Università
della
Svizzera
italiana

**Software
Institute**

Designing Concurrent Trees

Model Checking Concurrent Tree Data Structures With TLA+

Daniel Tisdall

June 2021

Supervised by
Prof. Carlo A. Furia

Co-Supervised by
Prof. Fernando Pedone

Software & Data Engineering Master Thesis

Abstract

Concurrent tree data structures are difficult to design and implement correctly. This is due to numerous challenges at the design level and also at the implementation level. For example, at the design level, orderings of operations between processes on the graph structure must be correct. At the implementation level, programmers use fine grained shared memory operations to improve performance. As a result, they are forced to navigate complex interactions between hardware and programming language or library memory models.

As a result of the complexity, concurrent data structures are often found to exhibit correctness bugs and other violations of invariant properties, which can be extremely damaging, and expensive and difficult to fix. There is a need to attain strong assurances of correctness for such data structures, and many approaches have been tried.

In this thesis we present a methodology for improving the extent to which (concurrent) trees can be believed to be correct, using the TLA+ language and model checking. Model checking is an automatic verification technique that can efficiently explore large state spaces of possible program executions (including the instruction interleavings that can occur in a concurrent setting) to check whether certain properties are satisfied or violated. TLA+ is a flexible specification language, suited to modelling concurrent processes, which can be interpreted by model checkers, giving a complete history of the state of a model in case of a property violation.

The methodology we present is focused on pointer based concurrent tree data structures and can be used to check or track down complex bugs in existing structures, or to design and assure correctness of new structures. We applied our methodology, developing a new, verified, concurrent tree. We also applied the methodology to an existing state of the art concurrent tree from the literature, using it to find and explain critical correctness bugs.

‘To believe with certainty we must begin
with doubting.’ - Polish proverb

Acknowledgements

I would like to thank Carlo for being a great supervisor all around and for really taking the time to discuss all the details of things with me, whenever I asked. Cheers Carlo!

I would also like to thank Fernando for being a great co-supervisor, trusting me to an awesome project, and for finding me a pretty incredible first job. Thanks Fernando!

Additionally I'd like to thank Enrique for taking the time to discuss the Chunked AVL tree with me even before the thesis started, and for inventing such a cool problem to work on.

I'd like to thank my parents for cooking me so many meals during the pandemic! And Martin for being a great friend and saving my butt countless times during the master. I'd also like to thank the other great teammates I had throughout many group projects, including Segi and Brenda.

Thank you also Markus Kuppe for helping me out with TLA+ issues, as well as the wider TLA+ community for being so helpful. A big shout out to David Barri for implementing a feature at my request in his invaluable tla2json tool.

Finally I'd like to thank everybody at USI Informatics and the Software Institute for putting together a great master. Massive thanks to Michele Lanza for creating a master thesis template which is way more beautiful than any other thesis template! Knowing that the end product would look good made writing so much less painful.

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

Concurrent data structures are notoriously difficult to design and implement, and there is a need for better tooling and methodologies for verifying their correctness and finding bugs both in the design stage and the implementation stage.

In this introduction we introduce the basic concepts of concurrent data structures and give an idea of the kind of difficulties that programmers and computer scientists face when working on them. We outline the objective of this thesis, which is to develop a pragmatic methodology for practitioners, and we give a brief overview of how our approach fits in with the currently available methodologies. We briefly discuss how we demonstrate and evaluate our methodology by applying it to a real existing concurrent tree as well as a novel concurrent tree. The later sections of this introduction briefly explain the the key points of the methodology itself, summarize our key results and mention possible directions for future work.

The rest of the thesis contains detailed reports on each of these areas.

1.1 Why concurrent data structures are difficult to create

The prevalence of multi-core processors in todays computers [32] makes concurrent data structures highly desirable as they typically offer much greater throughput than their sequential counterparts and are a high performance method of communication between processes[1]. Since programmers want to make these data structures[2] as performant as possible they are usually designed and implemented using low-level primitives offered by the operating system, programming language or specialized concurrency library, and use fine grained control over shared memory.

The interleaving of operations on shared memory creates difficulties at all levels of the stack, meaning that behavioral properties offered to the application developer depend on many layers of software being correct, all the way from cache coherence protocols [43] in the processor itself to the high level design of the concurrent data structure. In-between are the operating system, programming language memory models and concurrency toolkit libraries, each of which offer their own guarantees to the layer above and rely on correctness in the layers below.

Concurrent data structures implementers use various features of these layers and therefore need a good understanding of their interactions to ensure a correct implementation, this complexity make implementations notoriously hard to write. Complex data structures such as lists and trees with graph object models must deal with these problems, but also have to deal with design level challenges that arise in the interleaving of operations on the graph. Graph manipulations are usually localised and fine grained as this enables more concurrency and therefore performance, but the granularity increases the number of possible operation interleavings and leads to algorithm being necessarily complex in order to avoid race conditions.

[1] In this thesis we use ‘process’ in the abstract sense, using the term in the place of operating system threads.

[2] We may refer to only ‘structures’ but we always mean data structures. We usually mean concurrent tree data structures, unless the context clearly suggests otherwise.

As a result of the difficulty of creating concurrent data structures, it is not surprising that they often contain bugs. This is undesirable; data structures are a basic component of software at all layers of the stack. There is a great need for them to be correct.

1.2 Objective: a pragmatic and flexible methodology for correctness

As a result of the importance of creating correct structures, a lot of work is done on improving methods and tooling for assuring their correctness. Typically methods fall into the categories of testing, formal verification, and pen-and-paper proof. In practice, creators use the methods in combination. Testing is a wide field and has many sub methodologies of varying effectiveness, including stress testing or property testing, and exhaustive testing. Formal verification is another approach which aims to encode data structures in non-ambiguous formal logic, which can be proved by programs. Finally, pen-and-paper proof is often given to support the correctness of concurrent data structures. More details of these methods and how they are applied to concurrent data structures is given in Chapter 2.

Each method described above has advantages and drawbacks. Pen-and-paper proofs are often lengthy, difficult to communicate effectively and error prone. Formal verification, while having the potential to offer absolute correctness, is generally considered to be very difficult and to require an expert. Testing offers ease of use but many testing methods do not execute program paths which are the result of process interleavings, meaning that bugs in concurrent data structures are easily missed.

In this thesis we use model checking, which is a rigorous form of exhaustive testing that can give a much higher degree of confidence of correctness compared to traditional testing. Model checking works by efficiently exploring large state spaces of possible program executions (including the instruction interleaving that can occur in a concurrent setting) to check whether specified properties are satisfied or violated by any possible execution in the state space.

Our objective is to develop provide a pragmatic methodology for checking concurrent trees using model checking. We mean pragmatic in the sense that it should not require an expert user, should provide a high level of confidence in the correctness of a checked algorithm, and should not be prohibitively time consuming to carry out.

1.3 Applying the methodology to real life examples

We apply our methodology to two concrete data structures and their variants. The first structure is the Practical Concurrent Binary Search Tree due to Bronson *et al.* [11]. We call this tree the BAVL tree. The second structure is the Multi-Reader Single-Writer Chunked AVL Tree, which we designed. We call this tree the MRSW CAVL tree.

Both the BAVL and MRSW CAVL trees are concurrent AVL trees. AVL trees implement a concurrent Set or Map abstract data type. They are especially interesting and useful because they are balanced, meaning that the height[3] of the left and right children of every node in the tree does not differ in absolute value by more than 1. This property improves the worst-case bound on the latency of operations on the tree, compared to non-balanced trees, but makes the algorithms far more complicated in a concurrent setting (when compared to both their sequential counterparts *and* concurrent non-balanced trees).

Balanced concurrent trees are generalizations of many data structures such as concurrent and non-concurrent and balanced and non-balanced trees and lists. Therefore we think that others should be able to take inspiration from our work and borrow our techniques to help with checking a large variety of data structures.

The BAVL and (MRSW) CAVL trees are explained in detail in Chapter 3. The chapter also provides provides background on why we chose the trees for our work.

[3]The path length from a node to its farthest child $+1$, or 1 if the node has no children.

1.4 The methodology

Our methodology uses TLA+, and another language that is part of the TLA+ ecosystem, Pluscal, to develop models of data structures that are computationally tractable, and are able to find many classes of bugs. The methodology is intended to be concrete and practical.

1.4.1 TLA+

TLA+ is a specification language that can be used to models algorithms and systems. TLA+ stands for 'Temporal Logic of Actions (+)'. It is a flexible language that uses mathematical notation to clearly and non-ambiguously define series of actions within a system. Users can define properties that say that system states meet certain criteria, or that series of states (defined by actions) meet certain criteria. TLA+ can be interpreted by programs, usually model checkers. We use the TLC model checker.

We discuss how to model concurrent data structures in TLA+ and then check them in TLC (using other TLA+ compatible model checkers would requires little work). Using TLC we check that properties that should hold in the data structure do so. If TLC finds a violation of a property, it provides a trace[4] that we can work back from to identify the cause of errors.

1.4.2 Understanding and borrowing from the methodology

Our exposition of the methodology treats conceptual and practical challenges separately, so that readers can pick out the parts that are most interesting to them. Many of the techniques for overcoming the conceptual challenges of modelling should be transferrable to other model checking systems.

1.5 Summary of results

We present a methodology for modelling and model checking concurrent tree data structures with TLA+. We demonstrate the value and practicality of our methodology by using it both to design and verify a novel data structure, a Multi-Reader Single-Writer Chunked AVL Tree, and to discover critical bugs in the existing Practical Concurrent Binary Search Tree.

1.5.1 The Multi-Reader Single-Writer Chunked AVL Tree

We use the methodology to develop a concurrent version of an existing data structure, the Chunked AVL Tree. We provide a version of the tree which allows an arbitrary number of reading processes to operate concurrently with a writing process.

1.5.2 Finding bugs in the Practical Concurrent Binary Search Tree

We apply the methodology to the verification of an existing concurrent tree. We find that the tree violates a balance property. We also obtain error traces for two errors. The error traces contain the entire state of the system over the course of an erroneous execution.

1.5.3 Broader contributions of the thesis

Our methodology is practical and effective. Structures designed using our methodology can be trusted to a higher degree than testing alone would allow, and TLA+ models of data structures produced using our

[4]We use 'trace' to mean a chronological history of the entire state of the system.

methodology are unambiguous and can be clearly communicated. Additionally, by applying our methodology, it is possible to gain a deep understanding of complex bugs in concurrent structures very quickly, should bugs exist.

1.6 Outline of the thesis

1.6.1 Chapter 2

Explains the difficulty of designing concurrent trees, with a short review of state of the art concurrency techniques and concepts. Discusses various existing approaches to verifying correctness of concurrent data structures and concurrent trees, with emphasis given to work closely related to ours.

1.6.2 Chapter 3

Provides detailed background on the specific concurrent trees that we use to evaluate our methodology, including principles of the algorithms themselves and motivations for choosing them for our study.

1.6.3 Chapter 4

Describes the methodology in detail in a step by step fashion. Tackles conceptual and practical challenges in applying the steps, including motivation and the reasoning behind each step.

