete linearisability checkers do not identify the logical linearisability error in the history because it considers only the concrete response value and not the instance of that value which is being returned. The symbolic checker, however, considers the symbols `S1` and `S2` instead of the concrete values and thus notices that the instance of the value being returned, `S2`, is actually the one enqueued to the stack second; not correlating to the `S1` instance returned by the correct sequential specification.

Thus the symbolic linearisability checker correctly identifies the logical error where the concrete checker misses it. This illustrates that the symbolic linearisability checker is able to handle some situations with identical data-structure values while the concrete linearisability checker is only able to handle situations for unique values.

3.4.2 Challenges

Path conditions can be unsolvable. The Symbolic Linearisability Checker uses a solver, at different points in the symbolic model checking process, to find a solution to the current state's path condition constraints. If the path condition expression is unsolvable or overly complicated, often the case with effective hash functions, then the off-the-shelf solver may either produce an error or fall into an infinite loop; which prevents the linearisability checker from continuing execution.

One solution is to try different solvers, each with their own strengths and weaknesses, until one can effectively solve the path condition; but this solution does not work when the path condition is unsolvable. Pasareanu et al. proposes a mixed concrete-symbolic solving technique that uses annotations to indicate to the model checker which methods should be executed concretely [28]. By executing methods that contain complex formulae concretely it is possible to eliminate certain constraints from the path condition, such that it becomes a solvable expression.

Since annotations can only be added to methods within the SUT and hash functions are often located in compiled libraries, to which we do not have writing access except through exorbitant measures of run-time instrumentation of the compiled class's code, this solution was not considered viable and worth further investigation. At present, this problem is not solved.

Stellenbosch University https://scholar.sun.ac.za

Symbolic checking does not scale well. The symbolic checker does not scale with respect to time, particularly when multiple threads are used. Checking a program with loops or recursive behaviour using symbolic values can easily result in infinite loops and ultimately infinite execution paths, which causes SPF to perform very poorly. To improve efficiency, SPF utilises state matching to prevent re-computation of already explored branches and can be configured to backtrack at a predefined maximum search depth. It should be noted that the setting of a depth limit introduces the risk of missing linearisation errors due to premature path cut-offs and that even with these features enabled, SPF still does not scale well; it is thus most effective for unit or sub-system level testing.

The sequential oracle requires concrete values. Checking the validity of a linearisation is challenging when working with symbolic input values. Symbolic values do not hold comparative properties so they cannot be used for logical operations in the execution of the sequential oracle. A constraint containing expression similar in nature to a path condition could be maintained for the sequential oracle and a solver utilised, to determine representative concrete values to use when executing the sequential oracle's comparative operations.

A simpler solution is to define each symbolic variable as a tuple containing both a concrete value (a representative from the range of possible concrete values) as well as the unique symbolic symbol identifying it as different from another instance of the variable with the same concrete value. This allows the sequential oracle to make comparative logic operations as well as allow the listener to recognise the errors/violations missed by the concrete checker. This solution was chosen, even though it necessitates instrumenting the sequential oracle class to allow for tuple manipulation.

3.4.3 Implementation of Symbolic Checkers

We have implemented an external automatic symbolic linearisability checker. The internal and linearisation point symbolic checkers were not implemented for the following reasons:

- An internal symbolic checker implementation was found to be impractical. Preliminary results for this checker showed severe scalability problems which renders it unusable. As an example, the internal symbolic checker executed for 29 minutes over the BuggyQueue search space where the external symbolic checker took 3 seconds for the same search space traversal. We

Stellenbosch University https://scholar.sun.ac.za

thus decided to abandon the internal symbolic implementation.

- A linearisation point symbolic checker has its challenges for integration with SPF because the linearisation point linearisability checking strategy is not easily compatible with the symbolic model checker's state space traversal logic. It might, however, be beneficial for the symbolic domain.

  The symbolic checker requires, for most situations, a depth limit in order to execute within a reasonable amount of time. The depth limit results in end-state cut offs, and the end states are exactly where the automatic checking strategy does its linearisability checking. The depth limit prevents many generated history paths to be traversed up until the depth limit but not checked for linearisability by the automatic checking strategy because the end state is cut off. The linearisability checking strategy performs on-the-fly linearisability comparisons and thus would utilise all opportunities for linearisability checking up until the cut-off point in the path, irrespective of whether or not the end state was cut-off. Thus the linearisation point strategy would be beneficial over the automatic strategy for a badly scaling symbolic setting. This checker was not implemented, but could be investigated in future work.

We will now elaborate on the implementation details for the external automatic symbolic linearisability checker, the symbolic checker implemented in this thesis.

3.4.3.1 Listener API calls

The external implementations use a JPF listener and listener API calls to gain access to the model checker's search information, and then use the information to perform linearisability checking. We will now describe the API calls utilised by the external hybrid and the external symbolic checkers and the purposes for which they are used.

1. `void methodEntered(VM vm, ThreadInfo currentThread, MethodInfo enteredMethod)`. This method executes the same function as the first paragraph describing this method in Section 3.3.2.1 but applied to SPF instead of JPF; it does not execute the functionality of the second paragraph, that is the linearisation point logic.

2. `void instructionExecuted(VM vm, ThreadInfo thread, Instruction nextInsn, Instruction executedInsn)`. This method executes in the listener each time SPF executes

Stellenbosch University https://scholar.sun.ac.za

a operation response event, the time-ordered response information of each executed operation is used to keep a record of all the operation invocations pertaining to the concurrent history trace; each response value is recorded in the form of a symbolic identification key when available, otherwise the concrete value is stored.

3. `void stateAdvanced(Search search)`. This method executes in the listener each time SPF advances from some parent state to a child state. For the situations where this method executes for an advance to an end-state, the listener uses the concurrent history trace records and the path condition of the end state to perform linearisability checking of the trace history. In linearisability checking of the end state's path history, the path condition is passed to a solver which generates representative concrete values for the symbolic state variables; these concrete values, along with their symbolic names, are then used to execute the sequential oracle.

4. `void stateBacktracked(Search search)`. This method executes the same function as the first paragraph describing this method in Section 3.3.2.1 but applied to SPF instead of JPF.

3.4.3.2 Execution of the sequential oracle in a symbolic setting

SPF uses Choco as the default constraint solver, other solvers such as z3 can be configured instead of Choco, in the configuration file of jpf file for each SUT.

Executing the sequential oracle in a symbolic setting poses a challenge due to the nature of symbols not having the comparative properties of the concrete values they represent. The symbols cannot be used as input parameters to the sequential oracle since the oracle's operations may contain comparative logic; for which symbolic values are not suitable.

As discussed under Section 3.4.2, one approach to solving this is to use a path condition for executing the sequential oracle and a solver for generating concrete values where necessary. We have chosen a simpler approach of grouping each symbol used as input to the sequential oracle along with a concrete representative value found by using a solver. We then pass this tuple to the sequential oracle and allow it to use the concrete values, but always group the symbol identifying the exact instance of the value along with it in the Tuple.

Our chosen approach allows the execution of the sequential oracle so that the tuple response values

Stellenbosch University https://scholar.sun.ac.za

can be used for extracting the respective returned symbol for comparison to the concurrent history operation's symbolic response value.

3.4.3.3 Parameter and return type configuration capabilities:

The tool is currently able to handle no more than two parameter values for an operation in the SUT with types of integer, double, or boolean. For running the tool it is necessary to configure the parameter and return types for all method operations available to the SUT.

3.4.4 A Hybrid Checker

The symbolic linearisability checker uses automatic test case generation to generate and check all possible test cases for a given number of operations executed per thread, and it uses symbolic execution to check linearisability for all reachable program paths of those test cases. The Hybrid Checker, a concrete-symbolic hybrid, uses symbolic execution to check linearisability of all reachable program paths; but turns automatic test case generation off and instead checks linearisability of just one particular user-specified test case.

Table 3.2 shows the different input requirements of the Concrete, Hybrid, and Symbolic Checkers. The Concrete Checker performs linearisability checking for the particularly defined input test case: the sequence of operations and argument values per thread. The Hybrid checker performs linearisability checking for the particular defined sequence of operations per thread but instead of concrete values, uses symbolic argument values and symbolic execution to check all reachable program paths of the test case. The Symbolic Checker takes as input only a number of general operations per thread. It performs linearisability checking for all possible operation sequences given the number of operations per thread constraint and for each of these test cases then uses symbolic argument values instead of concrete values; thus performing linearisability for all possible test cases and reachable program paths for the input constraints.

	Concrete	**Hybrid**	**Symbolic**
BuggyQueue (unique value)	thr1: *dequeue()* thr2: *enqueue(8), enqueue(9)*	thr1: *dequeue()* thr2: *enqueue(sym), enqueue(sym)*	thr1: *1 operation* thr2: *2 operations*

Table 3.2: Test suite of SUT algorithms and the operation-sequence test cases used for the concrete, hybrid, and symbolic checkers.

Stellenbosch University https://scholar.sun.ac.za

We have described the design and implementation of the internal/external automatic/linearisation-point concrete linearisability checkers, the automatic external symbolic checker, and the automatic external hybrid checker. The completeness and soundness of these checkers, with respect to linearisability, is discussed in the next section.

3.5 COMPLETENESS AND SOUNDNESS

Completeness and soundness with respect to linearisability analysis is defined here according to the definitions and discussion provided by Pezzè and Young [31] and Meyer [25], respectively.

Definition 4. *A linearisability analysis is* ***complete*** *if it always reports that a data structure is linearisable when the data structure actually is linearisable (accepts all desirables) [31].*

Thus, a linearisability analysis is complete if it accepts all linearisable data structures. The linearisability checkers in this thesis are complete, because all caught violations are real violations (never reports false alarms) and when no violations are detected, it reports that the data structure is linearisable.

Definition 5. *A linearisability analysis is* ***sound*** *if it reports that a data structure is linearisable only when it actually is linearisable (accepts only desirables) [31].*

Thus a linearisability analysis is sound if it accepts only linearisable data structures. If a linearisation error is present the linearisability analysis must find it to be considered sound (no violations may be missed); otherwise it will report that the data structure is linearisable.

Soundness, in this thesis, is with respect to the input test case configured for the program execution. Different factors can contribute towards compromising soundness of the linearisability checkers such as a depth bound, conflicts in the hash function, model checking state-space optimisation techniques, and unsolvable path conditions.

We first elaborate on JPF's soundness-compromising state-space optimisation technique and present a soundness guaranteeing solution to this problem, we discuss a problem with the concrete linearisability checkers, which compromises soundness, and we then investigate soundness of the symbolic linearisability checkers.

Stellenbosch University https://scholar.sun.ac.za

3.5.1 JPF's state hashing optimisation causes unsoundness with respect to linearisability

As we discussed in Section 2.1.4.2, JPF uses state hashing as an optimisation technique to alleviate state space explosion. JPF's state hashing introduces the possibility of the linearisability checker missing errors because the optimisation could cause error containing paths to be cut-off. We explained that JPF maintains a hashtable of all the states in its search space. Each hashed state includes program information for the heap and thread-stack snapshots, but the hashing of these program elements does not guarantee soundness with respect to linearisability. Each time a state is re-encountered during JPF execution, the branch leading from that state is ignored because it was explored when the state was first encountered, the branch is cut off. Different concurrent history paths may lead to states that are equivalent in terms of the hash. This introduces the problem that JPF considers different history paths as equivalent, and therefore could ignore exploration of certain paths it considers re-encountered when in fact contains a previously unexplored concurrent history trace that should be checked for linearisability.

An example of such a situation is shown in Figure 3.11 for the LockFreeList algorithm. The figure shows two concurrent history traces generated by JPF, path 2 is generated after path 1. The vertical blue dashed line shows the point during both history paths at which JPF would consider the program state equivalent. We call this program state, State X; the data structure contains the value of seven at State X. The trace labelled path 1 is linearisable, but the trace in path 2 is not and contains an error for the *false* response value of the `contains` operation (depicted by the red oval).

The order in which these two history paths are generated affects path cut-offs. Lets assume path one is generated first, all five operations executed and the path's concurrent history proven linearisable. Later JPF begins to generate path two, but when it reaches State X it realises that it finds State X in its hashtable and thus ignores any branches leading from this state; the model checker therefore never generates the contains operation that would reveal the linearisation error and the error is not picked up.

Stellenbosch University https://scholar.sun.ac.za

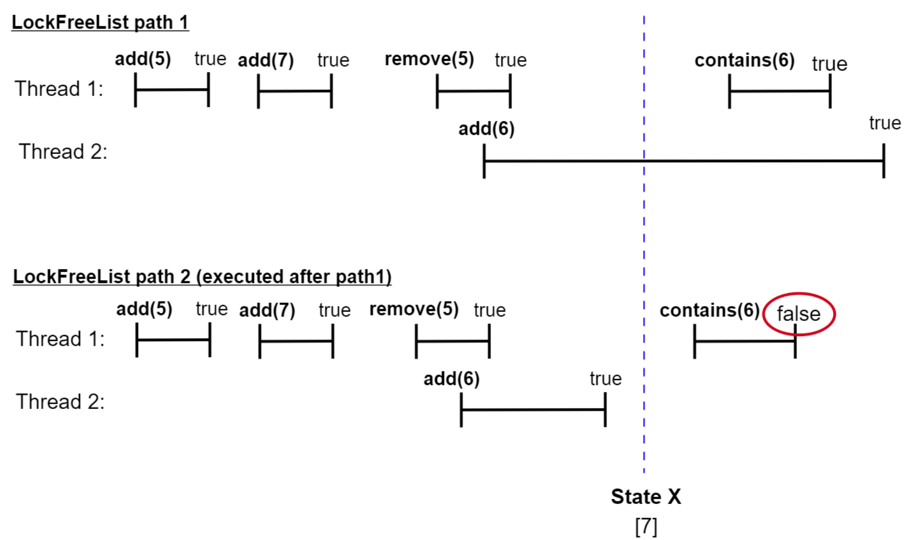


Figure 3.11: **Soundness in JPF.** An example showing two concurrent histories where, when traversed by JPF in order, JPF makes an incorrect cut off of path 2 due to state hashing and misses the linearisation error in the cut-off path. JPF hashes the history segments of both, from start until the vertical-dotted-blue line, as equivalent. These two hashed segments are not equivalent since they produce two different sets of linearisations. For an execution where JPF explored path 2 after path 1, then JPF will make an incorrect path cut-off and miss the linearisation error contained in path 2.

Stellenbosch University https://scholar.sun.ac.za

3.5.2 A strategy to guarantee soundness with respect to linearisability of the input, in JPF

One strategy to eliminate the unsoundness problem in JPF is to turn off the hashing optimisation. Our experiments showed that the model checking search space produced by an unoptimised execution is significantly larger than that of an optimised execution; even small examples execute past the timeout period and the tool is not usable.

A more efficient strategy is to add soundness guaranteeing code instrumentation to the SUT to force JPF to keep the trace information at its states, and thus include the trace information in the state hash, so that it will be able to differentiate all unique concurrent history paths. This solution ensures that JPF generates paths for all unique concurrent history possibilities, and does not skip any, while still utilising its hash optimisation technique to alleviate the state space explosion problem of model checking.

Each time a concurrent history invocation or response event is traversed by JPF, the operation name, argument(s), response value, and thread that executed the operation are included in an object and pushed to a queue data structure; the queue maintains the time-ordering of the events and the pushed object maintains all the uniqueness-related information about the event. An example soundness instrumentation code fragment is shown below:

Soundness instrumentation method:

```
private void soundnessMarkerBeginMethod(String methodName, boolean responseValue) {
  if (isSound) queue.add(new instructionForSoundness(methodName, responseValue,
  (int)Thread.currentThread.getId()));
}
```

Each of the concrete linearisability checking tools can be configured to execute in either sound or unsound mode. It is important to qualify that the soundness of an execution can be compromised if a bad hash function is used, resulting in hash conflicts that cause incorrect path cut-offs.

3.5.3 Missed Violations of the Concrete Checkers

It was found that the concrete linearisability checkers sometimes missed violations for data structures that may contain multiple instances of the same value; that is the history is incorrectly verified

Stellenbosch University https://scholar.sun.ac.za

as linearisable where a linearisation error was actually present.

This problem of the concrete checkers does compromise soundness in that errors can be missed for these duplicate-value situations. For purposes of this thesis we will refer to the sound concrete checkers as such, excluding the mention of this behaviour, and not using examples for which this behaviour will arise, when considering soundness of the linearisability checker executions.

An example of such a missed violation is when the BuggyQueue algorithm, that has a known bug (see the test suite in Section 4.1), is checked for linearisability with input: `Thread 1: dequeue()` and `Thread 1: enqueue(2), enqueue(2)`. A concurrent history is shown in Figure 3.12, in this history both `enqueue` operations put the value of two into the queue, we have identified each unique instance of the value as `2A` and `2B`.

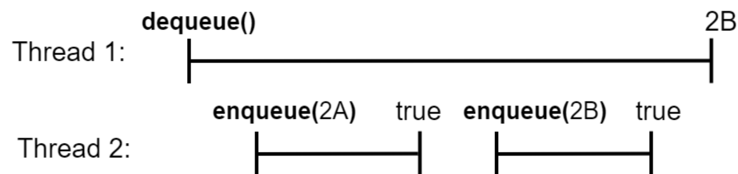


Figure 3.12: **An example situation where the concrete checkers incorrectly verify linearisability** The example is for the BuggyQueue algorithm.

If only the concrete values 2 and 2 are compared, the concurrent history does correlate to the execution of a correct sequential oracle, but if the instance of the value being returned is considered, we see that the history contains an error in that the instance not at the front of the queue was returned. Thus the logical error is not identified by the concrete checkers.

Section 3.4.1 explained that the Symbolic Linearisation Checker does not have this problem; allowing linearisability checking of data structures for situations where the data structure contains duplicate values.

3.5.4 Soundness in SPF

The symbolic and the hybrid linearisability checkers use SPF, the symbolic extension to the JPF model checker. SPF switches off the hash optimisation used by JPF, the soundness problem described for JPF does not apply to SPF. However, SPF is unsound due to symbolic execution's infinite looping behaviour and the possibility of unsolvable path conditions. For programs with

Stellenbosch University https://scholar.sun.ac.za

loops, the symbolic execution can generate an infinite sequence of interleavings; a depth limit is necessary, all possible interleavings cannot be checked for linearisability and thus an error can be missed. The solver used for SPF may encounter situations where a path condition is unsolvable, SPF will either throw an error or ignore that path; in either of these situations a linearisability error can be missed. However, up until the depth limit for those paths generated by SPF, the symbolic and the hybrid linearisability checkers are sound.

3.6 THE JPF/SPF LINEARISABILITY CHECKING EXTENSION FRAMEWORK

The concrete linearisability checkers are available as an extension to JPF called jpf-linearisable. The symbolic checker and hybrid checker are available as aan extension to SPF called jpf-symb. The extensions have all five test suite algorithms (see Section 4.1)) implemented; for jpf-linearisable for both internal and external implementations.