1.6.4 Chapter 5

Presents the major results of applying our methodology. Includes a description of the MRSW CAVL tree, as well as descriptions of the bugs found in the BAVL tree. Summarizes scalability results. Ends with a discussion of limitations and lessons learned.

1.6.5 Chapter 6

Reiterates on the key results and discusses directions for future work in model checking concurrent data structures.

Chapter 2

State of the art

This section outlines the state of recent research in rigorously developing and assuring correctness of concurrent tree data structures. It also motivates the need to assure correctness by giving examples of the kinds of bugs that can occur in such structures, and other problems that occur during the design process.

In the first part of this section we introduce concurrent data structures, specifically concurrent Set and Map abstract data types. We also introduce trees and balanced trees and their concurrent counterparts. Specific kinds of trees are discussed in depth in Chapter 3. The second part of this section illustrates some of the problems that occur in concurrent tree data structures.

In the third part of this section we introduce various approaches for assuring correctness, including pen-and-paper proof, testing, model checking, and formal verification. Since this thesis‘s contributions focus on model checking, this section includes a discussion of existing work on applying TLA+ model checking to concurrent data structures and other problems that use a shared memory model. TLA+ is discussed in further depth in Chapter 4.

The final part of this section outlines how our work fits into the state of the art.

2.1 (Concurrent) data structures

A data structure [19] is a scheme for organizing data in a computers memory[1] so that it can be efficiently stored and retrieved. The data structures we focus on, concurrent trees, are instances of pointer based data structures which implement the Set or Map abstract data type (ADT).

2.1.1 Abstract data types

Abstract data types [53, 57] are formal models of data types that only specify semantics visible to the user of the type. A Set ADT is a type offering $contains(k)$, $insert(k)$ and $erase(k)$ operations. These operations work on a mathematical set. The $insert(k)$ operation adds k to the set, $erase(k)$ removes k from the set and $contains(k)$ returns a boolean indicating whether or not k is in the set. In a Map, each key k is associated to a value v. The operations are then $get(k)$, $insert(k, v)$, and $erase(k)$, meaning retrieve the value associated to k if it is present, associate a value v with a key k, and remove a key k and its associated value v. All the data structures we discuss implement a Map and can be easily adapted to implement a Set. Note that a Map can be implemented from a Set by storing $\langle k, v \rangle$ pairs and a Set can be implemented from a Map by using only dummy values.

2.1.2 Binary trees

Abstract data types such as Map[2] are implemented in programs by representing the underlying set and associations as pairs or linked pairs, together with extra data needed for bookkeeping. We focus on tree

[1] Although many storage mediums use data structures, we focus on *memory*, also known as *primary storage* [63]

[2] Or Set unless otherwise mentioned.

data structures [19], where keys and values are connected by pointers, and pointers and additional bookkeeping data are themselves linked by pointers. The pointers and the bookkeeping data, together making nodes, are linked in such a way that the graph formed by nodes makes a tree shape. Figure 2.1 illustrates this.

The user of the data structure only needs a handle to the root[3] node to use the tree, and traverses the graph edge by edge from the root to find data. All the tree data structures that we study are binary trees, meaning that the out-degree of each node is in $\{0, 1, 2\}$.

Binary trees have the property that if the out-degrees of a large majority of nodes are in $\{0, 2\}$ then the tree will be approximately balanced. In an approximately balanced tree, traversal from the root to the leaves[4] is efficient. Precisely, an algorithm implementing a balanced binary tree, such as the AVL algorithm [30], provides $\mathcal{O}(\log n)$ running time complexity for *get*, *insert* and *erase*, where n is the number of nodes currently in the tree at time of invocation. Balance has to be restored whenever the tree is modified by *insert* and *erase* operations. This is done by *rotating* regions of the tree. A rotation is a rearrangement of pointers which aims to restore the balance of a subtree. The AVL algorithm is discussed in depth in Chapter 3, as all the trees that we study are based on it.

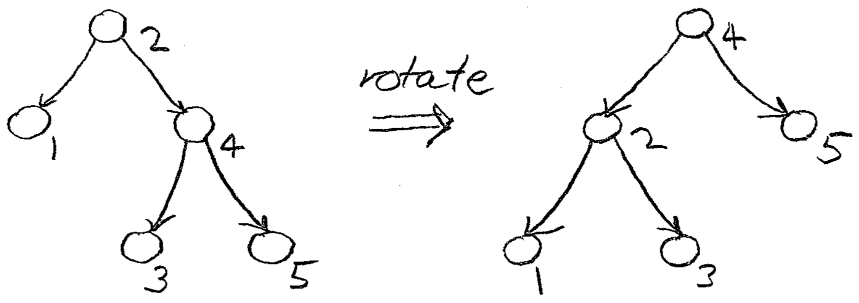

FIGURE 2.1: An illustration of a rotate operation on a binary tree. The left side shows the tree before the operation, and the right side after. The operation is a *rotate left* operation, meaning that the root of the original tree becomes the left child of the root of the tree after the operation.

2.1.3 Adding concurrency

A concurrent data structure augments the properties of its sequential counterpart by allowing multiple processes to operate on it at the same time without introducing race conditions. This means that they can be used by two or more processes for communication, but it is also a potential way for programs and systems to improve performance [36].

Concurrent data structures do not provide exactly the same semantics as sequential data structures. In the presence of concurrent operations on a Set, for example, processes may at any one time disagree on whether or not a certain element is present. To see this, consider two processes p_0 and p_1 operating concurrently on $\emptyset$. Since all operations have a running time, there could be a scenario in which p_0 invokes $get(k)$ and determines the result to be $false$ (k is 'not in the set'). Crucially however, before $get(k)$ returns (responds) to p_0, p_1 invokes $insert(k)$, which succeeds. Subsequently p_1s $insert(k)$ returns, indicating success, meaning that 'now' k is 'in the set'. Consider then that p_0s $get(k)$ completes and returns $false$. The real time ordering of operations returning to p_0 and p_1 does not match a possible any possible sequence of *sequentially* invoked operations. This is illustrated in Figure 2.2. Therefore, in order to precisely define the semantics of concurrent data structures, we need a correctness condition that allows us to consistently reason about concurrent behaviors.

[3]The unique node with an in-degree of 0.
[4]A leaf is a node with height 1

2.1.4 Linearizability

The concept of linearizability [55] is a widely used correctness condition that is helpful for talking precisely about the semantics of concurrent data types, avoiding the impreciseness of discussions similar to the one in Section 2.1.3. We call a concurrent abstract data type *linearizable* if all of the *event sequences* we can obtain by executing concurrent operations on it satisfy certain properties. Next we give a definition of an event sequence and then we give the properties which must be satisfied for it to be linearizable.

An event sequence is a sequence of ordered operation invocations and responses. For example the event sequence for the example in Section 2.1.3 is

$$invokeGet_{p0}(k)invokeInsert_{p1}(k)respondInsert_{p1}(true)respondGet_{p1}(false)$$

Since instruction execution is not atomic in real systems, recording a meaningfully ordered sequence of events can be tricky. In the example we assume that each event has an exact real time associated to it that we can order all events by.

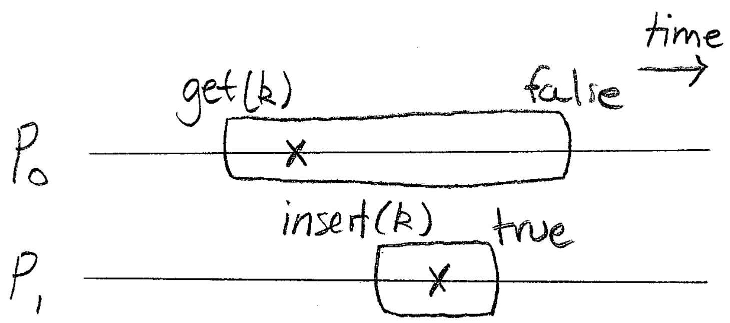

FIGURE 2.2: An event sequence diagram for a concurrent set execution with two processes. Linearization points are marked with an 'x' and the left and right sides of wedges are labelled with events. Process p_0 executes $get(k)$ and p_1 executes $insert(k)$. Although p_0's *get* begins before p_1's *insert* and returns after p_1 is finished, its linearization point is ordered ahead of p_1's.

Such an event sequence is said to be linearizable if it is possible to re-order the events such that

1. Each operations $< invoke, respond >$ pairs are adjacent.
2. An *invoke* event is never re-ordered to be before a *respond* event if it was before the *respond* event in the original sequence.
3. It is possible to obtain the re-ordered sequence from some sequence of operations on the data type.

Put another way, linearizability of an event sequence means that is possible to define 'instants' in-between operation invocations and responses, where the operation *instantaneously takes place*, and for the result to always match what was given by the respond event. A data type is linearizable if we can always define such instants, for any execution. A concurrent data structure is linearizable if it implements a linearizable data type. Such instants are commonly called *linearization points*. An operation is said to *linearize* at its linearization point, if one is defined. We will see that it is not always easy to define linearization points in real data structures, and that it is certainly not always easy to record them even if they are defined. Figure 2.2 shows an event sequence diagram for the example discussed. Possible linearization points are marked on it.

More detailed definitions of linearizability and associated concepts usually give a definition which takes into account event sequences in which not all *respond* events have been recorded [38].

2.2 Concurrency problems in concurrent trees

Concurrent tree creators have to overcome many challenges to ensure that the implemented Map is linearizable and highly performant. Broadly we can break the challenges into two categories, the design or conceptual level, and the implementation level. In reality the difference is blurred as at the conceptual level one will want to use concepts and abstractions from software lower in the stack. Using those abstractions is essentially implementation.

We briefly discuss the environment that concurrent tree programs are often implemented in, as it is important background for understanding how bugs can occur. Following that, we end the section by discussing major design level problems that are relevant for the rest of this work, one by one. For each problem we discuss the interactions with implementation, and how bugs or other errors might arise.

2.2.1 Programming environment

Concurrent data structures can be implemented in many programming languages and on many hardware platforms. The trees that we study have implementations in Java [3], C++ [42] or C [44], as well as other languages. All of the example data structures that we discuss in this work use a programming language that provides an environment that's significantly removed from hardware. The programming language typically builds on operating system features to provide its own features, although that isn't always the case. For example C and C++ programs are commonly written for systems without operating systems [5]. Nonetheless, concurrent trees typically use a common set of features offered by many languages and environments. These include, but are not limited to, pointers [44], a garbage collector [72], locks [36], atomic registers and memory fences [8], volatile fields [56] and many other tools.

Libraries or code generators and other programming language or even hardware features can be used to provide an even bigger toolset. These can include specialized memory reclamation techniques such as hazard pointers and epoch based reclamation [58, 36], multi-word compare-and-swap primitives [35] and software or hardware transactional memory primitives (STM/HTM) [66, 25].

Overarching specific algorithms and tools are motifs, which are repeated throughout concurrent algorithm literature. Examples of common motifs include optimistic concurrency [1], mutual-exclusion [36], read-copy-update [34], reference counting [24] and so on, ad infinitum.

Here we give a brief explanation of each of the concepts mentioned.

Language and environment features

A **garbage collector** is a process that finds non-accessible memory in a programs runtime and returns it to the operating system. **Locks** are mechanisms that synchronize processes, and are used by processes to communicate, usually to agree on an ordering for resource access. **Atomic registers** are processor registers [63] which do not allow writes at the same time as other operations, as this could cause multi-bit registers to exhibit undesired states. Such operations do not overlap with other operations, and are said to be *atomic*. **Memory fences** are instructions which control how processors reorder operations. This can be useful; without memory fences processors will often reorder operations (out of program order) to improve performance. The out of order execution can cause errors if program logic depends on execution being in a specific order.

Advanced environment or library features

A program using **hazard pointers** will enforce that each process registers and deregisters any shared object it is accessing in a shared hazard pointer. Processes check that no hazard pointer is pointing to a shared object before reclaiming the objects memory. **Epoch based reclamation** is based on the idea of dividing execution into epochs, common to all processes. When the current epoch changes, memory of previous epochs can be reclaimed. **Compare-and-swap** is a motif that refers to a processor checking that an address to be written to contains some value before writing a new value to it. The compare and write (swap) must be atomic. **Multi-word compare-and-swap** primitives allow processes to write to multiple words (spread across a processor) atomically, using compare-and-swap. **Software or hardware transactional memory** refers to software or hardware techniques for encapsulating series of operations on shared memory into transactions. A transaction, once *commited*, guarantees some properties over all of the operations it contains. For example, atomicity. Transactions may be *aborted* before being committed, meaning that to all processes the state of the system will appear as if the aborted transaction had not run at all.

Common motifs

Optimistic concurrency is based on the idea of processors *optimistically* trying to complete operations. If it is decided that an operation completing would cause an error then the operation is aborted. The scheme is optimistic in the sense that it performs well when many concurrent operations can take place and only a small percentage need to be aborted (and therefore retrospectively waste processor time) [2]. **Mutual exclusion** is the concept of two processes coordinating access to some resource, so that they do not access it at the same time. This can be implemented using locks, for example. **Read-copy-update** refers to a technique for concurrent modification in pointer based structures. A writing process will copy and modify its copy of an object before linking it into the original structure in place of the original object. **Reference counting** is another commonly used technique, and refers to programs maintaining a reference count for shared objects, the count being the number of processes with reference to it.

Concurrent data structures are created using several or many of the mentioned tools and motifs, often in ad-hoc, creative ways. A full discussion of each of these techniques is out of the scope of this thesis. Concurrency techniques however, are extensively studied and written about; a good reference is [36].

2.2.2 Linearizability

Linearizability violations occur when an event sequence is not linearizable. Here we illustrate two examples of how linearizability could be violated in a concurrent balanced binary tree, should the program be faulty. The first example demonstrates a faulty execution where a process running *get* is concurrent with another process doing a rotate operation on a part of the tree (Figure 2.3). The second example is the same as given in [11] and is similar to the first example but trickier. It occurs if two processes concurrently invoke *insert*, but one process also performs rotate operations (Figure 2.4).

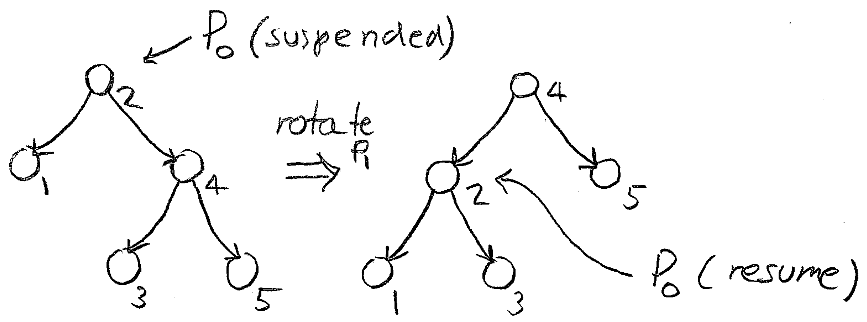

FIGURE 2.3: The state of a binary tree before (left) and after (right) a rotate. In the first frame p_0 is suspended, and hold a reference to node 2. In the second frame p_0 resumes.

Figure 2.3 illustrates an execution with a process p_0 performing $get(5)$ and a process p_1 performing a rotate, with pivot node 2. p_0 is suspended at node 2. p_1 then rotates node 2 downward in the tree, making it the left child of node 4, which used to be its right child. After the rotate completes, p_0 resumes, finding a state in which there is no traversal path from node 2, which it has a reference to, to node 5.

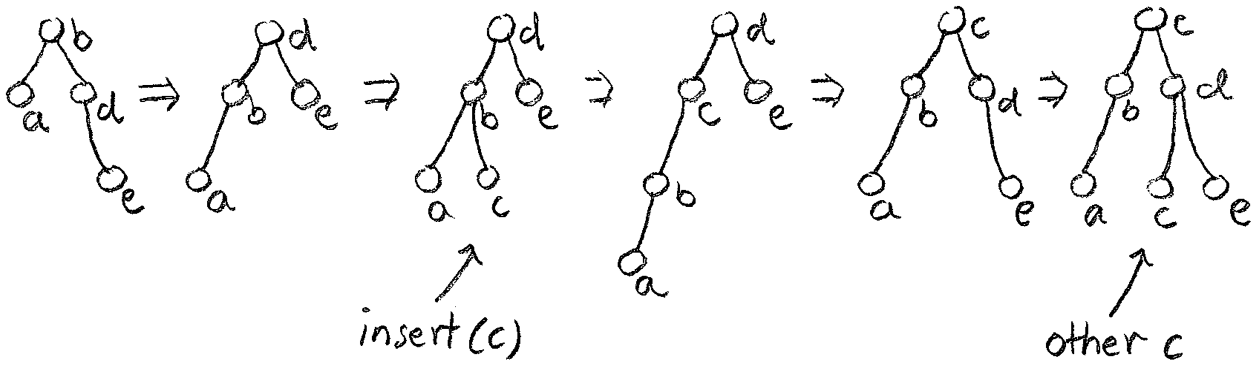

FIGURE 2.4: An illustration of n example of a tree property violation caused by concurrent behaviors. In the last frame the tree contains two nodes with key c, which is wrong.

Figure 2.4 illustrates a scenario in which two processes p_0 and p_1 are both attempting to execute $insert(c)$. In the first (illustrated) state, p_0 is suspended at node d while p_1 has not yet begun executing. In states 2, 3, 4 and 5, p_1 begins executing and executes a rotate followed by an insert and two more rotates. At the 6th state, p_0 resumes and, still holding a reference to d, inserts c as the left child of d. After this execution the tree is invalid because there are two instances of the key c.

Examples like the above are meant to illustrate the kinds of errors that can occur at the level of abstraction of the tree structure - far removed from any implementation detail.

2.2.3 Reclamation

Memory reclamation is a difficult problem for concurrent programs. Problems with reclamation can be reduced to the problem of one process wanting to reclaim some memory while another process intends to use it, either immediately or later. As discussed in Section 2.2.1, many techniques for reclamation in a concurrent settings exist.

Here is an example demonstrating a flawed reclamation scheme. This type of flaw has been found in existing concurrent tree structures [12]. Consider an object graph with three nodes $\{a, b, c\}$ and pointers $\{a \mapsto b, b \mapsto c\}$ and processes p_0 and p_1 with p_0 suspended and holding a reference to b. p_1 is running and holds a reference to a and b. p_1 decides that b and c are no longer logically in the tree and that they should eventually be reclaimed. An incorrect algorithm, considering references held by processes, might decide that only c can be immediately reclaimed, as there is a risk of p_0 resuming and accessing b immediately. While the risk exists, it's not the only one. If p_1 erases only c, an error can still occur if p_0 resumes and takes a reference to c via b. Errors like this seem obvious but can be easy to introduce.

2.2.4 Non-local graph mutations

Tree structures and especially balanced tree structures can involve complex mutations of the graph during *insert* and *erase* operations. This is especially true for balanced algorithms which need to perform additional rotate operations. Such rotate and rebalance operations can require references to several (upwards of 5) nodes and pointers [11]. The problem of how to mutate relatively large subgraphs of the structure without errors has one obvious solution: using a large region of mutual exclusion. Unfortunately taking locks for mutual exclusion on large areas of the graph is not a solution that will result in a tree competitive with state of the art trees in terms of performance. Recent trees all use techniques more sophisticated than naive locking of large graph regions [12, 20, 21, 2, 13, 41, 11, 65, 26, 27, 28]. Finding ways to perform these mutations without causing errors is one of the key challenges of concurrent tree design. The bugs we find in the BAVL tree [11] are of this form, and we describe some examples in Chapter 5.

2.2.5 Progress

Progress properties guarantee particular behaviors to processes using a data structure. A data structure with progress properties guarantees that processes can count on having fair and continued access to the data structure, independent of the actions of other processes.

Examples of progress properties concurrent tree creators may want to provide include deadlock freedom and starvation freedom [39, 67]. Deadlock freedom is the absence of the possiblity of deadlock. Deadlock is a state in the system where all processes are waiting for another process to perform a certain action before they themselves can take their next action. Starvation is a sequence of states in which a process is not able to progress even though other processes may do so. The term usually refers to when this occurs over a long period of time or in an infinite loop. In the context of the concurrent trees that we study a deadlocked state would be one in which all processes operating on the tree stop progress because they are waiting for some condition which can only be enabled by another process, such as the relese of a lock.

Here is an example of a starvation situation: consider two processes p_0 and p_1 with p_0 trying to perform a *get* operation while p_1 performs an *insert*. Suppose that the algorithm specifies that insert operations temporarily mark nodes while they perform rotate operations, and that these marks, if read by a concurrent *get* operation, cause the process executing *get* to restart its procedure. If p_1 crashes while nodes are marked, before it can unmark them, then it is possible for p_0 to keep restarting its procedure in an infinite loop.

The thesis of Brown [12], which describes techniques for designing concurrent trees, describes a possible starvation situation related to reclamation using hazard pointers. Brown argues that starvation can occur in at least 11 existing state of the art concurrent tree algorithms, violating claims made by the authors of those algorithms about their progress properties. Examples like this demonstrate that assuring progress properties is yet another challenge that structure creators have to face.

2.3 Assuring correctness

With creating concurrent data structures being so difficult it is natural that creators want to verify that their work is correct. Creators aren't the only people who care about concurrent data structures being correct: users care too of course. Since data structures are a relatively low level software component, large systems may come to depend on the data structure being absolutely correct. Moreover, the nature of many errors being caused by rare configurations of process interleavings means that errors can remain undetected for a long period of time. In fact, it has been found that concurrency bugs can be considered more severe than non-concurrency bugs [4]. They have been known to cause large financial losses and to require substantial effort to fix, with many fixes even being incorrect [23].

There is no one-solves-all approach to assuring correctness. Techniques are usually used to verify or assert correctness properties about one aspect of the structure. Several error-free verdicts, the results of analyzing different aspects, could then be taken as a strong indicator that the structure as a whole is error free.