3.6.1 jpf-linearisable

To add a new SUT to jpf-linearizable, the following files are required:

- **The SUT** object class included inside of the SUT wrapper Java class. This file should be added to the "src/examples" directory.
- **Code instrumentation must be added to the SUT for those respective checkers that require it.** A separate .java file must be used for the SUT to be tested using external checking (un-instrumented SUT) and to be tested using internal checking (instrumented SUT).
- **A correct sequential specification of the SUT** should be included in the "src/examples/sequentialExecutions" or the "src/main/za/ac/sun/jpf/linearizable/sequentialExecutions" folders for the internal and external checker's utility, respectively.
- **A configuration file** is required for each checker experiment; the configuration fields determine the test case and checker type for the experiment. The file has a .jpf extension and are located in the "src/examples/" directory.

Stellenbosch University https://scholar.sun.ac.za

A run.sh file is provided in jpf, this file can be used to run test cases (for which config files have been created) and toggle the execution settings. The following execution settings can be toggled from within this run.sh file:

- **Concrete Checker**: "Automatic", "AutomaticLazyRead", "LinPoints", "AutomaticHash" "AutomaticLazyReadHash".
- **Implementation**: "Internal", "External".
- **SUT**: "BuggyQueue", "PairSnap", "LockFreeSet", "LockFreeList", "SnarkDeque".
- **Soundness settings**: "Sound", "Unsound".

The user can thus use these configurations to execute any combination of the listed settings and execute their desired linearisability check on one of their own implemented non-blocking concurrent data structures.

3.6.2 jpf-symb

To add a new SUT to jpf-symb, the following files are required:

- **The SUT** object .java class and a SUT driver .java class for that SUT. This file should be added to the "src/examples/linearizability" directory.
- **A correct sequential specification** of the SUT should be included in the "src/main/gov-/nana/jpf/symbc/linearizabilityListeners/sequentialExecutions" folder.
- **A configuration file** is required for each checker experiment; the configuration fields determine the test case and checker type for the experiment. The file has a .jpf extension and are located in the "src/examples/linearizability" directory.

A run.sh file is provided in jpf, this file can be used to run test cases (for which config files have been created) and toggle the execution settings. The following execution settings can be toggled from within this run.sh file:

- **SUT**: "BuggyQueue", "PairSnap", "LockFreeSet", "LockFreeList", "SnarkDeque".
- **Symbolic Checker**: "Hybrid", "Symbolic"
- **Number of generic operations per thread**

Stellenbosch University https://scholar.sun.ac.za

- **Depth limit**

A user can thus use these configurations to choose any combination of the listed settings and execute the chosen symbolic linearisability check on one of their own implemented non-blocking concurrent data structures.

3.7 DESIGN AND IMPLEMENTATION CONCLUSIONS

The design of two linearisability checking strategies, called linearisation point strategy and automatic strategy, have been described in this chapter; as well as two optimisation techniques for the automatic strategy. There is two structurally different ways that each of these strategies can be implemented, internally and externally. We have developed both internal and external concrete checkers for each of the checking strategies. We presented the symbolic checker and a concrete-symbolic hybrid checker for which there are only external implementations. The concrete linearisability checkers use JPF, a concrete model checker, and the symbolic checkers use SPF, a symbolic model checker. An unsoundness causing JPF optimisation was described and we proposed a soundness guaranteeing solution to this problem. Finally, we described the jpf-linearisable and jpf-symb tools and how they can be used. In the next chapter we will evaluate the efficiency, scalability, and error finding ability of each of our implemented checkers.

Stellenbosch University https://scholar.sun.ac.za

CHAPTER 4

RESULTS AND ANALYSIS

In this chapter all of the checkers implemented for this thesis are evaluated and compared using the same model checking framework and on the same hardware. The details of the system on which the experiments were run and the checker's manual input requirements are given in Section 4.1, the checker's efficiency results are shown in Section 4.2, scalability of the checkers discussed in Section 4.3, and the error finding capability of each checker investigated in Section 4.4. The checkers can be categorised into the following four types, they have either an internal or external implementation and can run in either sound or unsound mode.

- **Concrete Linearisation Point** (Section 3.1).
- **Concrete Automatic** (Sections 3.2 and 3.2.1.1).
- **Symbolic** (Section 3.4)
- **Hybrid** (Section 3.4)

4.1 MACHINE SPECS AND CHECKER INPUTS USED FOR THE EXPERIMENTS IN THIS CHAPTER

All experiments were performed on a machine running Ubuntu 18.04.5 with 16GB RAM and an Intel Core i7-8665U processor (4 cores, 8 threads).

All of the checkers require input of *(a)* a SUT which is the concurrent data structure implemented in Java and *(b)* a correct sequential implementation of the SUT data structure, the sequential oracle. The other input requirements are specific to the respective checker categories, they are shown in Table 4.1.

The concrete checkers are the most user-intensive, they require *(c)* a user-specified test case, which contains a sequence of operations for each executing thread, and the *(d)* user-specified arguments for all operations in the test case. The concrete linearisation point checkers also require *(f)* manual code instrumentation for user-specified linearisation points to be added in the SUT. The internal concrete checkers require *(g)* manually added linearisability-checking code-instrumentation

Stellenbosch University https://scholar.sun.ac.za

	(a) SUT	(b) Sequential oracle	(c) Sequence of operations in the test case (per thread)	(d) Argument values for operations in the test case (per thread)	(e) Number of generic operations in the test case (per thread)	(f) Manual code instrumentation for linearisation points	(g) Manual code instrumentation for linearisability checking logic
Concrete (JPF)							
Linearisation Point	yes	yes	yes	yes	no	yes	internal yes external no
Automatic	yes	yes	yes	yes	no	no	internal yes external no
Symbolic (SPF)							
Symbolic	yes	yes	no	no	yes	no	no
Hybrid	yes	yes	yes	no	no	no	no

Table 4.1: **Input specification for the different linearisability checker implementations:** *(a)* Concurrent data structure (SUT), *(b)* Sequential specification of the SUT (sequential oracle), *(c)* Sequence of operations in the test case, for each executing thread, *(d)* Argument values for the operations in the test case, *(e)* Number of generic operations to be executed by each thread in the test case, *(f)* Manual code instrumentation for user-specified linearisation points added to the SUT, and *(g)* Manual linearisability-checking-logic code instrumentation added to the SUT.

in the SUT. For the concrete internal automatic checkers, little user knowledge of the SUT is required to add this correctly, but for the concrete internal linearisation point checkers an in-depth understanding of the SUT is required to add this correctly.

The symbolic checker is the least user-intensive, it requires only *(e)* one user-specified integer per executing thread; the integer defines the number of generic operations to be executed by the thread. The symbolic checker uses the integer input value(s) to generate all possible test cases, and generate argument values for the test case operations such that all reachable program paths are explored; for the input number bound. The hybrid checker requires *(c)* a user-specified test case, which contains a sequence of operations for each executing thread, but does not require the arguments for all operations in the test case; instead it uses symbolic execution to generate argument values such that all reachable program paths are explored.

The concrete checker's usefulness in finding linearisability errors is constrained by the user's ability to hand-craft test cases in which errors are present. **The symbolic checker performs linearisability checking on all possible test cases and verifies the linearisability of a data structure in general**, constrained only by the number of operations to be executed by each thread.

The test suite used for experiments in this chapter includes the following five data structures with six know linearisation errors: BuggyQueue [33] (1 linearisation error), LockFreeList [34] (2 linearisation errors), PairSnap [32] (1 linearisation error), SnarkDeque [7] (2 linearisation errors),

Stellenbosch University https://scholar.sun.ac.za

and LockFreeSet [39] (0 linearisation errors). For all experiments the maximum memory available was set to 2048 MB and the maximum timeout period was set to 12 hours. The test suite and test cases used for experiments in this chapter are shown in Table 4.2.

	Concrete	**Hybrid**	**Symbolic**
BuggyQueue (unique value)	thr1: *dequeue()* thr2: *enqueue(8), enqueue(9)*	thr1: *dequeue()* thr2: *enqueue(sym), enqueue(sym)*	thr1: *1 operation* thr2: *2 operations*
BuggyQueue (duplicate value)	thr1: *remove(0)* thr2: *add(8), add(8)*	thr1: *dequeue()* thr2: *enqueue(sym), enqueue(sym)*	thr1: *1 operation* thr2: *2 operations*
LockFreeList Bug1	thr1: *add(5), add(6),add(7), remove(5), remove(6)* thr2: *remove(6)*	thr1: *add(sym), add(sym),add(sym), remove(sym), remove(sym)* thr2: *remove(sym)*	thr1: *5 operations* thr2: *1 operation*
LockFreeList Bug2	thr1: *add(5), add(7), remove(5), contains(6)* thr2: *add(6)*	thr1: *add(sym), add(sym), remove(sym), contains(sym)* thr2: *remove(sym)*	thr1: *4 operations* thr2: *1 operation*
PairSnap	thr1: *write(1,101), write(3,201), write(1,301)* thr2: *readPair(1,3)*	thr1: *write(sym, sym), write(sym, sym), write(sym, sym)* thr2: *readPair(sym, sym)*	thr1: *3 operations* thr2: *1 operation*
SnarkDeque Bug1	thr1: *popRight(0)* thr2: pushRight(4), pushRight(5), popLeft(0)	thr1: *popRight()* thr2: *pushRight(sym), pushRight(sym), popLeft()*	thr1: *1 operation* thr2: *3 operations*
SnarkDeque Bug2	thr1: *popRight(0)* thr2: *pushRight(6), popLeft(0)*	thr1: *popRight()* thr2: *pushRight(sym), popLeft()*	thr1: *1 operation* thr2: *2 operations*
LockFreeSet	thr1: *add(5), add(5)* thr2: *remove(5)*	thr1: *add(sym), add(sym)* thr2: *remove(sym)*	thr1: *2 operations* thr2: *1 operation*

Table 4.2: Test suite of SUT algorithms and the operation-sequence test cases used for the concrete, hybrid, and symbolic checkers.

4.2 EFFICIENCY

In this section we evaluate and compare the resource-usage requirements of each of the checkers for their execution of the the test suite algorithms, over the entire search space. The performance-limiting factor of each checker is investigated and the effectiveness and overall benefit of the two automatic checker optimisations determined.

Stellenbosch University https://scholar.sun.ac.za

4.2.1 Resource usage of the Concrete Checkers

The time, memory, and search space requirements of all the concrete checkers are compared in this section. The resource usage of each checker's execution is compared to that of the stand-alone model checking execution, for a SUT, to determine the contribution of the model checker and the linearisability checker, respectively, on each tool's overall resource usage. There are eight different concrete checkers: two different linearisability checking strategies are used: 1. Linearisation Point Linearisability Checking, and 2. Automatic Linearisability Checking, each of the two strategies can be implemented either internally or externally, and each implemented checker can run in either a sound or an unsound mode.

The time and memory usage results for these experiments are shown in Table 4.3 and the model checking search space information is shown in Table 4.4. Memory results include the number of states created during the model checking execution and the maximum memory allocation necessary for the search; maximum memory allocation is adjusted dynamically by JPF. The model checking search space information shows the number of visited, backtracked, and end states for the execution and the maximum depth reached during state space traversal. The search space size is shown to be vastly different from algorithm to algorithm. This is because the base search space size is determined by the number of bytecode instructions contained in the un-instrumented SUT i.e., the size of the algorithm. Irrespective of which algorithm is tested, the comparative efficiency of the checker types remains constant.

The analysis of the experiment data in Table 4.4 shows that **there is a correlation between the amount of code instrumentation included in the SUT and the execution time of the checkers**. The amount of code instrumentation added to the SUT and the execution time of the linearisability checking for each checker, can be expressed in the following relation: *Sound Internal Linearisation Point* > *Sound Internal Automatic* > *Sound External Automatic/Linearisation-Point* > *Unsound Internal Linearisation Point* > *Unsound Internal Automatic* > *Unsound External Automatic/Linearisation-Point*. The influence of the amount of instrumentation, but also the kind of instrumentation, on the checker's resource usage will be analysed.

Stellenbosch University https://scholar.sun.ac.za

SUT algorithm	Data Format:	Lin. Checker turned off			
		stand-alone model checker		trace info. maintained at model checker's states	
		Unsound (time: +-1s)	Sound (time: +-10s)	Unsound (time: +-10s)	Sound (time: +-10s)
BuggyQueue	Time (seconds)	3.74	726	101	12,928
	New States	17,715	3,272,426	441,438	51,246,707
	Max Mem.(MB)	675	977	425	2,675
LockFreeList Bug1	Time (seconds)	4.01	11,983	800	Memory
	New States	17,966	46,885,440	3,418,545	limit
	Max Mem.(MB)	425	1,126	679	reached
LockFreeList Bug2	Time (seconds)	3.84	3,482	192	Memory
	New States	16,764	15,111,856	819,284	limit
	Max Mem.(MB)	675	569	427	reached
PairSnap	Time (seconds)	2.24	1,010	216	Memory
	New States	10,095	4,735,998	963,268	limit
	Max Mem.(MB)	425	425	425	reached
SnarkDeque Bug1	Time (seconds)	3.56	1,918	353	Memory
	New States	15,253	7,178,784	1,510,982	limit
	Max Mem.(MB)	425	425	676	reached
SnarkDeque Bug2	Time (seconds)	2.28	466	114	15,184
	New States	8,964	1,946,860	499,689	60,508,984
	Max Mem.(MB)	425	425	425	2,047
LockFreeSet	Time (seconds)	3.89	503	49	13,018
	New States	5,082	2,343,680	234,178	53,473,053
	Max Mem.(MB)	300	676	426	2,690

SUT algorithm	Data Format:	External (lin. checker turned on)				Internal (lin. checker turned on)			
		Unsound (time +-1s)		Sound (time +-10s)		Unsound (time +-10s)		Sound (time +-10s)	
		Aut.	Lin. Point	Aut.	Lin. Point	Aut.	Lin. Point	Aut.	Lin. Point
BuggyQueue	Time (seconds)	5.11	4.38	776	780	106	5,941	14,970	Memory
	New States	17,715	17,563	3,272,426	3,224,768	441,438	14,498,966	51,246,707	limit
	Max Mem.(MB)	675	425	674	676	425	680	2,675	reached
LockFreeList Bug1	Time (seconds)	5.76	4.46	12,268	12,155	885	4,329	Memory	Memory
	New States	17,966	17,946	46,885,440	44,970,952	3,418,545	12,753,166	limit	limit
	Max Mem.(MB)	425	425	1,126	1,069	679	698	reached	reached
LockFreeList Bug2	Time (seconds)	5.44	4.10	3,895	3,789	208	1,080	Memory	Memory
	New States	16,764	16,754	15,111,856	14,698,304	819,284	2,138,151	limit	limit
	Max Mem.(MB)	676	426	739	751	427	679	reached	reached
PairSnap	Time (seconds)	3.19	2.25	1,056	1,080	229	4,320	Memory	Memory
	New States	10,095	10,077	4,735,998	3,978,630	963,268	11,836,968	limit	limit
	Max Mem.(MB)	676	424	674	676	425	758	reached	reached
SnarkDeque Bug1	Time (seconds)	4.39	3.83	1,678	1,680	360	31,800	Memory	Memory
	New States	15,253	15,253	7,178,784	7,175,974	1,510,982	77,650,632	limit	limit
	Max Mem.(MB)	425	424	675	675	676	2,040	reached	reached
SnarkDeque Bug2	Time (seconds)	3.32	2.36	435	450	121	5,926	16,650	Memory
	New States	8,964	8,986	1,946,860	1,915,213	499,689	17,986,678	60,508,984	limit
	Max Mem.(MB)	425	424	676	674	425	682	2,047	reached
LockFreeSet	Time (seconds)	1.29	3.94	541	540	52	210	14,100	Memory
	New States	5,082	5,082	2,343,680	2,343,680	234,178	655,330	53,473,053	limit
	Max Mem.(MB)	300	300	676	676	426	678	2,690	reached

Table 4.3: **Concrete: Execution Time and Memory for Entire Search Space.** The execution time, number of new states, and maximum memory statistics for each of the internal/external sound/unsound linearisation-point/automatic/checker-turned-off concrete checker types for execution of the test suite over the entire search space.

Stellenbosch University https://scholar.sun.ac.za

SUT algorithm	**Data Format:**	**Lin. Checker turned off**			
		stand-alone model checker		**trace info. maintained at model checker's states**	
		Unsound	**Sound**	**Unsound**	**Sound**
BuggyQueue	Visited:	28,206	4,983,342	682,006	76,035,682
	Backtracked:	45,921	8,255,768	1,123,444	127,282,389
	End:	398	136,985	1,024	245,988
	Max Depth:	102	178	192	269
LockFreeList Bug1	Visited:	28,548	67,193,481	5,192,870	Memory limit reached
	Backtracked:	46,514	114,078,921	8,611,350	
	End:	12	248,472	4,323	
	Max Depth:	129	288	315	
LockFreeList Bug2	Visited:	26,761	21,801,688	1,235,913	Memory limit reached
	Backtracked:	43,525	36,913,544	2,054,705	
	End:	8	88,956	1,711	
	Max Depth:	122	257	274	
PairSnap	Visited:	15,944	6,856,994	1,482,222	Memory limit reached
	Backtracked:	26,039	11,592,992	2,445,420	
	End:	154	2,77,913	2,226	
	Max Depth:	90	193	204	
SnarkDeque Bug1	Visited:	23,735	10,519,554	2,333,332	Memory limit reached
	Backtracked:	38,988	17,698,338	3,844,014	
	End:	350	392,300	2,862	
	Max Depth:	118	214	234	
SnarkDeque Bug2	Visited:	13,955	2,877,073	765,533	90,160,688
	Backtracked:	22,919	4,823,933	1,265,222	150,669,672
	End:	220	93,815	1,104	279,972
	Max Depth:	94	170	181	257
LockFreeSet	Visited:	7,848	3,384,784	355,046	78,949,292
	Backtracked:	12,930	5,728,464	589,224	132,422,345
	End:	8	17,968	896	332,096
	Max Depth:	74	163	163	249

SUT algorithm	**Data Format:**	**External (lin. checker turned on)**				**Internal (lin. checker turned on)**			
		Unsound		**Sound**		**Unsound**		**Sound**	
		Aut.	**Lin. Point**	**Aut.**	**Lin. Point**	**Aut.**	**Lin. Point**	**Aut.**	**Lin. Point**
BuggyQueue	Visited:	28,206	27,998	4,983,342	4,914,229	682,006	22,589,835	76,035,682	Memory limit reached
	Backtracked:	45,921	45,561	8,255,768	8,138,997	1,123,444	37,088,801	127,282,389	
	End:	398	342	136,985	133,506	1,024	1,684	245,988	
	Max Depth:	102	102	178	178	192	1,405	269	
LockFreeList Bug1	Visited:	28,548	28,522	67,193,481	64,812,917	5,192,870	19,973,060	Memory limit reached	Memory limit reached
	Backtracked:	46,514	46,468	114,078,921	109,783,869	8,611,350	32,726,226		
	End:	12	12	248,472	183,324	4,323	10,728		
	Max Depth:	129	129	288	288	315	652		
LockFreeList Bug2	Visited:	26,761	26,748	21,801,688	21,319,096	1,235,913	3,278,144	Memory limit reached	Memory limit reached
	Backtracked:	43,525	43,502	36,913,544	36,034,886	2,054,705	5,416,295		
	End:	8	8	88,956	74,700	1,711	1,940		
	Max Depth:	122	122	257	257	274	577		
PairSnap	Visited:	15,944	15,914	6,856,994	5,800,658	1,482,222	19,138,067	Memory limit reached	Memory limit reached
	Backtracked:	26,039	25,991	11,592,992	9,779,288	2,445,420	30,975,035		
	End:	154	117	2,77,913	214,875	2,226	2,968		
	Max Depth:	90	91	193	19	204	377		
SnarkDeque Bug1	Visited:	23,735	23,735	10,519,554	10,514,946	2,333,332	122,197,714	Memory limit reached	Memory limit reached
	Backtracked:	38,988	38,988	17,698,338	17,690,920	3,844,014	199,848,346		
	End:	350	350	392,300	392,250	2,862	6,688		
	Max Depth:	118	118	214	214	234	1,730		
SnarkDeque Bug2	Visited:	13,955	13,925	2,877,073	2,832,428	765,533	28,087,386	90,160,688	Memory limit reached
	Backtracked:	22,919	22,911	4,823,933	4,747,641	1,265,222	46,074,064	150,669,672	
	End:	220	205	93,815	91,595	1,104	2,228	279,972	
	Max Depth:	94	89	170	167	181	1,370	257	
LockFreeSet	Visited:	7,848	7,848	3,384,784	3,384,784	355,046	1,023,932	78,949,292	Memory limit reached
	Backtracked:	12,930	12,930	5,728,464	5,728,464	589,224	1,679,262	132,422,345	
	End:	8	8	17,968	17,968	896	896	332,096	
	Max Depth:	74	74	163	163	163	393	249	

Table 4.4: **Concrete: Search Space Statistics for Entire Search Space** The number of visited, backtracked, and end states and the max depth reached for each of the checker types for execution on the test suite over the entire search space.

Stellenbosch University https://scholar.sun.ac.za

4.2.1.1 The effect of the code instrumentation on the model checker's search space

The model checker's search space size is influenced by the number of bytecode instructions included in the SUT for two reasons: 1. for an increase in the number of bytecode instructions in the SUT, the maximum depth reached and path length increases and 2. an increase in the number of bytecode instructions in the SUT causes an increase in the number of bytecode interleavings.

Figures 4.1 and 4.2 visually illustrate the execution times of the concrete checkers for the LockFreeSet SUT data of Table 4.3. The memory data follows the same trends as execution time because the time to traverse a search space is directly proportional to the size of the search space.

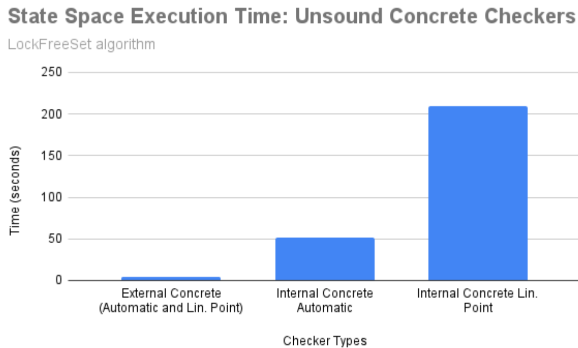


Figure 4.1: **Unsound Concrete Checkers:** Execution time results (from Table 4.3) for entire search space traversal of the test suite LockFreeSet algorithm.

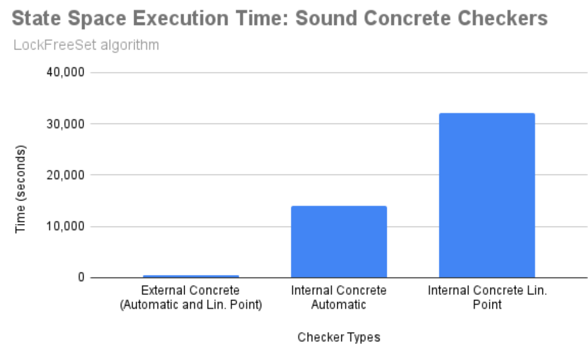


Figure 4.2: **Sound Concrete Checkers:** Execution time results (from Table 4.3) for entire search space traversal of the test suite LockFreeSet algorithm.

4.2.1.1.1 Cost of including code instrumentation for soundness in the SUT To guarantee soundness of linearisability, code instrumentation is added to the SUT at positions targeted at preventing JPF from making state-space optimising branch cut-offs. The code instrumentation necessary for soundness, effects the execution time results for two reasons: First, as previously mentioned, the instrumentation adds bytecode instructions to the SUT which increases the path length and the number of interleavings in the search space; and a larger state space takes longer to traverse. Second, the instrumentation prevents JPF from cutting off branches that contain concurrent history events, which would have otherwise been cut off, so that no linearisations are missed. This also increases the state space. Figures 4.1 and 4.2 illustrate the cost of soundness, the axis numbers show that the sound checkers generally take one or two orders of magnitude longer to

Stellenbosch University https://scholar.sun.ac.za

execute than the otherwise equivalent unsound executions.

4.2.1.1.2 Cost of including code instrumentation for linearisability checking logic in the SUT The internal checkers require that the SUT is manually instrumented with linearisability checking logic. The amount of instrumentation required for linearisability checking is large in comparison to the other instrumentation requirements, thus its effect on the model checker's search space size and the tool's overall execution time is significant. As shown in Figures 4.1 and 4.2, the internal checkers consistently execute one or two orders of magnitude longer executions than their corresponding external implementation. The increase in resource requirements for the internal checkers is because the large amount of code instrumentation added to the SUT causes a large model checking search space, which takes longer to traverse. Many of the internal sound checker experiments reached the memory limit before completing their execution because of the combined internal and soundness instrumentation causing a very large search space.

The external checkers do not require that the SUT is instrumented with linearisability checking logic, and the benefit of this is evident in Table 4.3's data which shows that the external checkers create between one and three orders of magnitude less new states than the corresponding internal checkers. Figures 4.1 and 4.2 show that the external checkers have an execution time practically equivalent to the stand-alone model checking execution, and also show that the external checking computation contributes an insignificant amount of time to the tool's overall execution time.

4.2.1.1.3 Cost of including linearisation point code instrumentation in the SUT The external linearisation point checkers only require linearisation points to be manually specified in the SUT, no other instrumentation. The actual linearisability checking logic, executed when one of the linearisation points are encountered during JPF's search space traversal, is performed from the external listener class. Figures 4.1 and 4.2 show that the small amount of code instrumentation required for linearisation point placement, combined with the insignificant computation time required for the external linearisability checking logic, does not have any significant impact on the overall execution time of the tool; there is an insignificant execution time difference between the external linearisation point and the base model checking execution.

Stellenbosch University https://scholar.sun.ac.za

4.2.1.1.4 Cost of checking linearisability on-the-fly versus at the end-states, using instrumentation in the SUT JPF makes use of a state-space optimisation technique called Partial Order Reduction (POR). POR groups together sets of instructions that are allocated to a single thread, and that do not affect anything outside of the thread itself, to execute within a single transition and not interleave with the instructions of other threads.

The internal automatic checker performs linearisability checking logic only at program end states; after all spawned threads have finished their execution, after the Thread.join operation completes for Java, the main thread uses the concurrent history information maintained throughout the execution of that path to perform linearisability checks. The main thread, as the only final executing thread, does not affect anything outside of the thread itself and JPF groups together these instructions into a single transition. Thus the bytecode instructions contributing to the automatic linearisability checking computation do not contribute to any increase in bytecode interleavings or significant path length increase for the search space.

The internal linearisation point checker uses each spawned thread to performs on-the-fly linearisability checks which are able to affect other threads, thus the linearisability checking logic for this checker contributes to an increase in possible interleavings and path length; resulting in larger search space which takes longer to traverse. The data in Table 4.3 confirms this relation in that the internal linearisation point checkers generally produce many more new states during execution, compared to the internal automatic linearisation point checkers even though similar linearisability checks are taking place.

In summary, there is not just a correlation between the amount of code instrumentation added to the SUT and the execution time but also the kind of instrumentation used for each respective checker. We have investigated the impact that soundness instrumentation, linearisability checking logic instrumentation, linearisation-point specifying instrumentation, and on-the-fly versus end-state instrumentation has on the execution time of the checkers.

4.2.1.2 Performance-limiting Factor of the Concrete Checkers

To determine the performance limiting factor of each checker, we performed two types of experiments for each checker, the first where the linearisability checking logic is turned on and the second

Stellenbosch University https://scholar.sun.ac.za

where it is turned off; so that the resource usage requirements of stand-alone model checker and of the linearisability checker components can be determined. The experiment result data is shown in Tables 4.3 and 4.4 in the columns labeled with "Lin. Checker turned on/off". The results will be used to confirm/falsify the claims made by Vechev et al., Liu et al., and Doolan et al.

Figure 4.3 shows the execution time results of Table 4.3 for the external automatic and external linearisation point checkers (the first two bar graphs from the left of the figure, displayed in green), the internal linearisation point checker (the third and fourth bar charts from the left of the figure, displayed in orange), and the internal automatic checker (the fifth, sixth, and seventh bar charts from the left of the figure, displayed in red).

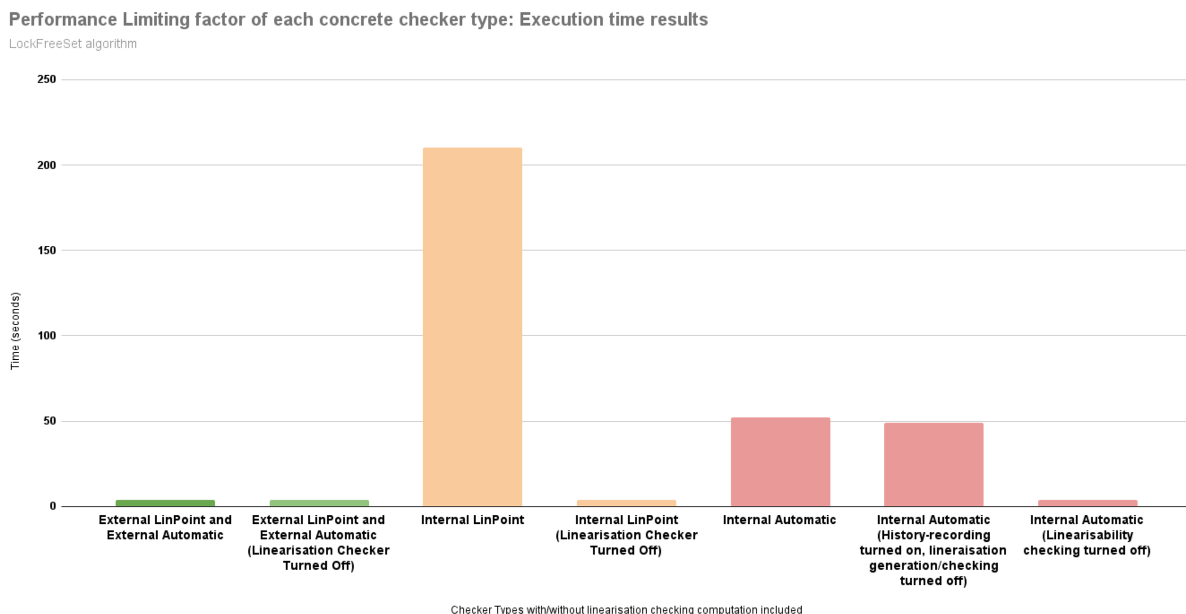


Figure 4.3: **Concrete checker performance-limiting-factor**. The execution time results used are for the LockFreeSet algorithm; the data is available in Table 4.3. The results show the execution time taken for the internal/external unsound automatic/linearisation-point checker types both with and without linearisability logic included.

The internal linearisation point checker includes linearisability-checking-logic code instrumentation in the SUT and as previously discussed, this instrumentation increases the search space size and thus the total execution time of the tool. The execution of this checker with linearisability checking turned off, that is the un-instrumented SUT, is shown in Figure 4.3 to take roughly five seconds,

Stellenbosch University https://scholar.sun.ac.za

the execution of the checker with linearisability checking turned on, that is the instrumented SUT, takes roughly 210 seconds; a significant difference which illustrates that **the amount of code instrumentation added to the SUT is the performance limiting factor of the internal linearisation point checker**.

For the internal automatic checker, instead of just a basic on/off for linearisability checking we also separate the instrumentation that maintains concurrent history information along the path from the logic that performs linearisation generation and checking. We perform three experiments: 1. where both history recording instrumentation and linearisation generation/checking instrumentation is added to the SUT (linearisability checking turned on), 2. where history recording instrumentation is included in the SUT but linearisation generation/checking logic is not, and 3. where no code instrumentation is added to the SUT (linearisability checking is turned off). The experiment results are shown as red bars in Figure 4.3. The results show similar execution times for the fully instrumented checker, roughly 52 seconds, and the same checker but with the linearisation generation and checking instrumentation excluded, roughly 49 seconds; a small difference but no significant change. The experiment with all instrumentation excluded shows a significantly shorter execution than the other two versions, roughly five seconds compared to 49 and 52 seconds. Thus it is clear that **the performance limiting factor of the internal automatic checker is the concurrent-history-recording code instrumentation added to the SUT.**

Various authors have made claims about the performance limiting factor of the internal automatic checker. Vechev et al. and Doolan et al. claim that the performance limiting factor of their internal automatic linearisability checker is the state space explosion caused by the code instrumentation added to the SUT; in the words of Vechev et al. "every time we append an element into [*sic*] the history, we introduce a new state" [39]. The results of our experiments confirm Vechev et al.'s evaluation of the performance limiting factor. However, we have determined that it is more specifically the concurrent-history-recording code instrumentation that is the performance limiting factor.

Liu et al., however, describe the performance limiting factor for Vechev et al.'s checker to be the linearisability checking computation, stating that the exponential worst-case time complexity of the automatic linearisability checking process is the cause of the poor scaling; in their words referring

Stellenbosch University https://scholar.sun.ac.za

to Vechev et al.'s approach "Their approach needs to find a linearisable sequence for each history, whose worst-case time is exponential in the length of the history, as it may have to try all possible permutations of the history." [23]. Our experiments falsify the claims made by Liu et al.; no noticeable change in execution time was found with the instrumentation for linearisability checking computation included or excluded.

Doolan et al. propose an external automatic checker that uses code instrumentation to output concurrent history records to an external log, and then perform linearisability checking for the logged histories using a tool external to the model checker's execution. The only code instrumentation added to the SUT is the logging instrumentation and because the history records are not kept in the model checker's search space, but output to an external log, the search space is not significantly impacted by the required instrumentation and the performance limiting factor is eliminated. Their experiment results showed that the optimised version executed about three times faster than the ordinary checker version. They explained that due to an optimisation used by their model checker (SPIN) their external checker is unsound; a sound implementation is proposed but left for future work.

The external automatic checkers implemented in this thesis have improvements to the external checker proposed by Doolan et al.; we do not require any code instrumentation added to the SUT, and we have implemented the external checkers for both sound and unsound modes. Furthermore we have extended the idea of an external checker to the linearisation point checking strategy as well. **We presented an external linearisation point checker** which excludes all linearisability checking logic from the SUT except the fixed linearisation point specifications.

The external checkers eliminate the performance limiting factor of the internal versions, the amount of code instrumentation added to the SUT, and their externally computing linearisability checking component contributes an insignificant amount of resource usage to the tool's overall execution. The result data in Table 4.3, displayed for the LockFreeSet SUT in Figure 4.3, thus show that the execution times of the external checkers are effectively equivalent to those of the stand-alone model checking search. **The stand-alone model checking state-space generation is the external checker's performance-limiting factor.**

Stellenbosch University https://scholar.sun.ac.za

4.2.2 Resource usage of the Symbolic Checkers

In this section the resource usage of the hybrid and the symbolic checkers is evaluated with respect to execution time, memory usage, and the model-checking search space.

The time, memory usage, and model checking search space data of our experiments are shown in Table 4.5. Memory is represented as the number of states created during the model checking search and the maximum memory allocation necessary for the search; maximum memory allocation is adjusted dynamically by JPF. The model-checking search space information shows the number of visited, backtracked, and end states for the search and the maximum depth reached during execution. A depth limit is chosen so that the experiments finish within a reasonable time; the symbolic linearisability checker's scalability is analysed in Section 4.3. Notice that the number of visited states for all symbolic or hybrid experiments is zero, this is because the visited states field represents the number of re-visited states and SPF does not use state hashing to re-visit states. The data in

The execution time results for the concrete, hybrid, and symbolic checker experiments, in Table 4.5, are illustrated in Figure 4.4. The data in the table, for the columns labeled "Concrete" and a depth limit of 17, show that the execution time of the concrete checker for all SUTs is at most just over one second; thus the yellow bars representing the concrete checker's execution times are not visible in Figure 4.4. The model-checking state space data in the table shows similar proportions to the execution time results because the time to traverse the search space is proportional to the size of the search space.

It is clear that the execution time of the concrete checker is consistently shorter than that of the hybrid checker, which is consistently shorter than that of the symbolic checker. The concrete checker explores a search space for a specific test case, including exact operations and argument values. The hybrid checker explores a more generic search space for an exact test case, but generates argument values for all reachable program paths. Finally, the symbolic checker explores an even more general search space for all possible test cases and program paths given an integer constraint on the number of operations per thread. The generality of each checker's search can be expressed by the following relation: $Concrete \subseteq Hybrid \subseteq Symbolic$. **The key advantage of the symbolic checker is its exhaustive search and ability to verify the linearisability of a data structure in general,**

Stellenbosch University https://scholar.sun.ac.za

instead of for particular test cases. The symbolic checker, however, scales badly because of the large search space size which takes longer to traverse. **It is not practical to use the symbolic checker unless a depth limit is imposed**, see Section 4.3 for details on the symbolic checker's scalability.