In this section we discuss some existing approaches to assuring correctness of concurrent data structures: pen-and-paper proof, formal verification, testing, and model checking.

2.3.1 Pen-and-paper proof

Pen-and-paper proof or argumentation is commonly given as support for correctness of a structure. A typical property to demonstrate with (pen-and-paper) proof is linearizability. Proofs can in theory be given for any aspect of the structure. For example the paper of Bougé *et al.* [9] presents a pen-and-paper proof for a property of AVL trees which states that it is possible for a daemon process to keep the tree balanced even in the presence of concurrent writers.

Another pen-and-paper argument that is relevant for this work is the proof of deadlock freedom given for the BAVL tree in [11]. The argument is short and easy to understand, in contrast with many proofs in the literature. For example, a recent publication presents a general proof technique for proving concurrent tree traversals correct [29]. The publication is over 20 pages long. Another example of a complex proof is the proof given for a concurrent binary search tree structure of Ellen *et al.* [28]. The difficulty and length of the proof is dependent on the specific structure and property being demonstrated, but creating the proof always requires expert knowledge of the structure in question.

2.3.2 Formal verification

Formal verification refers to developing models of systems or algorithms in terms of unambiguous formal logic and, usually, using programs to check the logic in an automatic or semi-automatic way. One example of formal verification of a concurrent tree is the topic of the master thesis of Chen [18]. The thesis formally proves the concurrent binary search tree of [28] by adapting the pen-and-paper proof given by the authors of that tree and running it with the Prototype Verification System (PVS) [60]. Using PVS, Chen was able to find errors in the original pen-and-paper proof, which were confirmed by an original author.

Formal verification and machine assisted proofs avoid the kind of errors that occur easily with pen-and-paper proofs, but the process is notoriously time consuming [45]. Like pen-and-paper proof, it also requires an expert operator.

2.3.3 Testing

Tests are extremely effective for finding bugs in sequential programs and systems but traditional testing methods like unit testing, and to a lesser extent property based testing, usually do not explore the large number of instruction interleavings that occur in concurrent programs. This means that traditional methods of testing often fail to find concurrency bugs. Testing methods which target concurrency bugs are a well studied area [52, 68, 47] and there are many different approaches and techniques. A common property of most testing methods however, is that they cannot completely assure correctness of concurrent structures, even if they *are* able to find *many* bugs. Testing has the distinct advantage that it is easy and cost effective to carry out compared to the other correctness assurance methods that we have discussed.

2.3.4 Model checking

Model checking is a technique that usually falls under the umbrella of Formal Verification but that could also be considered a form of exhaustive testing. Our work uses TLA+ to model structures before model checking using the TLC model checker. Other model checkers and systems using model checkers have also been used to verify shared memory concurrent programs and structures. Often, a model checking system targets a specific programming language. For example CPAChecker [7] and CDSChecker [62] target C and C++ programs respectively. Model checking TLA+ specifications has the advantage that one can tune the level of abstraction of the model exactly as required; you are not required to model code-level implementation details (although you may). Alternatively, you can represent the memory model of a processor or programming language in extreme detail [15, 43].

We found a number of works related to model checking to be very relevant to our project. Some are focused on TLA+ while some are focused on model checking concurrent structures in general.

Model checking concurrent structures

The work of Vechev *et al.* [70] discusses model checking concurrent Sets in the SPIN [40] model checker. Notably their work was the first that describes a method for instrumenting models to record non-fixed linearization points. Howley *et al.* [41] describe a concurrent tree for which they don't prove linearizability points but rather deduce them using SPIN. Liu *et al.* discuss another technique for demonstrating linearizability of a structure using the PAT [54] model checker, taking as optional input a set of linearization points suspected to be correct. Finally, Lowe discusses a testing approach to checking linearizability [55]. Although he does not use model checking, the work discusses various algorithmic approaches for testing linearizability.

Model checking using TLA+

We are aware of a few previous works which explore model checking concurrent structures in TLA+. We found the most instructive of these to be the thesis of Cai [15], chapter 5 of Lamport's 'Specifying Systems' book [49] and Lamport's paper on checking a concurrent Queue [50].

The thesis of Cai describes at a high level how he was able to model a Queue algorithm and successfully replicate a known bug. Lamport's book chapter is especially informative as it follows a tutorial style of presentation and deals with shared memory. Lamport's paper on checking a concurrent Queue was very useful, as he discusses some of the problems that we faced during this work. The problems are discussed in detail in Chapter 4.

Of the works mentioned, Lamport's paper on checking a concurrent Queue was the closest to ours. All of the works mentioned deal with structures that are far simpler than the balanced concurrent trees that we model, both in terms of the operations that they support and in terms of the implementation complexity.

2.3.5 This work relative to the state of the art

We have seen that there are a large number of challenges to be overcome in creating correct concurrent data structures. Following our discussion of methodologies for assuring correctness it is clear the trend on the spectrum of methodologies is that more effective and rigorous techniques are significantly more time consuming and difficult to carry out.

While our methodology is more time consuming than using basic tests to check for correctness, it is also significantly more effective at finding certain classes of errors than most forms of testing, and our methodology also offers a unique feature: it provides an error trace if a bug is found. In comparison to full formal verification, our methodology offers a weaker assurance of correctness than a well designed formal verification proof would, however, our methodology is significantly easier and less time intensive to carry out.

Chapter 3

Concurrent tree data structures

This section describes the specific concurrent tree data structures that we model and design with our methodology. All the data structures that we study are based on the sequential AVL tree algorithm [30]. Our work focuses on one existing concurrent variant of the AVL tree and various implementations of it as well as our own concurrent tree which adds concurrent read operations to the tree of Fynn *et al.* [31].

The AVL tree algorithm is a balanced binary search tree which can implement a Map ADT (see Chapter 2 for a discussion of balanced binary trees and ADTs). The existing concurrent variant of the AVL tree that we study is due to Bronson *et al.* [11]. We refer to that tree as the BAVL tree. The tree due to Fynn *et al.* is a sequential AVL tree which extends the original AVL algorithm with additional structures called chunks, which are groupings over nodes. We call that tree the CAVL tree.

This section illustrates the principles of the algorithms which are important to understand in order to understand our model checking models and methodology. We proceed by describing the AVL, BAVL and CAVL trees in turn. At the end of the section we briefly discuss how the BAVL and CAVL trees compare to other existing AVL tree variants in the literature. A detailed description of our novel concurrent additions to the sequential CAVL tree is in Chapter 5.

3.1 AVL Trees

The AVL tree [30] is a classic algorithm in computer science literature. Recalling Chapter 2, the AVL tree is a balanced binary search tree offering $\mathcal{O}(\log n)$ running time complexity for *get*, *insert* and *erase* operations, where n is the number of nodes currently in the tree at time of invocation of the operation. The AVL tree defines balance to mean that the heights of the left and the right child of all nodes differ by no more than 1 (where 0 is taken for the height of a missing child). Balance has to be restored after *insert* and *erase* operations and is done by modifying small regions of the tree structure in *rotate* operations.

The *get* operation in the AVL tree works by traversing pointers from the root of the tree downwards towards the leaves. In the AVL tree it is required that all nodes are marked with a key from some total order. Users of the tree associate keys to values and the tree is organized such that traversing processes can use the key argument of the $get(k)$ operation to locate the sought value by taking either the left or right child of each node encountered during traversal, based on the total order. The $insert(k, v)$ operation works by traversing, as in the *get* operation, to a leaf node and attaching a new node with a handle to v to the found leaf. The $erase(k)$ operation work like an *insert* operation in reverse, traversing to the found leaf and detaching it from its parent.

Care has to be taken in the case of *insert* and *erase* operations as the actual linking in or detaching of nodes can cause the parent to have only 1 child. Fixing this requires rotations, which generally speaking have to be done in several places. Ensuring that all necessary rotations are performed is usually done by the process traversing from the site of the linking or detaching back up to the root and performing necessary rotations along the way. A detailed description of the AVL algorithm is out of the scope of this thesis, but one can be found in [46].

3.2 A Practical Concurrent Binary Search Tree

The Practical Concurrent Binary Search Tree of Bronson *et al.* [11], or the BAVL tree for short, is a concurrent variant of the original AVL tree algorithm. The algorithm allows an arbitrary number of readers (processes executing *get* operations only) and writers (processes executing *insert* and *erase* operations) to execute concurrently. We use the shorthand MRMW (multi-reader, multi-writer) to denote this kind of property.

The BAVL tree does not only implement *get*, *insert* and *erase* operations but also offers additional operations which are useful in practice, such as ordered traversals of the underlying Set and concurrent copying, among others. The traversals are based on the total ordering of the key domain used, and the copying uses a combination of epochal and read-copy-update techniques (see Chapter 2 for a description of these). We focus on the *get*, *insert* and *erase* operations making up the Map ADT implementation.

Th BAVL tree uses various strategies to deal with some major challenges that we highlighted in Section 2.2. We briefly explain how the algorithm deals with a) linearizability invalidations caused by concurrent rotations, b) reclamation, c) locality of graph mutations, d) deadlock freedom.

3.2.1 Linearizability invalidations caused by concurrent rotations

The BAVL tree avoids the kinds of problems discussed in the section Section 2.2.2, which gives examples of the kinds of errors which can be caused by incorrect handling of rotations. The solution presented by Bronson *et al.* is to use optimistic concurrency with version numbers. All nodes in the tree are given an atomic *version number* counter which counts the number of times that the node has been subject to a so-called *shrink*. Optimistic concurrency and atomic counters are discussed in Chapter 2. They are also explained in detail in [36].

Shrinking refers to the change that occurs in the position of a particular node during rotate operations. A node is said to *shrink* if the size of the set of leaves that are reachable from the node after the rotate decreases when compared to the size of the set of leaves reachable from it before the rotate. This is a useful thing to identify because it characterizes precisely the problem shown in Figure 2.3. Figure 2.3 shows a situation in which a node, '2', shrinks.

The BAVL tree uses the *version number* to avoid shrinks using optimistic concurrency. Traversing processes use hand-over-hand transactions, without locks, commiting transactions against version numbers and rolling back in the case of a shrink (as shrinking increases the version number). In the original BAVL algorithm the transaction's commits are each contained in a single stack frame. A rollback returns from a method call and the algorithm retries from the previous stack frame. Likewise, successfully committed transactions push a new stack frame.

The entire code needed for safe traversals is reasonably complicated. Listing 17 shows the source code for the attemptGet method of one of our implementations of the original BAVL algorithm.

The motif of hand-over-hand optimistic transactions has two important components. The first component consists of reading the child of a node given that the version number of the node is known. After reading the child, the version number of the *node* is read again. If it has not changed then the read of the child is validated.

The second component is more subtle, and is necessary for the hand-over-hand transactions. It ensures that the version number of the child is read safely, as it will be needed to make a safe traversal from the child to any child of the child, possible in the future. After reading the child of a node, the version number of the child is read. Then, it is checked that the child is still the child of the node, before checking that the version number of the node has not changed.