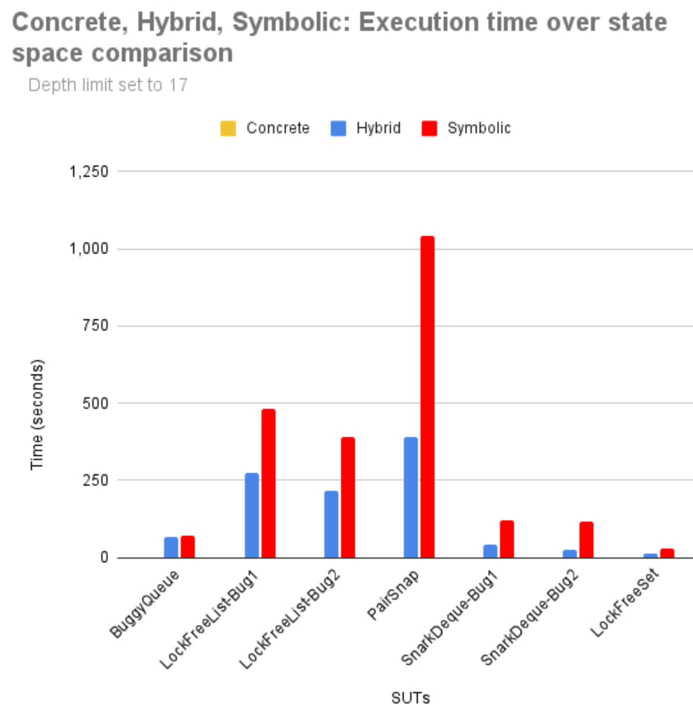


Figure 4.4: **Concrete, Hybrid, and Symbolic checker execution time comparison.** The results show the execution time for traversal of the entire search space, given a depth limit of 17, of each of the test suite algorithm examples for both the Hybrid and the Symbolic checker types. The concrete checker types complete their execution of all test suite algorithms in under one second, thus the yellow bars for the concrete checker are not visible in the figure.

An internal and/or sound symbolic linearisability checker is impractical for our current machine constraints. In Section 4.3 it is discussed that the external unsound hybrid and external unsound symbolic checkers have scalability problems. In Section 4.2.1.2 it was discussed that the internal/sound concrete checkers are significantly less efficient than their external/unsound concrete counterparts, respectively. It is therefore expected that the external unsound hybrid/symbolic checker, which already has scalability problems, will scale even more severely if executed in a sound setting or for an internal implementation; for the same reasoning given in Section 4.2.1.2 for the

Stellenbosch University https://scholar.sun.ac.za

SUT Algorithm	Data Format:	Concrete time: +-10s	Concrete time: +-1s	Hybrid time: +-5s	Symbolic time: +-5s
		Depth limit:			
		none	17	17	17
BuggyQueue	Time (seconds):	776	1.06	66	72
	New States:	3,272,426	606	544,567	691,045
	Max Mem. (MB):	674	238	425	424
	Visited:	4,983,342	451	0	0
	Backtracked:	8,255,768	1,057	544,567	691,045
	End:	136,985	11	43,183	11,064
LockFreeList Bug1	Time (seconds):	12,268	0.51	274	480
	New States:	46,885,440	249	2,307,370	4,636,177
	Max Mem. (MB):	1,126	238	1,459	1,465
	Visited:	67,193,481	277	0	0
	Backtracked:	114,078,921	526	2,307,370	4,636,177
	End:	248,472	0	43,740	204,323
LockFreeList Bug2	Time (seconds):	3,895	0.60	216	390
	New States:	15,111,856	249	2,141,446	3,857,415
	Max Mem. (MB):	739	238	1,448	1,467
	Visited:	21,801,688	277	0	0
	Backtracked:	36,913,544	526	2,141,446	3,857,415
	End:	88,956	0	3,888	204,011
PairSnap	Time (seconds):	1,056	1,12	391	1,040
	New States:	4,735,998	612	19,561,004	49,294,380
	Max Mem. (MB):	674	238	1,456	976
	Visited:	6,856,994	450	0	0
	Backtracked:	11,592,992	1,062	19,561,004	49,294,380
	End:	2,77,913	12	458,752	8,192
SnarkDeque Bug1	Time (seconds):	1,678	0.84	42	123
	New States:	7,178,784	598	301,548	2,305,033
	Max Mem. (MB):	675	238	674	675
	Visited:	10,519,554	448	0	0
	Backtracked:	17,698,338	2,046,	301,548	2,305,033
	End:	392,300	12	33,555	1,584
SnarkDeque Bug2	Time (seconds):	435	0.94	25	118
	New States:	1,946,860	597	170,062	2,231,852
	Max Mem. (MB):	676	238	424	675
	Visited:	2,877,073	450	0	0
	Backtracked:	4,823,933	2,047	170,062	2,231,852
	End:	93,815	11	22,344	1,530
LockFreeSet	Time (seconds):	541	0.56	12	29
	New States:	2,343,680	249	151,000	483,008
	Max Mem. (MB):	676	238	976	675
	Visited:	3,384,784	277	0	0
	Backtracked:	5,728,464	526	151,000	483,008
	End:	17,968	0	7,579	15,445

Table 4.5: **Concrete, Hybrid, Symbolic: Execution Time and Memory for Entire Search Space.** The execution time, number of new states, maximum memory, number of visited states, number of backtracked states, and number of end states for the concrete, hybrid and symbolic checkers, for execution on the test suite over the entire search space using no depth limit for the concrete checker and then a depth limit of 17 for all checkers.

Stellenbosch University https://scholar.sun.ac.za

concrete checkers. A proof-of-concept internal symbolic checker was implemented to check the BuggyQueue algorithm; the results showed that for even very small examples the execution time requirements exceed the timeout period. Similarly the soundness-guaranteeing symbolic checker implementation exceeded the timeout period for even small examples.

4.2.2.1 Performance-limiting Factor of the Symbolic Checkers

The performance limiting factor of the external concrete checkers is the stand-alone model checking state-space generation process, and the performance limiting factor of the internal concrete checkers is amount and type of code instrumentation added to the SUT; see Section 4.2.1.2. The symbolic and hybrid checkers are external implementations, it is expected that their performance-limiting factor would thus be the model checker's execution; as for the concrete external checkers. The symbolic linearisability checker and hybrid linearisability checker experiments with linearisability checking turned on and turned off show practically equivalent time to traverse the search space. The linearisability checking logic is thus shown to contribute an inconsequential amount of computation time to the tool's overall execution time; we conclude that **the performance limiting factor of the external symbolic checker is the stand-alone model checking state-space generation process; just as for the external concrete checkers**.

4.2.3 Optimisation techniques for the Concrete Automatic Checkers

The experiments in this section were performed to determine the effectiveness of the two different automatic checking optimisation techniques implemented and also measure their ability to improve the efficiency of the automatic linearisability checking tools.

- We do not expect that the lazy read optimisation to Vechev et al.'s automatic checking strategy [38], proposed by Long et al. [24], will significantly improve the tool's overall efficiency; it focuses on the linearisability checking segment of the tool's computation and as we showed in Section 4.2.1.2, the linearisation checking segment is insignificant compared to the model checking segment. Long et al. only reported experiment results on the efficiency benefits of the optimisation on pre-generated linearisations and not on the impact of the lazy read optimisation on the linearisability checking tool as a whole, which includes the state space generation of the model checker.

Stellenbosch University https://scholar.sun.ac.za

- We also do not expect the hash optimisation that prevents re-computation of previously checked but re-generated concurrent history paths, proposed in Section 3.2.1.2, to improve the tool's overall efficiency significantly because it too focuses on the linearisability checking segment of the tool's computation.

Experiments were run for the external (Tables 4.6 and 4.7) and internal (Tables 4.8 and 4.9) implementations of the concrete automatic checker, over the entire search space for each of the algorithms in the test suite. The "Time" columns for each table has a greater-than or smaller-than sign associated with each entry in the table, this indicates if the optimised checker experiment resulted in a benefit or a detriment to execution time when compared to the same experiment with the optimisation turned off. Four different optimisation combinations were used:

1. Vanilla – Ordinary Concrete Automatic Checker execution with no optimisations.
2. Lazy Read – Only the lazy read optimisation described in Section 3.2.1.1 is included.
3. Hash – Only the hash optimisation described in Section 3.2.1.2 is included.
4. Hash and Lazy Read – Both the hash and lazy read optimisations are included.

4.2.3.1 Effectiveness of the Lazy Read optimisation

The lazy read optimisation aims to reduce the number of linearisations generated during the linearisability checking process for each individual concurrent history. The "Number of generated linearisations" results in each of the tables show that **lazy read effectively reduced the number of generated linearisations in all experiments.** For example, in Table 4.8 for the "Number of generated linearisations" columns and the BuggyQueue SUT, the number of generated linearisations is reduced from 2,804 to 1,816 for the Vanilla versus Lazy Read experiments, respectively.

Stellenbosch University https://scholar.sun.ac.za

	External Unsound Hash and Lazy Read Optimisation Results for the Automatic Checker											
	Time (seconds) small variances in time occur experiment-to-experiment				**Number of generated linearisations**				**Number of concurrent histories checked for linearisability**			
	Vanilla	**Lazy Read**	**Hash**	**Lazy Read and Hash**	**Vanilla**	**Lazy Read**	**Hash**	**Lazy Read and Hash**	**Vanilla**	**Lazy Read**	**Hash**	**Lazy Read and Hash**
BuggyQueue	4.05	(>) 4.27	(<) 3.94	(>) 4.76	764	577	27	20	398	398	12	12
LockFreeList Bug1	5.41	(<) 5.20	(>) 5.54	(>) 6.54	32	24	8	6	12	12	3	3
LockFreeList Bug2	4.99	(>) 5.01	(>) 5.73	(>) 5.51	16	12	4	3	8	8	2	2
PairSnap	2.48	(>) 2.89	(>) 3.14	(>) 4.25	263	116	29	19	154	154	15	15
SnarkDeque Bug1	4.19	(<) 4.04	(>) 4.46	(<) 4.14	841	646	78	58	350	350	29	29
SnarkDeque Bug2	3.40	(>) 3.56	(>) 3.99	(<) 3.32	511	343	37	26	220	220	13	13
LockFreeSet	1.24	(>) 1.74	(>) 1.80	(>) 2.11	3	12	3	3	3	8	2	2

Table 4.6: Execution time, number of generated linearisations, and number of concurrent history paths checked for linearisability for the **external unsound** concrete automatic checker for execution on the test suite SUTs over the entire search space.

	External Sound: Hash and Lazy Read Optimisation Results for the Automatic Checker											
	Time (seconds) small variances in time occur experiment-to-experiment				**Number of generated linearisations**				**Number of concurrent histories checked for linearisability**			
	Vanilla	**Lazy Read**	**Hash**	**Lazy Read and Hash**	**Vanilla**	**Lazy Read**	**Hash**	**Lazy Read and Hash**	**Vanilla**	**Lazy Read**	**Hash**	**Lazy Read and Hash**
BuggyQueue	815	(<) 810	(<) 799	(<) 774	305,307	225,654	69	55	136,985	136,985	27	27
LockFreeList Bug1	11,797	(>) 12,317	(<) 11,654	(<) 11,761	857,520	593,952	562	350	248,472	248,472	130	130
LockFreeList Bug2	3,709	(>) 3,830	(<) 3,686	(>) 3,729	267,348	202,020	255	199	88,956	88,956	73	73
PairSnap	1,114	(>) 1,127	(<) 1,086	(<) 1,083	664,927	442,098	140	92	277,913	277,913	48	48
SnarkDeque Bug1	1,656	(>) 1,718	(<) 1,626	(<) 1,602	994,631	856,109	188	148	392,300	392,300	60	60
SnarkDeque Bug2	450	(<) 449	(<) 432	(<) 444	220,839	143,675	77	49	93,815	93,815	31	31
LockFreeSet	555	(<) 554	(<) 546	(<) 544	43,492	32,056	77	53	17,968	19,968	31	31

Table 4.7: Execution time, number of generated linearisations, and number of concurrent history paths checked for linearisability for the **external sound** concrete automatic checker for execution on the test suite SUTs over the entire search space.

Stellenbosch University https://scholar.sun.ac.za

	Internal Unsound: Hash and Lazy Read Optimisation Results for the Automatic Checker											
	Time (seconds) small variances in time occur experiment-to-experiment				**Number of generated linearisations**				**Number of concurrent histories checked for linearisability**			
	Vanilla	**Lazy Read**	**Hash**	**Lazy Read and Hash**	**Vanilla**	**Lazy Read**	**Hash**	**Lazy Read and Hash**	**Vanilla**	**Lazy Read**	**Hash**	**Lazy Read and Hash**
BuggyQueue	110	(>) 116	(>) 111	(>) 115	2,804	1,816	17	13	1,024	1,024	6	6
LockFreeList Bug1	899	(>) 902	(<) 863	(<) 569	17,993	7,456	62	33	6,464	6,464	13	13
LockFreeList Bug2	209	(>) 213	(<) 199	(<) 200	5,238	3,278	35	21	1,940	1,940	9	9
PairSnap	232	(>) 234	(<) 226	(<) 226	6,864	3,928	19	13	2,256	2,256	7	7
SnarkDeque Bug1	371	(>) 376	(<) 362	(<) 354	9,856	7,622	24	19	3,024	3,024	8	8
SnarkDeque Bug2	122	(>) 124	(<) 120	(<) 121	2,892	1,716	17	11	1,104	1,104	6	6
LockFreeSet	562	(>) 578	(<) 523	(<) 547	2,284	1,600	14	10	896	896	6	6

Table 4.8: Execution time, number of generated linearisations, and number of concurrent history paths checked for linearisability for the **internal unsound** concrete automatic checker for execution on the test suite SUTs over the entire search space.

	Internal Sound: Hash and Lazy Read Optimisation Results for the Automatic Checker											
	Time (seconds) small variances in time occur experiment-to-experiment				**Number of generated linearisations**				**Number of concurrent histories checked for linearisability**			
	Vanilla	**Lazy Read**	**Hash**	**Lazy Read and Hash**	**Vanilla**	**Lazy Read**	**Hash**	**Lazy Read and Hash**	**Vanilla**	**Lazy Read**	**Hash**	**Lazy Read and Hash**
BuggyQueue	14,954	(<) 14,550	(<) 14,106	(<) 14,088	552,276	391,716	47	38	245,988	245,988	27	27
LockFreeList Bug1	timeout				timeout				timeout			
LockFreeList Bug2	timeout				timeout				timeout			
PairSnap	out of memory				out of memory				out of memory			
SnarkDeque Bug1	out of memory				out of memory				out of memory			
SnarkDeque Bug2	15,914	(>) 16,307	out of memory	out of memory	612,468	399,924	out of memory	out of memory	279,972	279,972	out of memory	out of memory
LockFreeSet	14,015	(>) 14,051	out of memory	out of memory	693,008	513,872	out of memory	out of memory	332,096	332,096	out of memory	out of memory

Table 4.9: Execution time, number of generated linearisations, and number of concurrent history paths checked for linearisability for the **internal sound** concrete automatic checker for execution on the test suite SUTs over the entire search space.

Stellenbosch University https://scholar.sun.ac.za

4.2.3.2 Effectiveness of the Hash optimisation

The model checker generates all possible bytecode interleavings for the SUT, but multiple bytecode interleavings may reduce to the same concurrent history. The hash optimisation prevents re-computation of already checked concurrent histories by maintaining a hashtable and ensuring each unique concurrent history is only checked once during execution.

The "Number of concurrent histories checked for linearisability" results in each of the tables show that the hash optimisation considerably reduced the number of histories checked, from thousands to tens in the internal cases of Tables 4.8 and 4.9 and hundred to tens in the external cases of Tables 4.6 and 4.7.

Interestingly, as a side-effect of reducing the number of histories checked, **the hash optimisation considerably reduces the overall number of linearisations generated during execution; even more so than the lazy read optimisation** that has as its main aim the reduction of the number of generated linearisations. For example, in Table 4.8 for the "Number of generated linearisations" columns and the BuggyQueue SUT, the number of generated linearisations is reduced from 2,804 to just 17 for the Vanilla versus Lazy Read experiments, respectively; much more effective than the 2,804 to 1,816 reduction of the lazy read optimisation.

4.2.3.3 Effects of the optimisations on the checker's overall time efficiency

As expected, the table results show that **neither the lazy read or the hash optimisation have any significant impact on the overall execution time of the linearisability checking tools.** The optimisations focus on the linearisability checking component of the tool's execution and not the performance limiting factor, the model checking component.

The execution time results for all external experiments, shown in Tables 4.6 and 4.7, with and without optimisations, have similar running times with no noticeable benefit for the optimised versions. There does, however, seem to be a slight benefit for the hash optimisation on all experiments for the external sound checkers, but the actual speed-up is inconsequential compared to overall running time.

The "Time" columns of Tables 4.8 and 4.9 display the results for the internal unsound experiments, and the results for the internal sound experiments that finished. The results show that the lazy read

Stellenbosch University https://scholar.sun.ac.za

optimisation is mostly disadvantageous to the tool's overall execution time. For our experiments, the computational overhead of the lazy read optimisation, included as code instrumentation in the SUT, outweighs its benefit of generating fewer, but more complex, linearisations. The table results also show that the hash optimisation results in a benefit to execution time for most experiments. As discussed, the hash optimisation more effectively reduces the linearisability checking computation than lazy read does, and our experiment results show that the computational saving from the optimisation does outweigh the computation cost of the optimisation; but because it still does not focus on the performance limiting factor of the tool, the benefit is not significant.

Long et al., who proposed lazy read, found a speed-up for their experiments, but they perform efficiency experiments on pre-generated concurrent histories and thus do not test the efficiency benefit on the overall tool but just the linearisability checking segment of the tool. Additionally, their optimisation will perform better for experiments with large numbers of read operations, but our experiments mostly contain write operations, minimising lazy read's ability to improve efficiency; see Section 3.2.1.1 for more details on the lazy read optimisation and the reason it will provide more benefit in read-heavy situations.

4.2.4 Efficiency Conclusions

In this section the resource-usage requirements of the checkers was compared, the performance limiting factor of each checker determined, and the effectiveness and benefit of the two automatic checker optimisations evaluated.

The analysis showed that there is a correlation between the amount of code instrumentation included in the SUT and the execution time of the checkers. Further investigation showed that the execution time is not just influenced by the amount of instrumentation, but also the kind of instrumentation used for each respective checker. The impact that soundness instrumentation, linearisability checking logic instrumentation, linearisation-point specifying instrumentation, and on-the-fly versus end-state instrumentation has on the execution time of the respective checkers was evaluated.

The performance limiting factor of the internal linearisation point checker was determined to be the code instrumentation added to the SUT. The performance limiting factor of the internal automatic

Stellenbosch University https://scholar.sun.ac.za

checker was shown to be specifically the concurrent-history-recording code instrumentation added to the SUT. Finally, the performance limiting factor of both the concrete and symbolic external checkers was determined to be the stand-alone model checking state-space generation process.

The key advantage of the symbolic checker is its exhaustive search and ability to verify the linearisability of a data structure in general, instead of for particular test cases like the concrete checker. However, the generality of the symbolic checker's search causes it to scale badly. It was concluded that the symbolic checker is not practical unless a depth limit is imposed.

It was clear from the experiment results that the lazy read and hash optimisations do not have any significant impact on the overall execution time of the linearisability checking tools, despite effectively reducing their aimed linearisability checking computation. This is because the optimisations focus on the linearisability checking component of the tool's execution and not the performance limiting factor, the model checking component.

4.3 SCALABILITY

To determine the scalability of the concrete and symbolic checkers, experiments were performed on the external checker implementations. The scalability of JPF and SPF are the direct performance limiting factors of the external checkers. The performance limiting factors of the internal checkers are the model checkers and the code instrumentation which adds more information to the model checker's search space. Thus the internal checkers will scale at least as badly as the external implementations and worse for the code instrumentation added to the SUT.

Experiments were run for an increase in 1. the number of operations per thread, 2. the number of executing threads, and 3. the depth limit while keeping the others constant. The experiments for this section exclusively use the LockFreeSet test suite algorithm, which does not contain any linearisation errors.

4.3.1 Scaling of the Concrete Checkers

Experiments were run to determine the external concrete checker's scalability for one, two, and three SUT threads and for each case the number of operations executed per thread was increased from 1 to 16. Table 4.10 shows the results in terms of the number of unique histories explored, execution

Stellenbosch University https://scholar.sun.ac.za

time for full state space exploration, and the number of end states reached during execution for both sound and unsound execution settings. A "T" in the table denotes an experiment that timed out before finishing traversal of the search space.

4.3.1.1 Unique histories generated and execution time:

For one thread, the number of unique histories remains constant for an increasing number of operations per thread. This is an intuitive result because there is only one interleaving for a serial program, irrespective of how many operations are in that program. The execution time increases slightly for each step of increased number of operations. Each additional operation increases the path length, and thus takes slightly longer to traverse than the step before; but because the program is serial there is still only one interleaving to traverse so the execution time remains relatively short.

For two threads, the unsound experiments show an exponential increase in the number of unique histories traversed and a corresponding exponential trend in the execution time of the tool. The sound experiments show the same result but with a more drastic exponential trend that causes the timeout limit to be reached for experiments with more than three operations per thread. The reason the scaling of the two-thread example is so much worse than the serial example is because the second thread allows the model checker to generate all interleavings, which is exponential in the number of bytecode instructions. For each step that increases the number of operations, the number of bytecode instructions increase and thus the number of interleavings which takes exponentially longer to traverse; as shown in Table 4.10.

For three threads the sound checker times out on all experiments and the unsound checker times out for six or more operations per thread. The addition of a third thread results in an even higher exponential growth in interleavings, for the same number of operations, over the two or one thread examples. **It is clear that because the model checking concurrent history generation process itself does not scale, and that this process is the performance limiting factor of the external checkers, the checking tools also do not scale.**

Stellenbosch University https://scholar.sun.ac.za

Number of operations per thread	Operations where each [add,remove] pair uses a unique parameter value	(for 1 , 2, and 3 executing threads) Number of unique histories						Time taken to compute (seconds)						Number of end states reached					
		Unsound			Sound			Unsound			Sound			Unsound			Sound		
		1	**2**	**3**	1	**2**	**3**	**1**	**2**	**3**	1	**2**	**3**	**1**	**2**	**3**	1	**2**	**3**
1	***add(param1)***	1	5	35	1	6	**T**	0.21	1	39	0.46	57	**T**	3	4	5	3	1,392	**T**
2	***add(param1), remove(param1)***	1	21	408	1	70	**T**	0.33	2	389	0.69	17,579	**T**	3	4	5	3	402,872	**T**
4	***[add(param1), remove(param1)]×2***	1	97	7781	1	**T**	**T**	0.45	9	5,570	1.10	**T**	**T**	3	4	5	3	**T**	**T**
6	***[add(param1), remove(param1)]×3***	1	253	**T**	1	**T**	**T**	0.54	23	**T**	1.41	**T**	**T**	3	4	**T**	3	**T**	**T**
8	***[add(param1), remove(param1)]×4***	1	513	**T**	1	**T**	**T**	0.69	53	**T**	1.50	**T**	**T**	3	4	**T**	3	**T**	**T**
10	***[add(param1), remove(param1)]×5***	1	901	**T**	1	**T**	**T**	0.81	97	**T**	1.73	**T**	**T**	3	4	**T**	3	**T**	**T**
12	***[add(param1), remove(param1)]×6***	1	1,441	**T**	1	**T**	**T**	0.88	178	**T**	2.09	**T**	**T**	3	4	**T**	3	**T**	**T**
14	***[add(param1), remove(param1)]×7***	1	2,157	**T**	1	**T**	**T**	0.89	338	**T**	2.21	**T**	**T**	3	4	**T**	3	**T**	**T**
16	***[add(param1), remove(param1)]×8***	1	3,073	**T**	1	**T**	**T**	0.99	479	**T**	2.79	**T**	**T**	3	4	**T**	3	**T**	**T**

Table 4.10: **External Concrete scaling results for an increase in the number of operations executed per thread for 1, 2, and 3 executing threads.** The results show the number of generated unique concurrent histories, execution time, and number of end states reached during execution for each of the the external unsound/sound concrete checker types for execution of the LockFreeSet algorithm over the entire search space. The “T” stands for timeout.

Stellenbosch University https://scholar.sun.ac.za

4.3.1.2 End states reached:

Interestingly, for the experiments in Table 4.10 which run two threads, the unsound checker reaches only four distinct end states and the equivalent sound checker reaches an exponential number of distinct end states. The instrumentation for soundness, added to the SUT, forces JPF to differentiate between paths and create different end states for states it would otherwise consider equivalent; thus the increase in end states for the sound experiments over the unsound experiments. For example, the experiment data in Table 4.10 that runs two threads and uses two operations per thread shows that the unsound checker explores 21 unique histories but for the same experiment the sound checker explores 70 unique histories. This trend will generalise to the timed-out experiments but are not shown because of the timeout. It is concluded that though neither the sound nor unsound checkers scale, **the unsound checkers are more scalable than the sound checker** because the unsound checker's search space is a subset of the sound checker's search space.

4.3.2 Scaling of the Symbolic Checker

Due to the exhaustive nature of the symbolic checker's search, we expect the symbolic checker to scale badly. Experiments were run to determine the external symbolic checker's scalability for an increasing number of 1. operations per thread, 2. executing threads, and 3. depth limit; where, for each experiment, two of the three parameters remain constant with the third increased.

Combinatoric principles are used to determine the number of expected test cases generated by the symbolic checker. We investigate two factors that affect the number of possible test cases: the number of methods in the SUT, and the number of operations executing per thread.

4.3.2.1 Expected increase in the number of test cases and its effect on scalability

Each SUT has a number of methods which act on the data structure. For example a Queue data structure has two methods types namely `enqueue` and `dequeue`, and a Set data structure has three methods namely `add`, `remove`, and `contains`. Since the symbolic checker generates all possible operation sequences for the SUT methods, the number of methods directly impacts the number of possible test cases to be generated and tested.

Keeping the number of operations constant, say two operations executed by a single thread, and

Stellenbosch University https://scholar.sun.ac.za

Number of unique test cases for a single threaded execution

Number of operations to be executed (N)	1 SUT method (1^N)	2 SUT methods (2^N)	3 SUT methods (3^N)
1	1	2	3
2	1	4	9
4	1	16	81
6	1	64	729
8	1	256	6561
10	1	1024	59049
12	1	4096	531441
14	1	16384	4782969
16	1	65536	43046721

Table 4.11: **Symbolic Checker:** Data showing the exponential increase in the number of test cases for a single thread execution of a SUT with 1/2/3 methods when the number of operations for the thread to execute is increased

increasing the number of methods for example from one to six, we get the number of test cases as 1, 4, 9, 16, 25, 36 etc. Thus there is an exponential increase in the number of test cases for increased methods in the SUT. Letting the number of operations be slightly more, say 4 operations executed by a single thread, the number of test case possibilities becomes 1, 16, 81, 256, 625, and 1,296; thus the rate of the exponential trend for increased methods, is increased for increased number of operations per thread. Thus **an exponential increase in the number of test cases is expected for an increasing number of methods in the SUT**.

Table 4.11 shows scaled number of operations per thread results for SUT situations with one, two, and three methods. It is evident that for 2 and 3 methods there is an exponential increase in the number of test cases for an increasing number of operations per thread, the rate of exponential growth is also shown to grow for increased number of methods. In summary, **we expect an exponential increase in test cases for an increasing number of operations per thread.**

4.3.2.2 Actual increase in the number of test cases and its effect on scalability

We performed experiments for the symbolic checker using one thread and an increasing number of operations: from one to sixteen. We performed these number-of-operation scaling experiments for depth limit situations of fourteen, fifteen, sixteen, and seventeen. We did not perform experi-

Stellenbosch University https://scholar.sun.ac.za

ments for larger number of threads because a serial execution already necessitates a depth limit an increased number of threads will only worsen the scalability by way of an exponential increase in interleavings.

Table 4.12 shows the scalability results in terms of the number of unique histories explored, execution time for full state space exploration and the number of end states reached during execution, for both sound and unsound execution settings.

Number of operations per thread	**Number of expected unique histories for no depth limit** (excluding different parameter value options)	**Number of unique histories** (inclusive of cut-off histories and different parameter value options)				**Time taken to compute** (seconds)				**Number of end states reached**			
		Depth limit:				**Depth limit:**				**Depth limit:**			
		14	**15**	**16**	**17**	**14**	**15**	**16**	**17**	**14**	**15**	**16**	**17**
1	2	2	2	2	2	0.15	0.15	0.19	0.18	17	17	17	17
2	4	6	6	6	6	0.24	0.29	0.27	0.25	69	76	83	91
4	16	70	70	70	70	0.72	0.87	1.08	0.94	932	1,312	1,696	2,080
6	64	378	524	696	792	1.22	122.48	843.06	2,644.05	516	1,282	3,032	6,812
8	256	532	884	1,422	2,174	1.32	122.58	843.51	2,765.61	86	268	750	2,018
10	1,024	558	964	1,638	2,732	1.28	122.66	843.65	2,771.33	6	32	102	346
12	4,096	560	970	1,668	2,828	1.29	122.68	843.58	2,765.87	0	2	6	36
14	16,384	560	970	1,670	2,834	1.31	122.51	843.49	2,766.52	0	0	0	2
16	65,536	560	970	1,670	2,834	1.32	122.54	843.55	2,765.90	0	0	0	0

Table 4.12: **External Symbolic scaling results for an increase in the number of operations executed per thread for 1 executing thread.** The results show the number of generated unique concurrent histories, execution time, and number of end states reached during execution for the the external unsound symbolic checker types for execution of the LockFreeSet algorithm over the entire search space.

As expected, the experiment results confirm an exponential increase in the number of test cases as the number of operations per thread is increased. The data in the columns labeled "unique histories" of Table 4.12 corresponds to the generated test cases but also includes the different argument values for the test cases. Column 2 in Table 4.12 shows the number of expected test cases, the "Number of unique histories" columns show the actual number of generated test cases. Given an unbounded search space, the number of test cases and time taken to explore them, would increase indefinitely for an increase in the number of operations per thread; but when using a depth limit the search space size, and thus execution time, is bound. The effects of a depth limit on the search space will now be considered.

Stellenbosch University https://scholar.sun.ac.za

4.3.2.3 The effect of a depth limit on the search space

Figure 4.5 shows a graph of the execution time results for the data in Table 4.12. For each depth limit represented by a line graph there is an initial positive gradient indicating an increasing execution time as the number of executing operations per thread increases, but then each line reaches a plateau (at around six operations per thread) that remains constant for all larger number of operations per thread.

Figure 4.6 shows that, for the same set of experiments, the number of end states initially increases (until about six operations per thread) and then decreases gradually to zero end states (for test cases with ten or more operations per thread) for all experiments. We thus conclude that path cut-offs due to the depth limit are made for test cases with around five or more operations. Figure 4.5 shows us that the search space size remains constant from six operations per thread, because the depth limit now causes the breadth and depth of the search space size to remain constant for any increase in path length (caused by an increasing number of operations per thread); thus the constant search space size takes constant time to traverse.

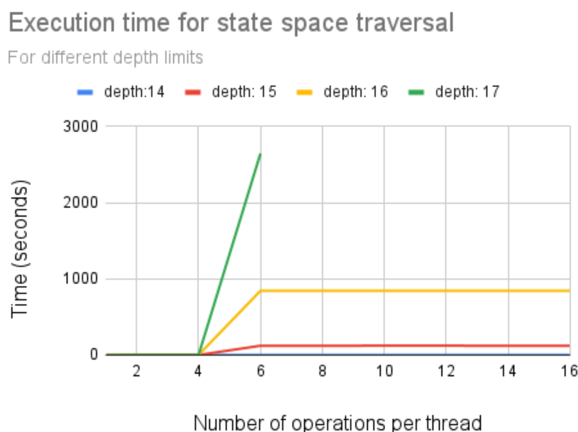


Figure 4.5: **Symbolic checker execution time for increasing number of operations executed per thread.** The experiments were run for the LockFreeSet SUT where the number of operations per thread are increased and situations for depth limits of 14, 15, 16, and 17 respectively; the data is taken from Table 4.12.

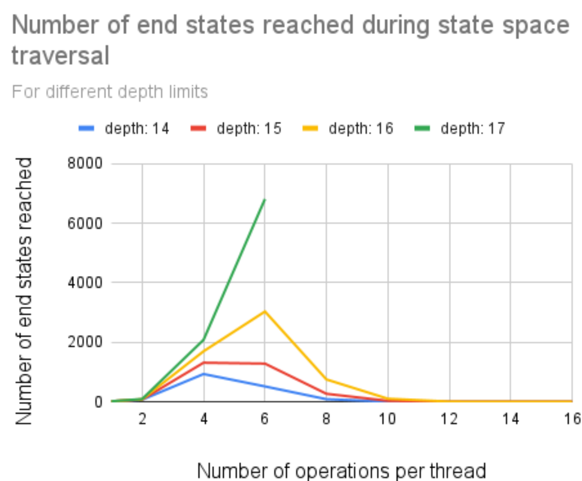


Figure 4.6: **Symbolic checker number of end states for increasing number of operations executed per thread.** The experiments were run for the LockFreeSet SUT where the number of operations per thread are increased and situations for depth limits of 14, 15, 16, and 17 respectively; the data is taken from Table 4.12.

Stellenbosch University https://scholar.sun.ac.za

Figure 4.7 shows that the depth limit affects the expected exponential growth in the number of generated histories from about six operations per thread; six operations per thread is the point where the depth limit starts cutting off end states in Figure 4.6. The path cut-offs result in the number of generated histories reaching a plateau at around 10 operations per thread, where the number of end states reach zero in Figure 4.6. However, the generality of the symbolic checker's search means that for the search space up until the depth limit, a large variety of histories are generated by the model checker; and for each increase in the depth limit there is an exponential increase in the number of generated histories.

Figure 4.7 shows the effect of the depth limit on the number of histories generated as the number of operations for the test case increases. As discussed previously, without a depth limit the number of generated histories will increase exponentially for each increase in number of operations per thread; the depth limit prevents this from happening and results in a plateau from around ten operations per thread.

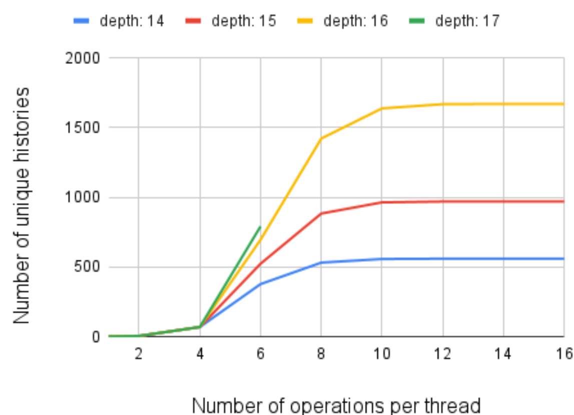


Figure 4.7: **Symbolic checker number of unique concurrent histories generated during execution for increasing number of operations executed per thread.** The experiments were run for the LockFreeSet SUT where the number of operations per thread are increased for depth limits of 14, 15, 16, and 17 respectively; the data is taken from Table 4.12.

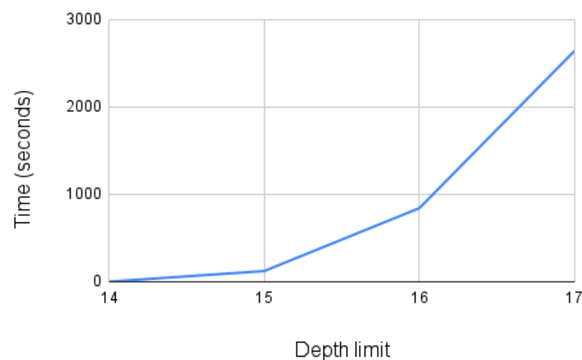


Figure 4.8: **Symbolic checker execution time results graph for increased depth limit**. The experiments were run on the LockFreeSet SUT where the depth limit is increased from 14 to 17 and the number of operations executed per thread kept constant at 16; the data is taken from Table 4.12.

Stellenbosch University https://scholar.sun.ac.za

In summary, the depth limit constrains the number of histories generated and thus the execution time of the symbolic checker. **For an increase in the depth limit, there is an exponential increase in the number of generated histories and also an exponential increase in the execution time cost to traverse these histories.**

4.3.3 Scalability Conclusions:

The model checking concurrent history generation process itself does not scale, and this process is the performance limiting factor of the external checkers, thus the checking tools also do not scale.

The **unsound concrete** checker has scaling problems for larger number of executing threads but is usable for one and two threads.

The **sound concrete** checker is only usable for one executing thread or two executing threads with small numbers of operations executed per thread. This checker does however explore more unique histories, for the same experiment, than its unsound counterpart and guarantees soundness with respect to linearisability for its execution. The sound execution thus provides a higher quality linearisability check than the unsound execution does, at the cost of a larger search space that takes longer to traverse.

The **symbolic checker** has severe scaling problems which can be managed by imposing a depth limit for the search. Although the depth limit prevents full state space traversal, this checker's ability to perform a very general search means that a large variety of histories are generated by the model checker up until the depth limit.

4.4 ERROR FINDING

To determine the error finding ability of the different checkers, each linearisability checker was executed on the test suite algorithms (Table 4.2) and results reported for the search space up until the errors. Efficiency is considered in terms of execution time, memory usage, and search space statistics up until the time of the first located error.

Table 4.13 shows the linearisation errors located by each checker and the time taken to locate the errors. The located errors are shown as green tick marks and the missed errors as black crosses.