Listing 1 shows a simplified version of the traversal algorithm, supposing that the intended direction of traversal from a node is always to the right child.

```
# Arguments
# n: a node
# nVer: the version of number of n read sometime in the past
#       the time of the read marks the beginning of one open transaction
traverse(n, nVer){
    c = n.right # read the right child of n in order to traverse to it later
    cVer = c.ver # read the version number of the read child, start a new transaction
    if c != n.right: # check that the read child is still the right child of n
        return # abort the new transaction
    if nVer != n.ver: # check if the version number of n has changed
        return # n has shrunk, abort the new transaction
    traverse(c, cVer) # commit the transaction that begun when nVer was read
                      # also traverse deeper into the tree
}
```

LISTING 1: Pseudocode showing a simplified version of the traversal algorithm used in the BAVL tree. The simplified algorithm assumes that the intended direction of traversal is always to the right child.

By ensuring that processes that execute rotates also increase the version number of any shrunk node, any traversing process that is suspended and then resumed as described in Section 2.2.2 (and illustrated in Figure 2.3) will surely encounter a version number mismatch and thus rollback to the previous stack frame (up the tree). The process will then resume downward traversal.

3.2.2 Reclamation

The BAVL tree relies on a garbage collector for reclamation. Using a garbage collector avoids many of the problems discussed in Section 2.2.3, as it pushes reclamation entirely to the environment. This is the approach that we take in all of the implementations that we model.

3.2.3 Locality of graph mutations

The section Section 2.2.4 discusses how balanced trees often use very fine grained regions of mutual exclusion in conjunction with rotations and other graph mutations. The BAVL tree takes a coarser approach compared to entirely lock-free algorithms. To ensure that mutating operations never cause errors in the presence of concurrency, the BAVL tree associates locks to all nodes and locks each node needed to perform a *rotate*, *insert* or *erase*. Locks are taken on the minimum number of nodes necessary and are always taken (released) as late (early) as possible. In this work we demonstrate that the scheme used is not entirely sufficient to guarantee balance in all situations. Example executions that demonstrate an error are given in Chapter 5.

Aside from the handling of rotations there is another challenge related to locality of graph mutations that the BAVL tree handles. The problem revolves around the physical unlinking of logically erased nodes. Consider an $erase(k)$ operation for which the node marked with key k is not a leaf in the tree. In the case that the node has two children the BAVL algorithm *marks* the node as logically deleted but does not physically unlink it. To avoid a gradual accumulation of logically deleted nodes all writing processes opportunistically unlink logically deleted nodes that only have one child. The opportunistic unlinking always takes place in conjunction with an *insert*, *erase* or *rotate*, once the process has taken the relevant locks.

3.2.4 Deadlock freedom

Deadlock freedom is defined in Section 2.2.5, and is an important property of the BAVL algorithm to verify because locks are used. The BAVL algorithm is not known to admit deadlocks. A supporting argument is given in the paper of Bronson *et al.* [11]. Summarizing: the lock of a node is always taken only by a process holding 0 locks, or by a process holding a lock on the direct parent of the node to be locked. In this way, locked regions are always contiguous regions of the graph and a region always expands downwards in the tree. This scheme means that no cyclic dependency can exist between processes requiring locks.

3.2.5 Balance

The usual concept of balance used for sequential AVL trees does not transfer directly to concurrent AVL trees. This is because concurrent writers can mutate different regions of the tree – as no process can have a global view of the tree at any one point in time, no process can be sure that the entire tree is balanced.

The definition of balance used for concurrent trees is a rephrasing of the usual statement given for sequential trees. We define a concurrent tree to satisfy the *relaxed balance* property if it holds that the tree is balanced following the usual definition at any time that no operation is executing on the tree. A period in which no operation is executing on the tree is called a *quiescent interval*. Even though the original BAVL tree paper [11] claims that the tree satisfies the property, in our work we found that the tree in fact violates it. Example executions that demonstrate an error are given and elaborated on in Chapter 5. Several trees do exist which are **not** known to violate the property, for example [26, 21].

3.3 Chunked AVL Trees

The Chunked AVL tree (CAVL) due to Fynn *et al.* [31] is a variant of the classic sequential AVL tree algorithm that extends the tree with additional structures called chunks. Chunks are used to store leaves of the tree; the values of the implemented Map are stored in the leaves. This 'chunking' of the tree is useful as it provides another way to access leaves (and therefore values) without needing to traverse the tree from the root node. Moreover, chunks are instances of a Chunk data structure which has a simple interface similar to a Map. This means that varying Chunk designs could allow a large spectrum of possible implementations, which in combination with the AVL tree could offer a spectrum of properties. For example, enhanced performance for certain kinds of operations, or additional operations that would be difficult to implement directly over a classic AVL tree.

The Chunked AVL tree was originally designed for use in a Byzantine fault tolerant distributed ledger system [61, 16] and our work was initially in part motivated by a wish to augment the tree with concurrency to improve performance. Due to the trees' use case in public Byzantine networks we required the algorithms to be absolutely correct. This requirement was a strong incentive for us to develop what is now the primary contribution of this thesis; a formal correctness assurance methodology.

We proceed by briefly describing the original use case for the CAVL tree; a structure for use in Byzantine fault tolerant distributed ledgers. After that we illustrate the chunking mechanism. For the discussion we assume a working understanding of the AVL algorithm. Small sections of our models of the CAVL trees are discussed in Chapter 4, and a detailed description of our novel concurrent MRSW (multi-reader single-writer) algorithms are given in Chapter 5, rather than in this chapter. Chapter 6 discusses possible directions for future work on the concurrent CAVL trees.

3.3.1 A data structure for Byzantine fault tolerant distributed ledgers

The original CAVL tree due to Fynn *et al.* was created for use in a blockchain [61], or distributed ledger system; the Tendermint system [61, 16, 10]. Augmenting the classic AVL algorithm with additional chunk structures containing the leaves means that the tree can be serialized and sent over a network connection chunk by chunk. Moreover, a process receiving chunks can reconstruct a tree given the full set of chunks. The chunking and chunk-wise serialization and deserialization is useful in blockchain systems where network participants all maintain their own local version of the tree. This is because a new participant who joins the network and wishes to construct their own version of the common tree can request different chunks making up the tree from different established network participants. This reduces the work load for individual established participants. Moreover, the tree was shown by Fynn *et al.* to improve performance on a range of tasks such as serialization and deserialization compared to the non-chunked AVL tree.

3.3.2 Chunks

The core concept in the CAVL tree is the chunking. Chunks are data structures which hold a set of tree nodes as well as an additional pointer to an additional single node. The additional pointed-to node is called the chunk root. The CAVL tree maintains some properties relating chunks

- The union of the sets of leaves held by chunks is exactly the set of leaves in the tree.
- The number of leaves in each chunk is bounded by some constant.
- The leaves of each chunk are the leaves of a subtree of the whole tree that is rooted at the chunk root.
- No chunk root is a descendant of another.

An important property emerges from the invariants, which is that the ordered set of keys of leaves of each chunk is a contiguous segment of the ordered set of keys of all leaves in the tree. Figure 3.1 shows some example illustrations of CAVL trees.

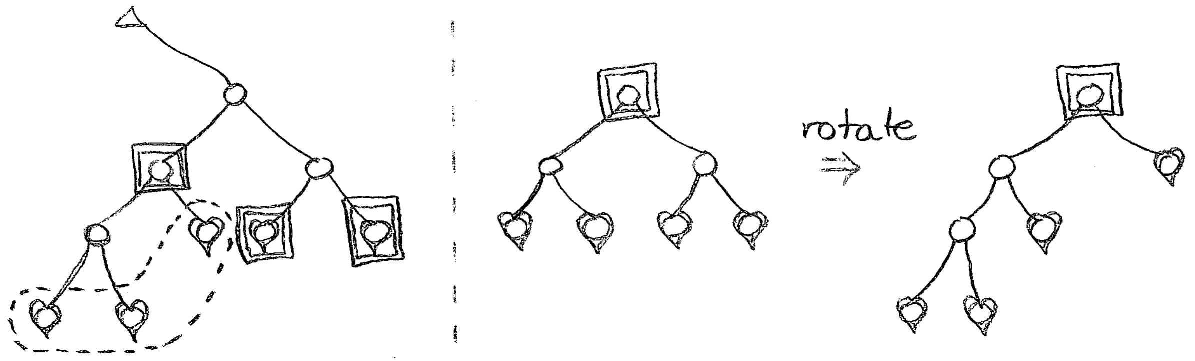

FIGURE 3.1: Left: an example of an entire CAVL tree with 9 nodes excluding the sentinel ‘rootHolder’ (triangle). Chunk roots are marked with double square outlines, leaves are marked with heart shapes and the leaves belonging to a common chunk are encircled in a dotted line. Note that a chunk root can be a leaf in the case of a chunk of size 1. Right: an example of a rotate in a subgraph of a CAVL tree (all the leaves belong to the same chunk).

To maintain its invariants the CAVL tree modifies the algorithms used in the *insert* and *erase* operations of the classic AVL. Care must be taken to ensure that the invariant which bounds the size of the chunks is maintained after *insert* operations, and that the third and fourth invariants constraining the shape of the chunks and the positions of the chunk roots is maintained by *rotate* operations, which are necessary to maintain balance. The idea behind the adaptation to maintain the chunk size constraint is to perform *chunkSplit* operations during downward tree traversal if a chunk is at its maximum size. The *chunkSplit* operation creates two chunks from one chunk. The idea behind the adaptation for the invariant relating to the shape of the chunks is to move leaves from one chunk to another during *rotate* operations. A full and detailed description of the original CAVL algorithms is contained in [31].

3.4 Comparison of the BAVL and CAVL trees with other concurrent trees

The BAVL and CAVL trees both differ from the classic AVL tree algorithm in that they both utilize nodes which are part of the structural tree but that do not have an associated value. In the BAVL tree nodes without values have been logically erased but not yet unlinked (see Section 3.2.3). In the case of the CAVL tree, all non-leaf nodes are such nodes, as only leaf nodes have a pointer to an associated value. This usage of nodes without associated values is common in the concurrent tree literature.

(Concurrent) AVL trees can usually be categorized as being internal, external or partially external. Internal trees are trees for which all nodes have an associated value, external trees are trees for which only the leaf nodes have associated values and partially external trees are trees for which *some* non-leaf nodes do not have associated values. The CAVL tree is an example of an external tree, while the BAVL tree is an example of a partially external tree.

State of the art concurrent binary and AVL trees can usually be classified along a spectrum that has the degree to which the tree is external on one axis and progress guarantees on the other axis. The BAVL tree is a blocking tree, that is, it does not claim to provide non-blocking guarantees such as lock-freedom [36, 37]. The original CAVL tree, being sequential, does not have a relevant position on the spectrum. The MRSW CAVL tree that we present in this thesis does not offer a non-blocking progress guarantee. The tree does not use traditional locks (such as Java monitors) but is nonetheless not lock-free (or even obstruction-free [36]), this is explained in Chapter 5. Indeed, arguments for non-blockedness require careful thought. As mentioned in Section 2.2.5, the thesis of Brown [12] argues that a number of trees in the literature do not truly satisfy technical definitions of lock-freedom.