Stellenbosch University https://scholar.sun.ac.za

SUT Algorithm	Data Format:	Concrete								Hybrid	Symbolic
		Unsound				Sound				External (Sound till depth limit)	External (Sound till depth limit)
		Internal		External		Internal		External			
		Aut.	Lin. Point	Aut.	Lin. Point	Aut.	Lin. Point	Aut.	Lin. Point		
BuggyQueue (unique-value)	Found error: Time (seconds):	✓ 3.78	✓ 140.72	✓ 0.45	✓ 0.36	✓ 81.17	✓ 331.82	✓ 2.52	✓ 2.36	✓ 0.56	✓ 1.09
BuggyQueue (duplicate-value)	Found error: Time (seconds):	✗	✗	✗	✗	✗	✗	✗	✗	✓ 0.56	✓ 1.09
LockFreeList Bug1	Found error: Time (seconds):	✓ 6.68	✓ 0.91	✗	✓ 0.53	✓ 433.6	✓ 3.77	✓ 30.38	✓ 32.46	timeout	timeout
LockFreeList Bug2	Found error: Time (seconds):	✓ 6.06	✓ 0.93	✗	✓ 0.49	✓ 408.95	✓ 3.59	✓ 31.19	✓ 33.48	timeout	timeout
PairSnap	Found error: Time (seconds):	✓ 0.98	✓ 0.75	✓ 0.18	✓ 0.15	✓ 7.21	✓ 3.29	✓ 1.51	✓ 1.6	✓ 20.47	✓ 11,644
SnarkDeque Bug1	Found error: Time (seconds):	✓ 8.44	✓ 1,071.1	✓ 0.42	✓ 0.43	✓ 577.06	memory limit reached	✓ 16.04	✓ 14.63	✓ 0.98	Bug1 ✓ 3,328.62
SnarkDeque Bug2	Found error: Time (seconds):	✓ 3.99	✓ 169.36	✓ 0.36	✓ 0.49	✓ 76.96	✓ 3855.43	✓ 2.29	✓ 2.22	✓ 0.79	Bug2 ✓ 79.43

Table 4.13: **Linearisability error finding ability for all checker types.** The results show the execution time taken until the error is found for each of the internal/external sound/unsound linearisation-point/automatic concrete checker types and the external unsound automatic hybrid and symbolic checker types. The table uses green ticks to denote errors that were successfully located and black crosses to denote errors that were not located.

Stellenbosch University https://scholar.sun.ac.za

4.4.1 Error finding of the Concrete Checkers

4.4.1.1 Unsound Checkers:

Comparing the results in the columns labelled under “Unsound” and “Sound” of Table 4.13, for the concrete checkers, it is clear that **the unsound checkers execute in a fraction of the time taken by the equivalent sound checker**. The unsound checkers explore a subset of the search space that the sound checkers do, thus less time is required to traverse the smaller search space.

The unsound linearisability checkers, however, **do not guarantee soundness with respect to linearisability**, they can therefore miss linearisability errors entirely. The results in Table 4.13 illustrate an example of this behaviour: the columns under the headings “Concrete” and “Unsound” show that unsound external automatic checker missing the two LockFreeList errors when and the sound version of this checker locates it.

Surprisingly, despite being unsound, the other three unsound concrete checkers (external linearisation point, internal automatic, and internal linearisation point) locate all test suite errors (see columns 1, 2, and 4 under the “Unsound” label for concrete checkers); bar the duplicate-value input which cannot be located by the concrete checkers (see Section 4.4.2 for details). These implementations add code instrumentation to the SUT, which is different from the unsound external automatic checker which does not add code instrumentation and does not find the two LockFreeList errors. This instrumentation is not targeted at guaranteeing soundness but it has the side effect of, for these examples, providing the model checker with enough information to resolve the soundness problem.

In summary, the **unsound concrete checkers find the linearisation errors most of the time** and execute in a fraction of the time taken by the sound checkers; but they do not guarantee soundness and cannot be used to prove that an error is not present for the input situation.

4.4.1.2 Sound Checkers:

Since sound linearisability checking guarantees that if a linearisation error is present then the checker will find it, it is not surprising that the experiment results show in the “Sound” columns of Table 4.13 that all errors in our test suite are found by all of the concrete sound checkers; bar the duplicate value case which concrete checkers cannot find (see Section 4.4.2 for details),

Stellenbosch University https://scholar.sun.ac.za

and experiments that did not finish. However, comparing the "Unsound" and the "Sound" result columns, it is evident that **there is a high execution time cost to guarantee soundness**.

4.4.2 Error finding of the Symbolic Checker

The concrete checker's usefulness in finding linearisability errors is constrained by the user's ability to hand-craft test cases in which errors are present. The symbolic checker performs linearisability checking on all possible test cases and execution paths, constrained only by the number of operations to be executed by each thread.

The symbolic checker verifies the linearisability of the data structure in general, but is only sound until its depth limit. The error-finding results in Table 4.13 show, in the column labelled "Symbolic", that **the symbolic checker was able to locate most of the test suite errors for general input; despite its scalability problems and depth limit requirement**. The experiment results show two other error-finding benefits of the symbolic checker over the concrete checker: the symbolic checker is able to locate linearisation errors in duplicate value situations, and the automatic test case generation allows multiple errors to be found in one run.

4.4.2.1 Duplicate value handling Benefits

The concrete checkers can miss linearisation errors in situations where input causes the data structure to contains duplicate values at one point in time. The concrete checkers compare the concrete response values of the SUT operations and the sequential oracle operations, but they do not consider which instance of the concrete value is being returned by the operation. The symbolic checker assigns unique symbols to each value in the SUT data structure, and it compares the symbol values returned by the SUT operations and the sequential oracle operations; thus it is sensitive to the instance of the concrete value being returned, not just the concrete value. See Section 3.5.3 for an in depth discussion of the duplicate-value problem for the concrete checkers, details of how the symbolic checker avoids this problem, and an example of a BuggyQueue SUT concurrent history for which the symbolic checker finds the error but the concrete checker misses it. The error-finding results in Table 4.13, see the column labelled "Symbolic", illustrate that **the symbolic checker locates a duplicate-value error in the BuggyQueue SUT which all concrete checkers miss**.

Stellenbosch University https://scholar.sun.ac.za

4.4.2.2 Automatic test case generation allows multiple errors to be found in one run

The symbolic checker uses automatic test case generation to all possible test cases for the input number of operations per thread. Thus **it is able to find multiple linearisation errors during a single run**. The column labelled “Symbolic” in Table 4.13 shows that both SnarkDeque errors were located during a run of the symbolic checker on general input. This is an example of the symbolic checker’s usefulness in finding linearisation errors with little user input, where the errors are not originally known.

4.4.3 Error finding of the Hybrid Checker

The hybrid checker was able to find most of the linearisation errors, the same errors found by the symbolic checker; see the columns labelled “Hybrid” and ‘Symbolic” in Table 4.13.

The hybrid checker makes use of symbols instead of concrete values and thus **is able to find the duplicate value error that the concrete checkers miss**. However, because it does not use automatic test case generation, it is not able to find multiple errors in one run as the symbolic checker can.

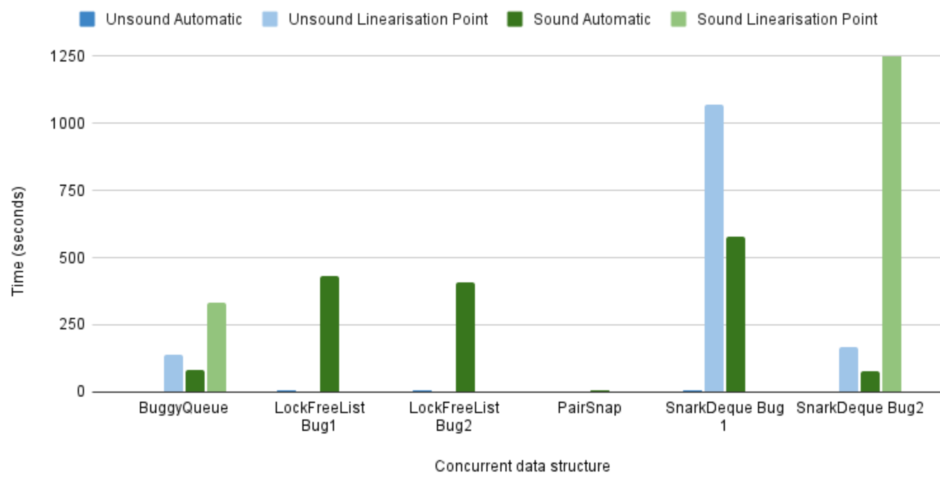
4.4.4 Error finding efficiency

In this section we compare the execution time taken by each checker up until they find the respective linearisability errors. The memory usage and search space statistics for the linearisation error finding executions can be found in Table C.1 of Appendix C. Figure 4.9, Figure 4.10, and Figure 4.11 show the execution time results for all internal concrete, external concrete, and symbolic checkers given in Table 4.13, respectively.

For the cases where the unsound checkers do find the linearisation errors, they are significantly more efficient at finding the errors than the equivalent sound checkers are. Figures 4.9 and 4.10 illustrate that the unsound checker took tens of seconds and sound checker took hundreds of seconds in the internal case, and that the unsound checker took seconds and the sound checker took tens of seconds in the external case.

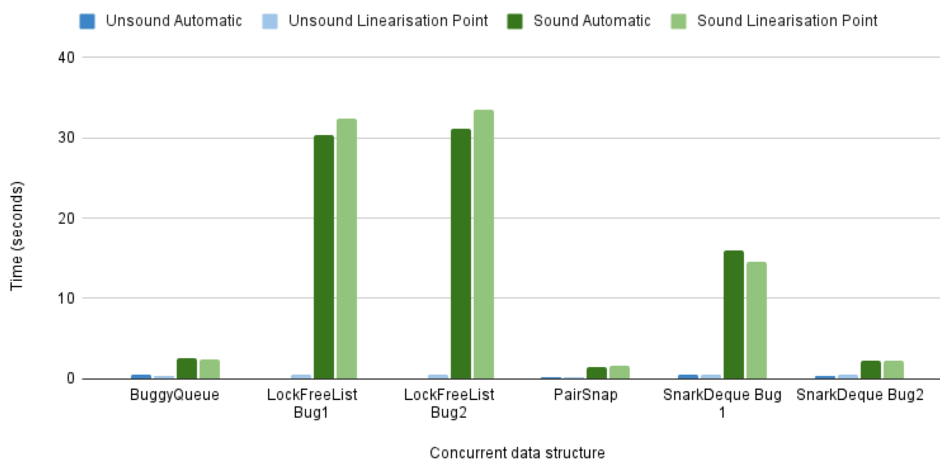
The external checkers find the errors consistently more efficiently than the internal checkers do. This is expected since the external tools have been shown to be considerably more

Stellenbosch University https://scholar.sun.ac.za



Internal Concrete Checker Types: *Sound linearisation point does not find SnarkDeque Bug1 and takes 3,855 seconds to find bug 2 (overflow after 1,250 seconds not shown in the figure); all other errors found.* Execution time results (from Table 4.13) for the time taken until the respective linearisation error was found for each of the test suite SUTs.

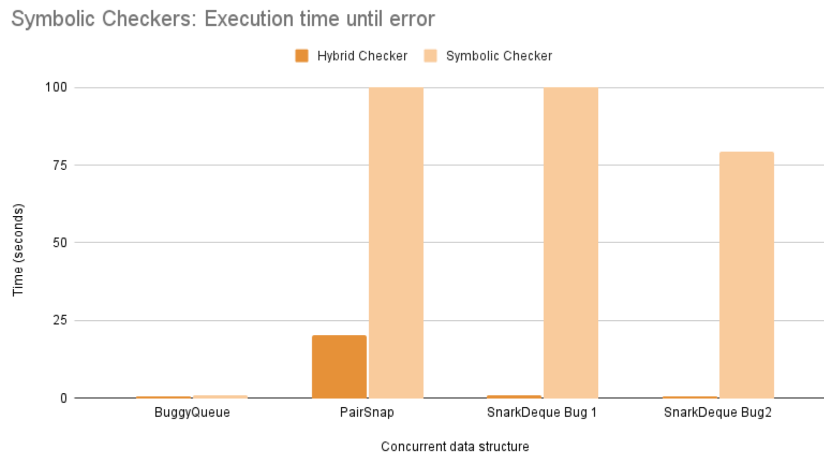
Figure 4.9:



External Concrete Checker Types: *Unsound Automatic does not find error for LockFreeList Bug1 or Bug2. All other errors found.* Execution time results (from Table 4.13) for the time taken until the respective linearisation error was found for each of the test suite SUTs.

Figure 4.10:

Stellenbosch University https://scholar.sun.ac.za



Symbolic Checker Types: *For the Symbolic Checker the SnarkDeque Bug1 took 3,328 seconds and PairSnap took 11,644 seconds, overflow not shown in figure.* Execution time results (from Table 4.13) for the time taken until the respective linearisation error was found for each of the test suite SUTs.

Figure 4.11:

efficient than their equivalent internal counterparts, see Section. Figures 4.9 and 4.10 show that the external checkers took seconds and the internal checkers took tens of seconds in the unsound case, and that the external checkers took tens of seconds and the internal checkers took hundreds of seconds in the sound case.

Interestingly we have found that **the symbolic checker's are able to, for the right depth-limit, find the linearisation errors in a reasonable amount of time**. It is difficult to determine the 'sweet spot' for the depth limit since too shallow will not ever reach the error and too deep will result in too large a search space to even get to the error. An iterative-deepening approach to error finding will likely yield the best error-finding results for this checker. Figure 4.11 shows that four out six errors were able to be found within our 12h timeout period.

The hybrid checker does not use automatic test case generation but generates argument values for a user-specified test case such that all reachable paths of the test case are explored. **The hybrid checker thus finds the linearisation errors much quicker than the symbolic checker, but it requires much more specified input and searches only one of the test cases generated by the symbolic checker.**

Stellenbosch University https://scholar.sun.ac.za

4.4.5 Depth until error

In this section the depth at which each error was found, and the maximum depth reached up until the errors, is shown and the reasons for these results discussed. The depth results for the error-finding experiments is shown in Table 4.14.

The depth of each error is normally very close to the maximum depth reached during execution, three main factors contribute to this:

1. Applicable to all the *concrete linearisation checkers*: The concrete input situations are hand-crafted for the exact linearisation error which is to be located, thus the input includes the minimum number of operations necessary for the error to arise which implies that the error will take place just short of the operation sequences ending. For this reason the results will show the linearisation errors close to the depth limit.

2. Specific only to the *automatic linearisability checkers*: the linearisability checking segment of the computation is performed at JPF's end states and because the search tree is composed of many different bytecode interleavings of the same bytecode instructions it follows that the program end states will be at similar depths.

3. Specific only to the *symbolic checkers*: Since the symbolic checker suffers scalability problems, a depth limit is imposed linearisability checking handle the search space size. An iterative deepening approach to linearisation error finding was used because preliminary results indicated that this was the most efficient way to locate errors for this checker; thus the linearisation errors are often close to the imposed depth limit.

Stellenbosch University https://scholar.sun.ac.za

SUT Algorithm	Data Format:	Concrete Checkers								Symbolic Checkers	
		Unsound				Sound				External (Unsound)	
		Internal		External		Internal		External			
		Aut.	Lin. Point	Aut.	Lin. Point	Aut.	Lin. Point	Aut.	Lin. Point	Hybrid	Symbolic
BuggyQueue (unique-key)	Depth of error:	166	426	63	72	221	471	11	110	33	22
	Max Depth:	181	1,064	63	77	254	1,129	116	116	33	22
BuggyQueue (duplicate-key)	Depth of error:	✗	✗	✗	✗	✗	✗	✗	✗	33	22
	Max Depth:									33	22
LockFreeList Bug1	Depth of error:	274	254	✗	114	394	368	261	158	timeout	timeout
	Max Depth:	297	363		119	418	499	271	271		
LockFreeList Bug2	Depth of error:	228	207	✗	97	327	300	221	170	timeout	timeout
	Max Depth:	234	301		102	352	416	231	231		
PairSnap	Depth of error:	169	179	23	19	247	257	55	51	22	25
	Max Depth:	187	187	25	27	282	282	67	67	22	25
SnarkDeque Bug1	Depth of error:	184	1,116	35	33	260	memory limit reached	83	77	21	Bug1: 25 32
	Max Depth:	213	1,592	45	41	308		107	107	21	Bug2: 32 32
SnarkDeque Bug2	Depth of error:	146	571	47	31	201	616	95	84	29	Bug1: 25 32
	Max Depth:	161	1,061	48	40	233	1,181	99	99	29	Bug2: 32 32

Table 4.14: **Depth at which each error was found and max depth till the error.** The results show the depth at which the first linearisation error was found, for each algorithm-checker combination, and the maximum depth reached until that error was found; for execution on the test suite SUTs until the error.

Stellenbosch University https://scholar.sun.ac.za

4.5 ERROR FINDING CONCLUSIONS

In this section we investigated the error-finding ability, and efficiency in finding those errors, of all the checkers.

The analysis for the concrete checkers showed that the sound concrete checkers were able to find all the linearisation errors but at a high execution time cost for the guarantee of soundness. The unsound concrete checkers were shown to execute in a fraction of the time taken by the corresponding sound checkers, and were able to find the linearisation errors most of the time; although they cannot be used guarantee soundness with respect to linearisability. The external checkers were also found to be consistently more efficient at error finding than the equivalent internal checkers. Results that correlate to their relative efficiency over the entire search space.

The symbolic checker was able to locate most of the test suite errors for general input, despite its scalability problems and depth limit requirement. The symbolic checker was shown to have two benefits, other than general input, in linearisability error-finding: 1. the symbolic checker is able to find errors which the concrete checkers miss for input situations that cause the data structure to contain duplicate values at one time, and 2. the symbolic checker is able to find multiple errors in one run because it checks linearisability for all possible test cases and execution paths. The symbolic checkers were able to, for the right depth-limit, find the linearisation errors in a reasonable amount of time despite the checker's scalability problems. An iterative deepening approach to error finding was found the most useful in finding the test suite errors.

Stellenbosch University https://scholar.sun.ac.za

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 CONCLUSION

In this thesis we investigated various strategies for checking linearisability of non-blocking concurrent data structures using model checking as a basis.

Linearisability checking was integrated into the Java PathFinder (JPF) model checker by implementing the existing strategies reported in the literature. The design and implementation details of the linearisation point and automatic checking strategies have been discussed as well as the details of our proposed improvements and extensions. Soundness was discussed in detail and the linearisation checkers were implemented such that they can run in a mode that guarantees soundness or a mode that does not guarantee soundness.

We further proposed a symbolic strategy for checking linearisability using the Symbolic PathFinder model checker. Two versions were presented: 1. a Symbolic Linearisability Checker that makes use of symbolic execution and automatic test case generation to check all program paths for all operation sequence possibilities, for a given number of operations per thread, and 2. a Hybrid (concrete-symbolic) Linearisability Checker that uses symbolic execution, but not automatic test case generation, to check all program paths for a user-specified operation sequence.

Finally, extensive experiments of all our implementations of existing techniques and the new approaches proposed in this thesis were performed, in the same model checking framework and were run on the same hardware, to compare their relative strengths and weaknesses in user-requirements, efficiency, scalability, and error finding ability. Our findings, for all experiments on the same system hardware and model checking framework, were as follows.

5.1.1 Efficiency

The analysis showed that **the main factor contributing to the efficiency of the linearisation checkers is the amount and type of code instrumentation added to the SUT**. Model checking suffers from the state explosion problem and code instrumentation adds information to the SUT, which results in an increase of the model checker's search space and thus exacerbates the

Stellenbosch University https://scholar.sun.ac.za

scaling problem. The internal checkers add the largest amount of code instrumentation to the SUT, including all linearisability checking logic into the model checker's search space. Thus, as expected, the internal checkers were shown to perform longer executions than the external checkers. The external checkers do not require linearisability checking logic in the SUT, which excludes the extra linearisability checking state information from the model checker's search space and thus eliminate the main performance-limiting factor of the internal checkers.

As expected, the linearisability checkers execute significantly longer searches when running in sound mode than when running in unsound mode. The soundness instrumentation, not only adds information to the model checker's search space but also prevents JPF's state-space-handling optimisation techniques from performing certain branch cut-offs during execution. The guarantee of soundness therefore, comes at an expensive execution time cost.

The external checkers do not require code instrumentation and add an insignificant amount of resource usage to the verification process. It was clear from our results that the external checker executions were equivalent in time to a purely model checker's execution where linearisability checking was turned off. We thus conclude that the linearisability checking logic contributes an insignificant amount of time and space resource usage compared to that of the model checker. The Symbolic Checkers are external and thus, similarly to the external concrete checkers, their only performance limiting factor is the symbolic model checker's execution.

The symbolic linearisability checker is more automatic, but more resource intensive than the hybrid linearisability checker. The Symbolic Checker uses both automatic test case generation and symbolic execution to generate all possible test cases, that is sequences of operations for a give number of operations per thread, and generate argument values such that all possible program paths of the test cases are explored. The Hybrid Checker uses only symbolic execution and generates the argument values for all the reachable program paths but for one particularly user-defined test case, that is sequence of operations for each executing thread. Thus, the search space of the hybrid checker is a subset of that of the symbolic checker's and, as expected, the hybrid checker executes much shorter than the symbolic checker on input for the same number of operations per thread.

Analysis of the lazy read optimisation, proposed by Long et al [24], and the hash optimisation,

Stellenbosch University https://scholar.sun.ac.za

proposed in Chapter 3, showed **no noticeable benefit of the automatic checking strategy optimisations** to the overall execution times of the automatic checkers. The experiment results, with and without optimisations, have similar execution times except the hash optimisation which shows a slight benefit for all external sound experiments and internal sound experiments that finished. These optimisations focus on optimising the linearisability checking component of the tool's execution, not the performance limiting factor which is the model checking component; thus it is not surprising that there is no noticeable benefit for the optimised experiments.

5.1.2 Scalability

It was shown that **the scalability of the linearisability checkers is dominated by the scalability of the model checkers that they use**: Java PathFinder and Symbolic PathFinder. The external unsound concrete checkers are the most scalable. The unsound linearisability checkers explore only a subset of the search space explored by their sound counterparts. The internal checkers include extra linearisability state information which exacerbates the state explosion problem. The **symbolic checker** has severe scaling problems which can be managed by imposing a depth limit for the search. Although the depth limit prevents full state space traversal, this checker's ability to perform a very general search means that a large variety of histories are generated by the model checker up until the depth limit. This checker was found to be most effective for linearisability error finding when an iterative deepening approach is applied.

5.1.3 Input

The checkers require input of a SUT which is the concurrent data structure implemented in Java and a correct sequential implementation of the SUT data structure, the sequential oracle.

The concrete checkers are user-intensive, they require a user-specified test case which contains a sequence of operations for each executing thread, and the user-specified arguments for all operations in the test case. The concrete checker's usefulness in finding linearisability errors is constrained by the user's ability to hand-craft test cases in which errors are present.

The symbolic checker performs linearisability checking on all possible test cases and verifies the linearisability of a data structure in general, constrained only by the number of operations to be executed by each thread. The symbolic checker requires only one user-specified integer per executing

Stellenbosch University https://scholar.sun.ac.za

thread; the integer defines the number of generic operations to be executed by the thread. The symbolic checker uses the integer input value(s) to generate all possible test cases, and generate argument values for the test case operations such that all reachable program paths are explored; for the input number bound.

The hybrid checker requires a user-specified test case, which contains a sequence of operations for each executing thread, but does not require the arguments for all operations in the test case; instead it uses symbolic execution to generate argument values such that all reachable program paths are explored.

5.1.4 Concrete Checkers

We found that **although the unsound checkers do not guarantee that each error present will be found, in general they find most of the linearisation errors.** Thus these two tools can be used in complementary ways: 1. If an error cannot be found within a reasonable amount of time using the sound checker, the unsound checker can be used to find most errors in a shorter execution time. 2. If the unsound checker completes its execution and does not find any errors then the sound checker can be used to either find an error that the unsound checker misses or guarantee the linearisability of the data structure.

We found that the **unsound checkers which include code instrumentation in the SUT for purposes other than soundness were able to find errors that the un-instrumented unsound checkers missed**. Although this instrumentation is not targeted at soundness and thus these checkers cannot be used to guarantee soundness, the instrumentation supplies the model checker with enough information to prevent at least some cut-offs, made by JPF's state-space handling optimisations, which could contain the otherwise cut-off error-containing histories and in our case do; thus the errors are found by these unsound versions.

5.1.5 Symbolic Checkers

One symbolic execution is equivalent to multiple concrete executions. The symbolic linearisability checker found, for example, both of the SnarkDeque, one of the test suite algorithms, linearisation errors in one run with only the input of the number of operations to execute per thread. To find these same two errors with the concrete checkers, input of the exact test case for

Stellenbosch University https://scholar.sun.ac.za

each respective linearisation error and a separate run for each test case is required. This illustrates the benefit of the symbolic checker's automatic test case generation, which generates all possible operation sequences, and symbolic execution, which generates argument values for exploration of all program paths.

The symbolic checker can find errors in cases that the concrete checker cannot. For example, the Symbolic Linearisability Checker found a linearisation error in the BuggyQueue algorithm, for a case that was missed by the Concrete Linearisability Checker. For a case in which an operation returns an incorrect response variable but the value of the variable happens to be the same as the correct response value then the concrete checker will not identify the error. The symbolic checker executes using symbols instead of concrete variable values, thus each variable is completely unique to the symbolic checker even if its value is not unique. The symbolic checker would thus pick up on the linearisation error in the case of an incorrect variable with the correct value operation response.

For example, the BuggyQueue algorithm has a linearisation error where it incorrectly returns the variable not at the front of the Queue. If the incorrectly returned variable happens to be the same value as the variable at the front of the queue then the concrete checker will evaluate the response value as correct when it is not. The symbolic checker, however, will correctly identify the variable returned as the variable not at the front of the queue, irrespective of response value, and locate the error.

5.2 FINAL COMMENTS

In this thesis we have provided a thorough evaluation and comparison of the different linearisation checkers and we have shown the results for our symbolic linearisability checker, the major novel contribution in this thesis. We have shown that the symbolic checking tool, although has scaling problems, is effective at linearisation error finding in a reasonable amount of time for most situations, provides a very general, high quality and robust linearisability check by way of its general search and ability to handle duplicate value situations where the concrete checker is not, and it eliminates the the user-intensive task of hand-crafting input situations for a test suite by way of its automatic test case generation and symbolic execution that exhaustively explores all possible test cases and

Stellenbosch University https://scholar.sun.ac.za

reachable program paths. An iterative deepening approach was found to be the most useful method of error finding given our machine constraints and for more powerful hardware this checker could be an effective tool for linearisability checking of non-blocking concurrent data structures.

5.3 FUTURE WORK

In future work we would like to extend the Symbolic Linearisability Checker to automatically extract the sequential specification for a data structure and extend functionality so that it will work for more parameter and return types than are currently supported. We would also like to extend the linearisability checking tools to be able to check multiple non-blocking concurrent data structures in a program, in a single run.

It was found that linearisation point checking is not easily compatible with SPF's framework, but we would like to develop a Symbolic Linearisation Point Checker for its benefit in on-the-fly linearisability checks instead of end-state checking as for the automatic strategy; the on-the-fly checks will be especially advantageous in the badly scaling symbolic setting where end states are often cut off by the depth limit.

Pasareanu and Rungta have noted the scalability problems of SPF and considered parallelising SPF for future research [27]. Considering the linearisability checking benefits of a more scalable symbolic model checker, and our preliminary results which show most errors found on general input with the scaling problems, we would like to implement parallelising mechanisms for SPF's model checking search and run the linearisability checking tool using better hardware.

Stellenbosch University https://scholar.sun.ac.za

Bibliography

[1] Juan Alemany and Edward W Felten. "Performance issues in non-blocking synchronization on shared-memory multiprocessors". In: *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*. 1992, pp. 125–134.

[2] Saswat Anand, Corina S Păsăreanu, and Willem Visser. "JPF–SE: A symbolic execution extension to java pathfinder". In: *International conference on tools and algorithms for the construction and analysis of systems*. Springer. 2007, pp. 134–138.

[3] Greg Barnes. "A method for implementing lock-free shared-data structures". In: *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*. 1993, pp. 261–270.

[4] Sebastian Burckhardt et al. "Line-up: a complete and automatic linearizability checker". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2010, pp. 330–340.

[5] *CFSerializer (JPF API)*. `https://wuyongzheng.github.io/jpfdoc/gov/nasa/jpf/vm/serialize/CFSerializer.html`. Accessed: 2021-07-01.

[6] Edmund M Clarke Jr et al. *Model checking*. MIT press, 2018.

[7] David L Detlefs et al. "Even better DCAS-based concurrent deques". In: *International Symposium on Distributed Computing*. Springer. 2000, pp. 59–73.

[8] Brijesh Dongol and John Derrick. "Verifying linearisability: A comparative survey". In: *ACM Computing Surveys (CSUR)* 48.2 (2015), pp. 1–43.

[9] Patrick Doolan et al. "Improving the scalability of automatic linearizability checking in SPIN". In: *International Conference on Formal Engineering Methods*. Springer. 2017, pp. 105–121.

[10] Tayfun Elmas, Serdar Tasiran, and Shaz Qadeer. "Vyrd: verifying concurrent programs by runtime refinement-violation detection". In: *ACM SIGPLAN Notices* 40.6 (2005), pp. 27–37.

[11] Cormac Flanagan. "Verifying commit-atomicity using model-checking". In: *International SPIN Workshop on Model Checking of Software*. Springer. 2004, pp. 252–266.

Stellenbosch University https://scholar.sun.ac.za

[12] David Friggens. “On the Use of Model Checking for the Bounded and Unbounded Verification of Nonblocking Concurrent Data Structures”. In: (2013).

[13] Maurice Herlihy. *A methodology for implementing highly concurrent data objects*. Cambridge Research Laboratory, Digital Equipment Corporation, 1991.

[14] Maurice P Herlihy and Jeannette M Wing. “Axioms for concurrent objects”. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1987, pp. 13–26.

[15] Maurice P Herlihy and Jeannette M Wing. “Linearizability: A correctness condition for concurrent objects”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[16] *Java PathFinder*. `https://github.com/javapathfinder/jpf-core/wiki/Testing-vs.-Model-Checking`. Accessed: 2021-07-01.

[17] *JenkinsStateSet (JPF API)*. `https://wuyongzheng.github.io/jpfdoc/gov/nasa/jpf/vm/JenkinsStateSet.html`. Accessed: 2021-07-01.

[18] *JPF-Core Wiki: Partial Order Reduction*. `https://github.com/javapathfinder/jpf-core/wiki/Partial-Order-Reduction`.

[19] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. “Generalized symbolic execution for model checking and testing”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2003, pp. 553–568.

[20] James C King. “Symbolic execution and program testing”. In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[21] Anthony LaMarca. “A performance evaluation of lock-free synchronization protocols”. In: *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*. 1994, pp. 130–140.

[22] Hongjin Liang and Xinyu Feng. “Modular verification of linearizability with non-fixed linearization points”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2013, pp. 459–470.

Stellenbosch University https://scholar.sun.ac.za

[23] Yang Liu et al. "Model checking linearizability via refinement". In: *International Symposium on Formal Methods*. Springer. 2009, pp. 321–337.

[24] Zhenyue Long and Yu Zhang. "Checking linearizability with fine-grained traces". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. 2016, pp. 1394–1400.

[25] Bertrand Meyer. *Soundness and Completeness: With Precision*. Apr. 2019. URL: `https://cacm.acm.org/blogs/blog-cacm/236068-soundness-and-completeness-with-precision/fulltext`.

[26] Madanlal Musuvathi et al. "Finding and Reproducing Heisenbugs in Concurrent Programs." In: *OSDI*. Vol. 8. 2008. 2008.

[27] Corina S Păsăreanu and Neha Rungta. "Symbolic PathFinder: symbolic execution of Java bytecode". In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 2010, pp. 179–180.

[28] Corina S Păsăreanu, Neha Rungta, and Willem Visser. "Symbolic execution with mixed concrete-symbolic solving". In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 2011, pp. 34–44.

[29] Corina S Păsăreanu and Willem Visser. "Verification of Java programs using symbolic execution and invariant generation". In: *International SPIN Workshop on Model Checking of Software*. Springer. 2004, pp. 164–181.

[30] Corina S Păsăreanu et al. "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis". In: *Automated Software Engineering* 20.3 (2013), pp. 391–425.

[31] Mauro Pezze and Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[32] Shaz Qadeer, Ali Sezgin, and Serdar Tasiran. "Back and forth: Prophecy variables for static verification of concurrent programs". In: *Tech. Rep. MSR-TR-2009-142* (2009).

[33] Chien-Hua Shann, Ting-Lu Huang, and Cheng Chen. "A practical nonblocking queue algorithm using compare-and-swap". In: *Proceedings Seventh International Conference on Parallel and Distributed Systems (Cat. No. PR00568)*. IEEE. 2000, pp. 470–475.

Stellenbosch University https://scholar.sun.ac.za

[34] Maurice Herlihy. Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers, 2012.

[35] Jun Sun et al. "PAT: Towards flexible verification under fairness". In: *International conference on computer aided verification*. Springer. 2009, pp. 709–714.

[36] Oleg Travkin, Annika Mütze, and Heike Wehrheim. "SPIN as a linearizability checker under weak memory models". In: *Haifa Verification Conference*. Springer. 2013, pp. 311–326.

[37] Viktor Vafeiadis. *Modular fine-grained concurrency verification*. Tech. rep. University of Cambridge, Computer Laboratory, 2008.

[38] Martin Vechev and Eran Yahav. "Deriving linearizable fine-grained concurrent objects". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2008, pp. 125–135.

[39] Martin Vechev, Eran Yahav, and Greta Yorsh. "Experience with model checking linearizability". In: *International SPIN Workshop on Model Checking of Software*. Springer. 2009, pp. 261–278.

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

APPENDIX A

CONCURRENT HISTORY EXAMPLES

A.1 CONCURRENT HISTORY GENERATION POSSIBILITIES FOR THE LOCKFREELIST ALGORITHM ON A PARTICULAR INPUT EXAMPLE

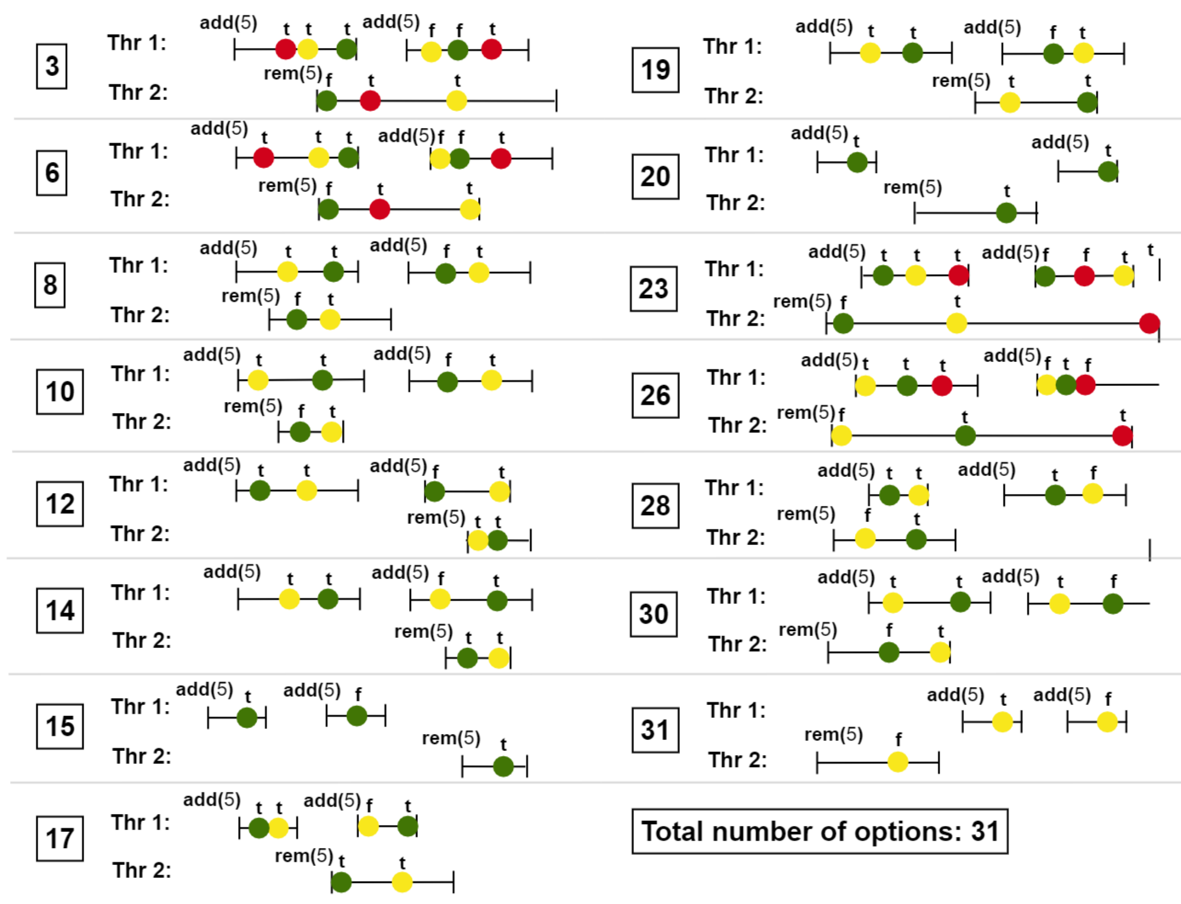


Figure A.1: **Diagram depicting the 31 possible concurrent histories for the LockFreeSet SUT** input specification of thread one execution two *add(5)* operations and thread two executing a *remove(5)* operation. The different possible operation event orderings are shown in different colours for each concurrent history and the corresponding response values of each operation, for a particular colour-coded ordering, shown as a ‘t’ or ‘f’ value above the coloured circle which indicates the point at which the event occurred; a ‘t’ refers to a true response value and a ‘f’ refers to a false response value. The cumulative number of concurrent history operations are shown in a rectangular box to the left of the concurrent history.