Table 3.1 shows the position of a few state of the art concurrent trees along the spectrum. Not all of the trees are balanced trees.

YEAR	PAPER	EXTERNALITY	PROGRESS
2010	Non-blocking Binary Search Trees [28]	external	lock free
2012	Fast Concurrent Lock-Free Binary Search Trees [65]	external	lock free
2013	A Fast Contention-Friendly Binary Search Tree [21]	partially external	blocking
2014	Practical Concurrent Binary Search Trees via Logical Ordering [26]	internal	blocking
2014	Efficient Lock-free Binary Search Trees [17]	internal	lock free
2015	CASTLE [64]	internal	blocking

TABLE 3.1: The externality of the tree and the progress property as claimed in the original paper. Note that the progress property is the property of the structure as a whole; individual methods may have stronger progress guarantees.

Chapter 4

A methodology for modelling and analysing concurrent tree data structures in TLA+

Our methodology is a sequence of hands-on steps that can be applied to model and verify concurrent structures, in order to find bugs, or to assert the absence of bugs. By applying the methodology it is possible to design concurrent structures that can trusted to be correct, to a very high degree. It is also possible to check existing structures, either in order to check for bugs, or to debug known problems. Should a bug exist, our methodology can provide an error trace containing the entire state of the system leading up the error – this is an invaluable resource for developers who are designing and repairing algorithms.

The methodology is intended to be a pragmatic one, that practitioners can adapt and use in real projects. Although we study concurrent tree data structures, the techniques we use should be transferable to other data structures. We use our work on modelling the BAVL tree to guide our exposition of the methodology, and make it concrete. In relevant places we will also describe parts of our models of the CAVL trees.

Developing and checking complex models with TLA+ is subject to two categories of problems. The first category is practical problems. These are the same problems that developers face in many medium to large scale software projects and relate to day-to-day usage of languages and tooling, and developer productivity. The second category is conceptual problems. These are problems that relate to deciding how to model a particular system so that model checking is tractable and effective in terms of finding errors.

This chapter is split into three parts. Section 4.1 gives a roadmap of our methodology. Section 4.2 contains details on how to go about solving the conceptual problems that are part of applying the methodology. Section 4.3 contains a discussion on the more practical issues related to using tools in the TLA+ ecosystem, that we encountered while applying the methodology.

4.1 A high level overview of the methodology

A valuable model of a concurrent tree in TLA+ should be computationally tractable and also effective at finding errors. Due to the state space explosion problem [69], the two goals are opposed, and tradeoffs must be made. A model that simulates the real world more closely will have a larger state space than one that is more abstract, and require more computing power to check.

Much of the methodology revolves around finding ways to reduce the state space of the model, while retaining the ability to find errors. Additional desired properties of a model checking methodology include the ability to link detected errors to the original system. In our case this means linking errors such as violations of linearizability or balance properties to behaviors in data structures at a design level. To assist with this, we use a trace visualiser that we developed.

In this section we first broadly describe the abstraction level of the models that we write, and introduce Pluscal, an intermediary language that is automatically translatable to TLA+. Following that, we explain the major components of the methodology at a high level.

4.1.1 Using TLA+

A TLA+ specification uses the notations of first order logic and set theory, together with a small number of temporal logic statements, to express systems as series of actions. While it's possible to express a huge variety of systems succinctly using this style [49, 71], our methodology focuses on translating models written in the Pluscal language [51] to TLA+, before model checking.

Pluscal

Modelling primarily in Pluscal, a pseudocode-like language, has a number of benefits that make it well suited to checking shared memory data structures. Most of the errors that we are checking for can be caused by bytecode or assembly instruction interleavings between processes. Due to the sheer size of most concurrent tree algorithms, in terms of the number of instructions, and the fact that the model checking state space increases with the number of possible of interleavings of instructions betweens processes, and thus exponentially with the number of instructions, we must sacrifice granularity of the model to make model checking tractable. The instruction granularity that we use is similar to Java source code. We usually model one or a few (1-10) lines of Java code, or the equivalent, as one action in Pluscal. Our Pluscal code closely resembles Java because of this. This has a benefit of making the resulting models more easily approachable for non-TLA+ users.

With the level of granularity that we use, the state space explosion problem [69] is still the limiting factor of our approach in terms of the scale of the models that we can check; we are however still able to find critical bugs.

A whirlwind tour of TLA+ and Pluscal

Hands on experience using the TLA+ ecosystem and Pluscal would be required for anybody wishing to directly develop their own models borrowing our techniques, but it should be possible for anybody familiar with basic programming notions to understand the methodology as we present it in this chapter. Readers who would like more details on physically writing and running TLA+, Pluscal and TLC may read Appendix A, 'A TLA+ and Pluscal whirlwind tour; modelling a re-entrant lock', which gives a compact introduction to the ecosystem.

4.1.2 Major components of the methodology

Our methodology, at a high level, revolves around making as many facets of the data structures as possible abstract, while still maintaining the ability to tractably be able to detect errors. We check for many of the errors described in Chapter 2 and Chapter 3, either explicity or implicitly. We explicitly check for structural property violations and linearizability violations. The way that we write our Pluscal models, and the way that TLC model checks mean that many other errors are implicitly checked for. For example, type errors, some instances of null dereferences, and some progress guarantees.

The methodology in short

The methodology consists of roughly 7 major steps

1. **Abstract what can be abstracted** from the data structure and the environment in order to shrink the state space while still maintaining the ability to detect errors.

2. **Create a source code mirror of the model** that mimics Pluscal and TLA+ as closely as possible. Translate the source code to Pluscal and TLA+.

3. **Record operation invocations and responses** in order to check linearizability. In the model, designate actions in which a process is sure if its operation has succeeded or failed.
4. **Define invariants** for your data structure, for example, relaxed balance. Implement operators for them which can be checked by TLC.
5. **Create initial configurations for the data structure** to start model checking from. This step may not be necessary if the model is already very computationally tractable.
6. **Run the model checker**.
7. **Visualize error traces** found by TLC in order to quickly link errors to problems in the algorithm or model.

Here we outline each step in further detail.

1: Abstract what can be abstracted

We abstract aspects of the data structures such as the operating system scheduler, process memory and addressing, locks, and more. The goal of this step is to reduce the size of the state space generated by the model in order to make model checking tractable.

2: Create a source code mirror of the model

We find that it's useful to have a source code version of the model: Java, in our case. This is for practical reasons; Java IDE integration and development speed is extremely good, whereas developing TLA+ and Pluscal models is comparatively difficult. We find that we have the highest productivity when we maintain Java code that mimics Pluscal and translate the Java code to Pluscal before model checking.

3: Record operation invocations and responses

We check linearizability by recording operation sequences in a bookkeeping variable and checking the sequences for linearizability during model checking, using TLA+ operators overridden with Java code. It is not necessary to define linearization points to do this step.

4: Define invariants

It is possible to check graph properties using TLA+ operators without increasing the size of the state space. We use this to check structural properties, including the relaxed balance property.

5: Create initial configurations for the data structure

Using initial data structure states to begin model checking from has numerous advantages including improving model tractability. We find that this approach also improves development speed by allowing for faster iteration on models and algorithms.

6: Run the model checker

We find it useful to arrange our Pluscal and TLA+ code in a way that means we can easily configure model checking runs for different target simulations and to check for different errors.

7: Visualize any error traces

We wrote a basic but functional TLC error trace visualizer which allows us to debug error traces very efficiently, either leading to bugs in the model itself or in the actual data structure design.

A note on the steps

The steps described summarize the major components of our methodology. In practice we did not follow the steps in linear order but rather iterated over parts of each step one or several times in order to obtain models which were tractable and effective. The presentation follows a sequence as it makes individual steps easier to understand.

The following section, Section 4.2, discusses conceptual problems that we tackled while developing our models, and links the problems to the steps in this section. Section 4.3 does the same for practical problems encountered, linking them to the steps.

4.2 Conceptual problems

Of the steps discussed in 4.1.2, it is steps 1, 3, 4 and 5 that are the most conceptual. This sections discusses the motivation behind the steps in detail, and explains how we applied them to our models.

4.2.1 Tackling complexity and intractability with abstraction

The trees that we model are quite complex. Our shortest version of the BAVL tree, that only implements a Map, is 684 lines of Java code in total. The original Java BAVL tree due to Bronson is 2237 lines of code, although it includes some extra utility functions. Our sequential version of the CAVL tree, implemented in Java, is 388 lines of code. The shortest version of one of our MRSW CAVL trees is 452 lines of Java code. Measuring lines of code is a very primitive way to measure complexity, but knowing the numbers does give some sense of scale. Beyond lines of code, complexity comes from many places. For instance, the CAVL trees are far more complex due to the additional Chunk data structure that is used, and the additional bookkeeping implied, compared to a standard AVL tree. Although we do not present a MRMW CAVL tree in this thesis, it is safe to say that the algorithm would be more complex than any existing MRMW AVL tree.

It's clear that, to make running the models tractable, it's necessary to abstract as many parts of the algorithms as possible, ideally without sacrificing the ability to catch errors. In this subsection we describe major abstractions that we made in our models.

Modelling memory, data and pointers

Listing 2 shows constant operator definitions and variable definitions that are used to model memory, data and pointers in our BAVL tree models. We model memory addresses with a set of integers, and `Node` object fields are represented as a function mapping addresses to values.

This scheme is advantageous in that there is little bookkeeping overhead required which could expand the model checking state space. Using a more realistic representation of memory may find more errors, however it would require more state. The thesis of Cai [15] explores an approach which does model memory in a more nuanced way, but that work only models a concurrent Queue.

```
NodeAddressSet == 1..14

// ..

RootHolder == 0
AddressSetWithoutRootHolder == NodeAddressSet
AddressSetWithRootHolder ==  AddressSetWithoutRootHolder \cup {RootHolder}

// ..

IntegerShift == 100
Null == IntegerShift + 1
VersionNumberInitValue == 0

// ..

    variables

        ver           = [e \in AddressSetWithRootHolder    |-> VersionNumberInitValue];
        key           = [e \in AddressSetWithRootHolder    |-> Null];
        val           = [e \in AddressSetWithRootHolder    |-> Null];
        height        = [e \in AddressSetWithRootHolder    |-> Null];
        parent        = [e \in AddressSetWithRootHolder    |-> Null];
        left          = [e \in AddressSetWithRootHolder    |-> Null];
        rite          = [e \in AddressSetWithRootHolder    |-> Null];
```

LISTING 2: A snippet of our BAVL tree model showing the representation of memory, data and pointers.

Modelling reclamation

We do not model any kind of reclamation scheme, instead allowing memory which could be reclaimed to remain 'dirty'. We call a memory address *dirty* if there is some variable for which the value of the variable at the address is not the initial value (we use integer values `Null` or `VersionNumberInitValue`).