Stellenbosch University https://scholar.sun.ac.za

A.2 LINEARISATION-ERROR-CONTAINING CONCURRENT HISTORY DIAGRAMS FOR THE BUGGYQUEUE, SNARKDEQUE, LOCKFREELIST, AND PAIRSNAP ALGORITHMS

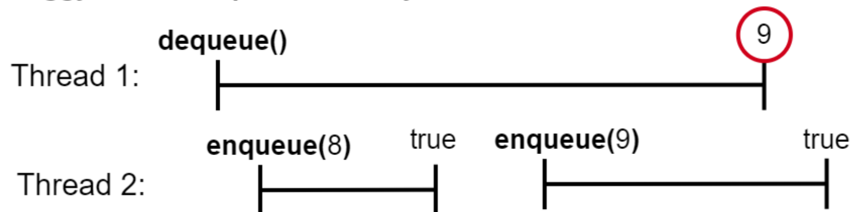


Figure A.2: **A linearisation error containing concurrent history for the BuggyQueue algorithm** where the dequeue operation of one thread is interrupted by two enqueue operations of a different thread; resulting in the incorrect return of a value not from the front of the queue.

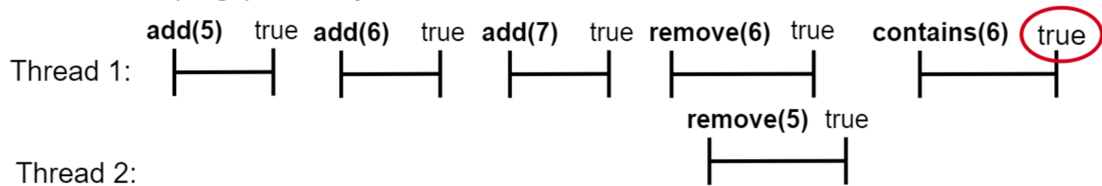


Figure A.3: **A linearisation error containing concurrent history for the LockFreeList algorithm (bug1)** where overlapping *remove(5)* and *remove(6)* operations result in the 5 value correctly removed but the 6 value still present in the data structure; as illustrated by the successful *contains(6)* operation which executes after the response events of the overlapping operations.

Stellenbosch University https://scholar.sun.ac.za

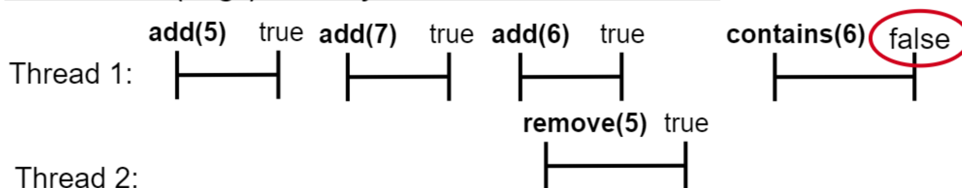


Figure A.4: **A linearisation error containing concurrent history for the LockFreeList algorithm (bug2)** where overlapping successful *remove(5)* and successful *add(6)* operations result in the value of 5 correctly removed but the value of 6 not added to the list data structure; illustrated by the unsuccessful *contains(6)* operation which executes after the response events of the two overlapping operations.

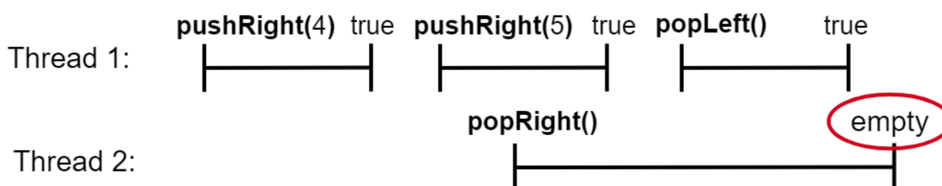


Figure A.5: **A linearisation error containing concurrent history for the SnarkDeque algorithm (bug1)** that causes a pop operation to return empty when the queue is not empty.

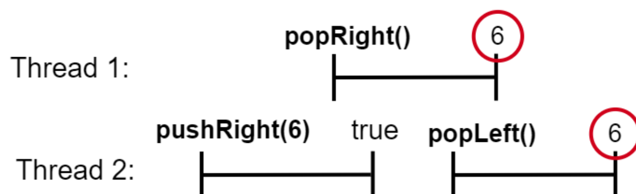


Figure A.6: **A linearisation error containing concurrent history for the SnarkDeque algorithm (Bug2)** that contains an ABA=type error resulting in two pop operations returning the same value.

Stellenbosch University https://scholar.sun.ac.za

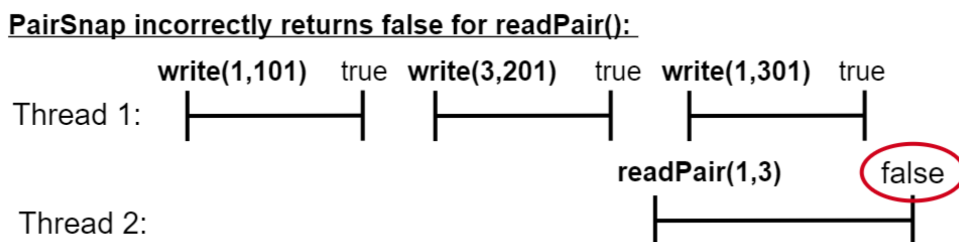


Figure A.7: **A linearisation error containing concurrent history for the PairSnap algorithm** that causes the readPair operation to incorrectly return false when the two memory locations read contain non-null contents.

A.3 NON-FIXED LINEARISATION POINT EXAMPLES FOR AN UNSUCCESSFUL *ADD*, AN UNSUCCESSFUL *CONTAINS*, AND A SUCCESSFUL *CONTAINS* OPERATION OF THE LOCKFREESET DATA STRUCTURE.

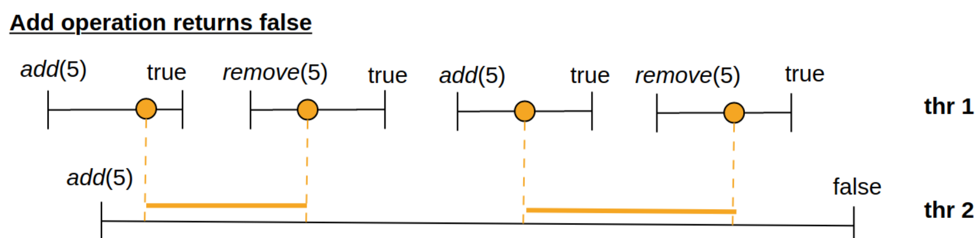


Figure A.8: **An example of a concurrent history where thread 2 contains an add operation with non-fixed linearization points.** The example shows, as illustrated by the line segment, the segments of thread 2's operation which represent the add(5) operation returning false; the areas in between these demarcated segments would result in the add(5) operation returning true.

Stellenbosch University https://scholar.sun.ac.za

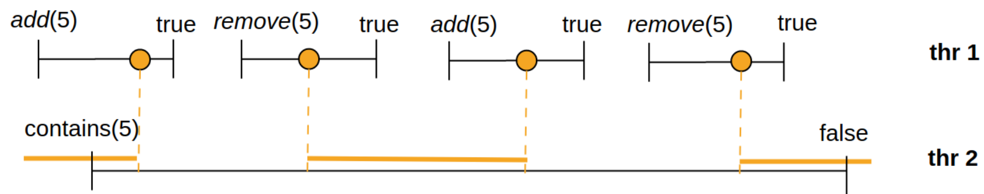


Figure A.9: **An example of a concurrent history where thread 2 contains a contains operation with non-fixed linearization points.** The example shows, as illustrated by the line segment, the segments of thread 2's operation which represent the contains(5) operation returning false; the areas in between these demarcated segments would result in the contains(5) operation returning true.

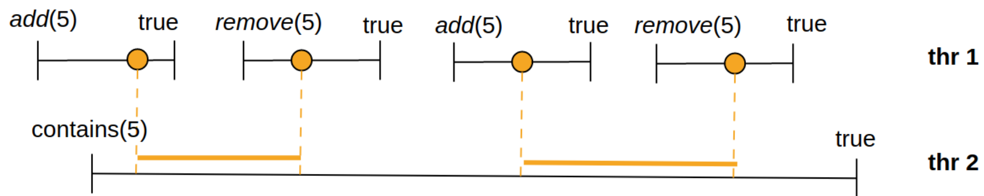


Figure A.10: **An example of a concurrent history where thread 2 contains a true returning contains operation operation with non-fixed linearization points.** The example shows, as illustrated by the line segment, the segments of thread 2's operation which represent the contains(5) operation returning true; the areas in between these demarcated segments would result in the contains(5) operation returning false.

Stellenbosch University https://scholar.sun.ac.za

APPENDIX B

JAVA CODE

B.1 BUGGYQUEUE JAVA CLASS (SUT)

```
public class BuggyQueue {
 int NullForInt = Integer.MIN_VALUE;
 AtomicInteger FRONT = new AtomicInteger(0);
 AtomicInteger REAR = new AtomicInteger(0);
 AtomicReferenceArray Q_atomic;
 int L = 4;
 Node[] Q = new Node[L];

 public BuggyQueue() {
  for (int i = 0; i < L; i++) {
   Q[i] = new Node(NullForInt, 0, true);
  }
  Q_atomic = new AtomicReferenceArray(Q);
 }

 /** return true if success, else false */
 public boolean enqueue(int itemEnq) {
  int rear;
  Node x;
  boolean resultFound = false;
  do {
   do {
    rear = REAR.get();
    x = (Node) Q_atomic.get(rear%L);
   } while ((rear != REAR.get()
   || rear == FRONT.get()+L));
   if (x.getIntIsNull()) {
    if (Q_atomic.compareAndSet(rear%L, x,
      new Node(itemEnq, x.counter+1,
      false)))
    {
     REAR.compareAndSet(rear, rear+1);
     resultFound = true;
    }
   } else {
    REAR.compareAndSet(rear, rear+1);
   }
  } while (!resultFound);
   return true;
 }

 /** return value if success else -1 */
 public int dequeue() {
  int front;
  Node x;
  boolean resultFound = false;
  int result = -1;
  do {
   do {
    front = FRONT.get();
    x = (Node) Q_atomic.get(front%L);
   } while ((front != FRONT.get()
   || front == REAR.get()));
   if (!x.getIntIsNull()) {
    if (Q_atomic.compareAndSet(front%L, x,
      new Node(NullForInt, x.counter+1,
      true)))
    {
```

Stellenbosch University https://scholar.sun.ac.za

```
      FRONT.compareAndSet(front, front+1);
      result = x.val;
      resultFound = true;
     }
    } else {
     FRONT.compareAndSet(front, front+1);
    }
   } while (!resultFound);
   return result;
  }

  private static class Node {
   private int val;
   private int counter;
   private boolean intIsNull;
   public Node(int val, int counter,
   boolean intIsNull) {
    this.val = val;
    this.counter = counter;
    this.intIsNull = intIsNull;
   }
   public void setVal(int val) {
    this.val = val;
   }

   public void setCounter(int counter) {
    this.counter = counter;
   }

   public void setIntIsNull(boolean intIsNull){
    this.intIsNull = intIsNull;
   }

   public int getVal() {
    return this.val;
   }

   public int getCounter() {
    return this.counter;
   }

   public boolean getIntIsNull() {
    return intIsNull;
   }
  }
 }
```

B.2 LOCKFREESET JAVA CLASS (SUT)

```
class LockFreeSet {
 private final Object lock = new Object();
 private volatile Entry head;


 /** Return the location of key in the set
 (where it is or should be added) */
 private void locate(EntryLocationHolder loc,
                    int key) {
  loc.pred = null;
  loc.curr = head;

  while ((loc.curr != null)
  && (loc.curr.key < key)) {
```

Stellenbosch University https://scholar.sun.ac.za

```
   loc.pred = loc.curr;
   loc.curr = loc.curr.next;
  }
 }

 /** Returns true if key is in the set and
 false if key is not in the set */
 public boolean contains(int key) {
  EntryLocationHolder loc =
  new EntryLocationHolder();
  locate(loc, key);
  return ((loc.curr != null)
  && (loc.curr.key == key));
 }

 /** Add an entry (with the key field set
 to key) to the set */
 public boolean add(int key) {
  EntryLocationHolder loc =
  new EntryLocationHolder();
  Entry entry;
  boolean success = false;
  boolean alreadyAdded = false;

  do {
   locate(loc, key);
   if ((loc.curr != null)
   && (loc.curr.key == key)) {
    // possibly by another thread since locate
    alreadyAdded = true;
   } else {
    entry = new Entry(key);
    entry.next = loc.curr;
    synchronized (lock) {
     if ((loc.pred != null)
     && (loc.pred.next == loc.curr)
     && !loc.pred.marked) {
      loc.pred.next = entry;
      success = true;
     } else if ((head == loc.curr) &&
     (head == null || (!head.marked))) {
      /** another thread has not added an entry
      at loc.pred.next (or head)
      or marked it for deletion */
      head = entry;
      success = true;
     }
    }
   }
  } while (!success && !alreadyAdded);
  return success;
 }

 /** Remove the entry
 (with the key field equal to key)
 from the set */
 public boolean remove(int key) {
  boolean success = false;
  boolean alreadyRemoved = false;
  EntryLocationHolder loc =
  new EntryLocationHolder();

  do {
   locate(loc, key);
   if ((loc.curr == null)
   || loc.curr.key != key) {
    // possibly by another thread since locate
    alreadyRemoved = true;
```

Stellenbosch University https://scholar.sun.ac.za

```
    } else {
     loc.curr.marked = true;
     synchronized (lock) {
      if ((loc.pred != null)
      && (!loc.pred.marked
      && (loc.pred.next == loc.curr))) {
       loc.pred.next = loc.curr.next;
       success = true;
      } else if (head == loc.curr) {
       head = loc.curr.next;
       success = true;
      }
      /** if no other thread marked pred
      for deletion or added another entry
      before curr since locate */
     }
    }
   } while (!success && !alreadyRemoved);
   return success;
  }

  // An entry in the Set
  static class Entry {
   int key;
   boolean marked;
   Entry next;

   // Constructor
   Entry(int key_val) {
    key = key_val;
    marked = false;
   }
  }

  /** Used by locate to return the
  field(s) required by add, remove, etc. */
  static class EntryLocationHolder {
   volatile Entry pred;
   volatile Entry curr;
  }
 }
```

B.3 PSEUDOCODE FOR JPF'S DEPTH-FIRST-SEARCH MODEL CHECKING TRAVERSAL

```
def JavaPathFinderDFS:
  def DFS:
    while searchNotCompleted()
      if isEndState() or !isNewState()
      then backtrackState()
      and listenerClass.stateBacktracked()
      endif
      if hashUnexploredChildren()
      then advanceToUnexploredChildState()
      and listenerClass.stateAdvanced:()
      endif
    endWhile

  def advanceToUnexploredChildState:
    forEach getInstructionsForNextTransition()
      then executeInstruction()
```

Stellenbosch University https://scholar.sun.ac.za

```
and if methodInvocation()
  then listenerClass.methodEntered()
else if methodResponseEvent()
  then listenerClass.methodExited()
  endif
endForEach
```

B.4 PSEUDOCODE FOR AUTOMATIC LINEARISABILITY CHECKING USING JPF'S LISTENER CLASS

```
def ListenerClass:
/** define a structure to store the
ordered events of operation
invacation and response events
for a concurrent history path */
def concurrentHistoryRecord
/** Instantiate instance of
the sequential oracle */
def seqOracle

def stateAdvanced:
if isEndState()
then doLinearisabilityCheck(
        concurrentHistoryRecord)
endif

def stateBacktracked:
if backtrackedPastOperationEvent()
then restorePreviousSequential-
OracleState()
and restorePreviousConcurrentHistory-
RecordState()
endif

def methodEntered:
if MethodInvocationForSUTOperation()
then storeInformationAboutInvocation-
EventInconcurrentHistoryRecord()
endif

def methodExited:
if MethodResponseForSUTOperation()
then storeInformationAboutResponse-
EventInconcurrentHistoryRecord()
endif

def doLinearisabilityCheck:
generateSequentialWitnesses-
OfConcurrentHistory()
TestEachConcurrentHistory()

def TestEachConcurrentHistory():
forEach concurrent history:
if concurrentHistoryIsLinearisable()
then continue
else stopSearch() and
     throwLinearisationError()
endif
endForEach
```

Stellenbosch University https://scholar.sun.ac.za

APPENDIX C

EXPERIMENT RESULTS

C.1 ERROR FINDING

Data Format: NewStates Max Memory (MB)	**Internal Checker Implementations**				**External Checker Implementations**					
	Automatic	**LinPoint**	**Automatic**	**LinPoint**	**Automatic**	**LinPoint**	**Hybrid**	**Symbolic**	**Automatic**	**LinPoint**
	Unsound		**Sound**		**Unsound**				**Sound**	
	+- 1s	+- 5s	+-5s	+-5s	+- 1s	+-1s	+-1s	+-1s	+-1s	+-1s
BuggyQueue (unique-value)	14,715 426	541,147 679	387,561 677	1,155,102 680	314 238	309 238	1,935 238	2,802 238	6,522 425	6,505 425
BuggyQueue (duplicate-value)	not found	not found	not found	not found	not found	not found			not found	not found
LockFreeList Bug1	23,928 427	1,254 238	1,876,390 678	9,182 678	not found	887 238	timeout	timeout	146,603 425	146,218 1,178
LockFreeList Bug2	22,427 425	1,113 238	1,853,882 678	8,329 678	not found	845 238	timeout	timeout	150,575 678	150,196 675
PairSnap	1,962 238	1,051 238	24,202 425	9,337 425	57 238	57 238	807,308 1,456	622,840,540 1,456	3,498 300	3,497 425
SnarkDeque Bug1	35,539 425	4,432,644 684	2,725,592 550	Memory limit reached	459 238	490 238	2,184,943 675	41,514,040 675	70,449 675	71,301 675
SnarkDeque Bug2	15,316 425	724,078 681	369,503 426	15,370,242 683	302 238	329 238	1,820 238	1,075,311 1,175	6,001 425	5,993 425

Table C.1: **Execution Time and Memory until error is found**