Not modelling a reclamation scheme explicitly means that we avoid the state that would be created by its simulation. The disadvantage is that dirty memory itself can generate extra state which has no value for correctness assurance. For example, two states that are exactly equivalent in terms of all of the information visible to processes, would differ if the sets of dirty but unreachable addresses were not identical in each state. This would lead to twice the number of states being model checked than is really necessary. We will see that the step Section 4.1.2 is critical to mitigating the impact of this effect.

Modelling the scheduler

We model the operating system process scheduler using fair processes in Pluscal. Pluscal defines the actions of a fair process to be weakly fair, meaning that an action will be eventually be taken if is continuously possible to do so. This behavior is the behavior that is interesting for concurrent data structures as they are created assuming that the operating system does not indefinitely suspend processes for no reason. Lines 1-11 in Listing 8 show the definition of a fair reader process in our BAVL model.

Modelling locks

We represent a re-entrant mutual exclusion lock using a function that maps addresses to either a process identifier or a null value. Listing 3 shows macros used for locking and Listing 4 shows a snippet of the `attemptNodeUpdate` method from our model of the BAVL tree. The usage of the lock abstraction in the method simulates Java `synchronized` blocks. Actions that lock an address are made strongly fair. A strongly fair action enjoys a stronger guarantee than a weakly fair action; strongly fair actions will eventually be executed given that the action is infinitely often enabled, as opposed to requiring the stronger condition that it be continuously enabled, as in the weakly fair case.

```
    locked        = [e \in AddressSetWithRootHolder    |-> NullModelValue];
    // ..
macro lock(Maddr){
    await locked[Maddr] = NullModelValue \/ locked[Maddr] = self;
    locked[Maddr] := self;
}
macro unlock(Maddr){
    locked[Maddr] := NullModelValue;
}
```

LISTING 3: A snippet of our BAVL tree model showing macros used by processes to lock and unlock an address.

Modelling re-entrant locks in this way increases the size of the state space. More sophisticated locks would likely require more bookkeeping and thus generate an even larger state space.

```
1  procedure attemptNodeUpdate(ANU_newValue, ANU_parent, ANU_node)
2      // ..
3  {
4  // ..
5  anuLock0:+
6          lock(ANU_parent);
7  anu2:
8              if (IsUnlinked(ver[ANU_parent]) \/ parent[ANU_node] # ANU_parent) {
9  anu3:
10                 retry();
11                 unlock(ANU_parent);
12                 return;
13             };
14 anuLock1:+
15             lock(ANU_node);
16 anu4:
17                 ANU_prev := val[ANU_node];
18                 if (ANU_prev = Null) {
19 anu5:
20                     ret[self] := ANU_prev;
21                     unlock(ANU_node);
22 anu6:
23                     unlock(ANU_parent);
24
25                     operation_fail(); \* Erase - element absent
26                     return;
27                 };
28 anu7:
29                 call attemptUnlink_nl(ANU_parent, ANU_node);
30 anuT0:
31                 if (ret[self] = FalseInt) {
32 anu8:
33                     retry();
34                     unlock(ANU_node);
35 anu9:
36                     unlock(ANU_parent);
37                     return;
38                 };
39 anu10:
40             unlock(ANU_node);
41 anu11:
42             call fixHeight_nl(ANU_parent);
43 anu12:
44             ANU_damaged := ret[self];
45         unlock(ANU_parent);
46 }
```

LISTING 4: A snippet of our BAVL tree model showing the usage of lock abstraction replacing Java 'synchronized' blocks. The snippet shows the usage of locks in part of the `attemptNodeUpdate` method. Locking actions (lines 15-16 and 24-25) are made strongly fair.

Modelling version numbers

Version number locks, also called seq-locks (sequence number locks) [36], are a commonly used tool for implementing speculative or optimistically concurrent behaviors. The BAVL tree and our MRSW CAVL tree use this kind of lock, as described in Chapter 3 and Chapter 5.

There are two major challenges related to modelling version numbers. The first problem is how to model them. In theory, version numbers should be unbounded and monotonically increasing totally ordered values. Real systems use bounded fixed-width integer version numbers and let them wrap on overflow. Either of these approaches will lead to the same problem as described in Section 4.2.1, namely that more than one state can be created which are equivalent from a correctness assurance perspective. We take a simple approach, modelling version numbers as monotonically increasing and starting at 0. Using initial data structure configurations can mitigate the impact of creating additional states. Section 4.2.4 elaborates on this point.

Modelling chunks

Our implementations of the sequential CAVL tree, and our Java implementation of the MRSW CAVL, use sequential tree-based Maps to implement the Chunk data structure itself. In our models we do not attempt to model the chunks themselves in full but instead model chunks using a Set. This drastically reduces the complexity of our models (and thus the size of the state space).

Modelling the Chunk structure as a Set lead to making a decision that was was a typical example of a tradeoff between model complexity and the degree of correctness assurance attainable. The Chunk structure in the MRSW CAVL algorithm includes a *shift* method that transfers nodes between the calling Chunk and another. The Chunks themselves could be implemented in many ways, so 'transfer' could mean either that one Chunk directly takes ownership of the memory of the other, or that the transferred nodes are copied. In our models we use the copying approach. Chapter 5 contains a description of the Chunk algorithms.

There are two options for modelling the *shift* operation on top of an underlying Set, that use different granularities of action atomicity. The first option is to transfer all the nodes that are to be transferred in one atomic step, while another option is to transfer each node in its own atomic step. We opted for the second choice, with the nodes being copied in several atomic steps and the original nodes being erased from their original chunk in one atomic step. This incurs more process interleavings during model checking and improves the degree of correctness assurance attained, as the model more closely mimics the real implementation. Listing 5 shows Pluscal code that contains one of the *shift* operations of the Chunk structure.

```
 1  procedure chunk_shift_rite(CSR_chunkAddr, CSR_minKeyToMove)
 2      variables
 3          CSR_shift_seq;
 4          CSR_i;
 5  {
 6  csr0:
 7      CSR_shift_seq := SetToSeq({e \in left_wing[CSR_chunkAddr] : CSR_minKeyToMove <= key[e]});
 8      CSR_i := 1;
 9  csr1:
10      while(CSR_i <= Len(CSR_shift_seq)){
11          chunk_insert_rite(
12              UnusedLeafNodeAddr,
13              CSR_chunkAddr,
14              key[CSR_shift_seq[CSR_i]],
15              parent[CSR_shift_seq[CSR_i]]
16          );
17  csr2:
18          CSR_i := CSR_i + 1;
19      };
20      left_wing[CSR_chunkAddr] := left_wing[CSR_chunkAddr] \ SeqToSet(CSR_shift_seq);
21      return;
22  }
```

LISTING 5: The chunk `shiftRite` procedure of our model of the CAVL tree. The procedure defines two important actions. Action 1 (lines 6-8) defines the set of node addresses which are to be transferred. Action 2 (lines 9-16 and 20-21) performs the copying and erase of the nodes.

Using a Set instead of a Map

It's a good idea to consider whether or not there are aspects of your data structure that are not worth modelling. For example, if it is the case that your data structure can very easily be converted from a Map to a Set, and vice versa, then it may not make sense to make your model more complex by using a Map. We take this approach with the models of the CAVL tree. We do not use the same approach with the BAVL tree because it overloads the `val` field of nodes to implement logical erasure: using a Set instead of a Map would require reworking parts of the algorithm significantly.

Eliminating recursion

Pluscal simulates a call stack in TLA+, which adds to the size of a single state. This can lead to the same problems described in the discussions on modelling reclamation and version numbers. It is worth considering if it possible to make recursive operations in the modelled data structure iterative (without storing an additional stack). Both the BAVL and MRSW CAVL algorithms use a recursive hand-over-hand locking scheme that can be *modified* and made iterative. In Chapter 5 we compare models that use full recursive hand-over-hand locking with simpler iterative versions. The models are not equivalent because some subtle semantic properties are lost: the hand-over-hand locking can go forwards, but cannot go backwards in increments. The iterative versions, however, are much faster to model check with TLC than the recursive ones. Therefore, unless the lost semantic properties of a particular structure are especially important to verify, using an iterative version is likely a good idea.

There is an additional problem that can occur when using recursive operations in models: an infinitely growing call stack due to loops in the data structures graph. A technique that is sometimes used in concurrent balanced tree algorithms is to temporarily create loops in the graph during rotate operations. This is useful because it means that traversing processes will loop in the same region of the graph, instead of, for example, causing a segmentation fault or otherwise doing the wrong thing. This is fine in practice for implementations, given a progress guarantee on the rotating process, as the traversing process will stop looping once the rotate is complete and the probability of exhausting stack memory due to excessive recursive calls is very low. When using model checking however, the model checker will certainly explore this looping behavior and *will not prune the state space, because the simulated call stack is not constant*. This will lead to a non-terminating model checking run that is very hard to detect and debug.

Modelling one read operation only

If certain operations of the data structure strictly only read data then modelling more than one instance of the operation will not discover more errors, as more race conditions will not be triggered. This is because TLC will execute all possible orderings of the single read operation. We apply this reasoning to the *get* operation of our BAVL and CAVL tree models, and only model one read operation. The driver process that executes a single read for the BAVL tree is shown in lines 1-11 of Listing 8. We always define `Readers` to contain a single process identifier; the use of a set is nothing more than a code design choice.

4.2.2 Recording operation invocations and responses

In concurrent data structure implementations there is usually no straightforward way to record the execution of linearization points, even if linearization points can be easily defined. This could be, for example, because the linearization point of an operation is dynamic and defined at runtime, possibly as a function of the behavior of several processes. This is the case in the BAVL tree, due to the way the *erase* operation works.

Here's a concrete scenario. In the BAVL tree it is possible for the linearization point of a *get* operation to dynamically depend on a concurrent *erase*. Figure 4.1 shows an example execution between three processes p_0, p_1 and p_2 executing $erase(k)$, $get(k)$ and $insert(k)$. Chronologically p_1 reads a pointer to k, after which p_0 unlinks k from the tree and sets the value of k to `Null`. Following that, p_2 inserts a new node with key k. Finally p_1 reads the value of the version of k that it has a pointer to, the value is `Null`. In this instance the linearization point of p_1s *get* is the linearization point of p_0s *erase*.

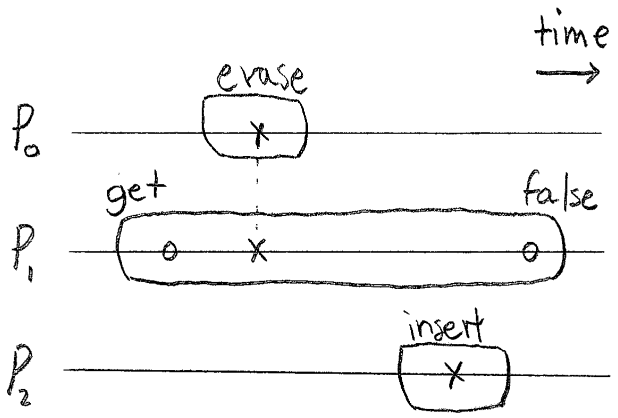

FIGURE 4.1: An event sequence diagram showing processes p_0, p_1 and p_2s execution over time. Linearization points are marked with an 'x'. The two circles inside the wedge belonging to p_1 represent the reading of a node and the reading of the nodes values, respectively.

```
1  variables
2
3      event_sequence = <<>>;
4      operation_succeeded = [p \in Procs |-> Null];
5
6  // ..
7
8  macro invoke(Moperation_name, Marg){
9      event_sequence := event_sequence \o <<<<"invoke", self, Moperation_name, Marg>>>>;
10 }
11
12 macro respond(Msucceeded_int_bool){
13     assert IsIntBool(Msucceeded_int_bool);
14     event_sequence := event_sequence \o <<<<"respond", self, IntBoolAsBool(Msucceeded_int_bool)
        ↪  >>>>;
15 }
16
17 (*
18 Get succeeds if the key was present (and not marked deleted)
19 Erase succeeds if the key was present and update removes it
20 Insert succeeds if the key was not present and update inserts it
21 *)
22 macro operation_succeed(){
23     assert operation_succeeded[self] = Null;
24     operation_succeeded[self] := TrueInt;
25 }
26
27 macro operation_fail(){
28     assert operation_succeeded[self] = Null;
29     operation_succeeded[self] := FalseInt;
30 }
```

LISTING 6: A Pluscal snippet common to all our models which declares variables used to record the operation invocation and response event sequence. The assertion on line 13 helps catch errors related to not properly marking an operation as succeeded (or failed).

Keeping track of linearization points like this in an adhoc manner would require a lot of bookkeeping and lessen the generality of our methodology. Instead, we record an event sequence of operation invocations and responses between processes and feed them to an external linearizability checking algorithm, written in Java, that is wrapped by a TLA+ operator.

Listings 6, 7 and 8 show the event sequence variable and relevant macros, an example of a procedure recording a success or failure for an *insert* or *erase* operation, and finally driver processes which are used to execute operations and record the moments of their invocations and responses.

Processes use macros `invoke` and `respond` to append events to the `event_sequence` sequence variable. In TLA+ each action is executed atomically, so the `event_sequence` variable does not itself need to be a concurrent List.

```
1  procedure attemptNodeUpdate(ANU_newValue,
2                              ANU_parent,
3                              ANU_node)
4      variables
5          ANU_prev;
6  {
7
8  // ..
9
10 anu16:
11         ANU_prev := val[ANU_node];
12
13         (* -- START of bookkeeping section (not part of algorithm) -- *)
14         if(ANU_newValue = Null){
15             if(ANU_prev = Null){
16                 operation_fail();  \* Erase fail
17             }else{
18                 operation_succeed(); \* Erase succeed
19             }
20         }else{
21             if(ANU_prev = Null){
22                 operation_succeed(); \* Insert succeed
23             }else{
24                 operation_fail();  \* Insert fail
25             }
26         };
27         (* -- END of bookkeeping section (not part of algorithm) -- *)
28
29         val[ANU_node] := ANU_newValue;
30
31 // ..
32
33 }
```

LISTING 7: A snippet of the BAVL tree model which shows a part of the attemptNodeUpdate procedure. The procedure is used for both *insert* and *erase* operations, with the argument `Null` being passed to the variable `ANU_newValue` to indicate erasure. Not part of the BAVL tree algorithm itself are lines 13-27 which simply calculate whether or not the operation succeeded or failed based on the previous value.

Listing 7 contains an instance of an action in a procedure used by *insert* and *erase* that demonstrates how a process records the success or failure of its operation. We found that it was usually possible to incorporate this logic without introducing additional actions (which would significantly expand the state space).

The `verifier` process in Listing 8 has the sole job of calling a TLA+ operator `IsLinearizable`, which is written in Java and included using TLCs operator override feature. The second argument to the operator, `reachable_at_start`, stores data about the initial state of the data structure before the first event. This is necessary in combination with the use of initial data structure configurations.

Checking the linearizability of an event sequence is NP-complete; however, we simply brute force each possible sequential permutation of the event sequence. This is sufficient in practice as the number of operations we execute is always small (< 10), and hence using a sophisticated algorithm would not bring a measurable performance benefit.

```
 1  fair process (reader \in Readers)
 2  {
 3  readInv:
 4      await pc["initializer"] = "Done";
 5      with (e \in KeySetToOperateOn){
 6          invoke("get", e);
 7          call get(e);
 8      };
 9  readResp:
10      respond(operation_succeeded[self]);
11  }
12
13  fair process (writer \in Writers )
14  {
15  writeInv:
16      await pc["initializer"] = "Done";
17      with (e \in KeySetToOperateOn){
18          either{
19              invoke("insert", e);
20              call update(e, e);
21          }
22          or {
23              invoke("erase", e);
24              call update(e, Null);
25          }
26      };
27  writeResp:
28      respond(operation_succeeded[self]);
29  }
30
31  fair process (verifier = "verifier")
32  {
33  verif:
34      await \A p \in Readers  : pc[p] = "Done";
35      await \A p \in Writers  : pc[p] = "Done";
36      assert IsLinearizable(event_sequence, reachable_at_start);
37  }
```

LISTING 8: A snippet showing the driver processes for the BAVL tree models, not including the initial state configuration process `initializer`. Three processes are used. The `reader` process (line 1) and `writer` process (line 13) are thin wrappers around Map operations. The `verifier` process (line 31) is not part of the BAVL tree but is only used to call the `IsLinearizable` operator. It only has 1 action.

4.2.3 Defining invariants

In the case of a balanced tree like the AVL tree and its derivatives it is important to verify that the tree is indeed balanced when it supposed to be. Recalling Chapter 3, the *relaxed balance* property states that a concurrent AVL tree should be balanced during any quiescent interval. We check for this constraint using TLA+ operators.

```
1  // ..
2  EXTENDS Naturals, FiniteSets, TLC, Sequences, Linearizability, TreeGeneration, TreeStructure
3  // ..
4  \* BEGIN TRANSLATION (chksum(pcal) = "b5f78852" /\ chksum(tla) = "ecefdcd0")
5  // ..
6  \* END TRANSLATION
7
8  VersionNumbersAreBounded == \A x \in Range(ver) : x <= VersionNumberBound \/
   ↪  IsShrinkingOrUnlinked(x)
9
10 CallStackSizesAreBounded == \A p \in Procs : Len(stack[p]) < CallStackLimit
11
12 SanityChecks == CallStackSizesAreBounded /\ VersionNumbersAreBounded
13
14 QuiescenceGuarantees ==
15
16                     LET
17
18                         IsQuiescent(p) == pc[p] \in
19                             {
20                                 "writeResp",
21                                 "Done"
22                             }
23
24
25                     IN  (\A p \in Writers: IsQuiescent(p)) =>
26                         (
27                         /\ IsCycleFree(Null, left, rite, RootHolder)
28                         /\ IsBalanced(Null, left, rite, RootHolder)
29                         )
30
31 AllInvariants == /\ SanityChecks
32                  /\ QuiescenceGuarantees
```

LISTING 9: A snippet showing invariant definitions for our BAVL and CAVL models.

Listing 9 shows operator definitions placed underneath the translation of a Pluscal model. The operators call other operators which are defined in a module `TreeStructure`. Lines 14-29 of the snippet show the definition of the `QuiescenceGuarantees` operator. The operator asserts that whenever the structure is in a quiescent state, the two properties `IsCycleFree` and `IsBalanced` hold. `IsCycleFree` and `IsBalanced` are defined in the `TreeStructure` module. Listing 10 shows the code for the `IsCycleFree` operator.

```
 1  IsCycleFree(Null, left, rite, RootHolder) ==
 2
 3      LET
 4          (*
 5          Takes pairs <<x,y>> and <<y,z>> to <<x,z>>
 6          *)
 7          Closure(R, S) ==
 8          \* R**S ==
 9              LET
10                  T == {rs \in R \times S : rs[1][2] = rs[2][1]}
11              IN {<<x[1][1], x[2][2]>> : x \in T}
12
13          NodesOf(R) == { r[1] : r \in R } \union { r[2] : r \in R }
14
15          (*
16          Transitive closure, taken from Lamport's hyperbook, S 9.6.2
17
18          Takes a set of relations. A set whose elements are of the from <<a, b>>
19          meaning a |-> b.
20          *)
21          TransitiveClosure(R) ==
22
23              LET
24
25                  RECURSIVE STC(_)
26                  (*
27                  (1 edge) u (2 edges) u (3 edges) u ...
28                  *)
29                  STC(n) == IF n=1
30                              THEN R
31                              \* ELSE STC(n-1) \union STC(n-1)**R
32                              ELSE STC(n-1) \union Closure(STC(n-1), R)
33
34              \* By starting with n+1 then we include (full length) cycles
35              IN IF R={} THEN {} ELSE STC(Cardinality(NodesOf(R))+1)
36
37
38          FullTransitiveClosure ==
39
40              LET
41
42                  NonNullRelations(R) == { r \in R: r[2] # Null }
43
44                  Relations(f) == {<<x, f[x]>> : x \in DOMAIN f }
45
46              IN TransitiveClosure(
47                  NonNullRelations(Relations(left)) \cup NonNullRelations(Relations(rite))
48              )
49
50          TransitiveClosureFromRootHolder ==
51              {r \in FullTransitiveClosure : <<RootHolder, r[1]>> \in FullTransitiveClosure}
52
53      IN \A r \in TransitiveClosureFromRootHolder : r[1] # r[2] \* There is no cycle
```

LISTING 10: The `IsCycleFree` operator defined in the `TreeStructure` module. The operator takes as argument a value that is used to represent `Null`, as well as functions `left`, `rite` and an address `RootHolder`. Line 53 is the main operator body while lines 4-51 define helper operators.

The `IsCycleFree` operator is used to check that the graph reachable from the `RootHolder` does not contain cycles. This is a property that should hold independently of the relaxed balance property, as the presence of a cycle is an error on its own.

The operator works by building the transitive closure of the graph (lines 4-48) and filtering out nodes which are not reachable from the RootHolder (lines 50-51) This is important as in general the graph represented by the `left` and `rite` variables contains multiple components. Line 53 then checks that the transitive closure does not contain any self edges.

Using operators to check structural properties does not expand the model's state space but can add moderate constant runtime overhead; this is discussed in Section 4.3.

4.2.4 Creating initial configurations

Launching model checking from non-zero/non-empty data structure *initial configurations* has many advantages that we found to be crucial to making our large models tractable. We have already discussed how not explicitly modelling reclamation and letting version numbers monotonically increase can lead to unwanted expansion of the state space over time. Using initial configurations mitigates this effect, and has other advantages too.

One advantage is that it provides the ability to target model checking at particular situations. This is useful for reproducing errors but it is also an effective means of speeding up the development cycle. The ability to target particular situations for model checking, and the reduction in the state space size, mean that we can reach and model check algorithm executions that would be intractable to model if beginning model checking from a zero-state. This increases the degree of correctness assurance that our methodology can achieve.

Using initial configurations for our BAVL tree models allowed to us *find errors that we would not have been able to find otherwise*. This is because executing all the rotation code instructions requires fairly large trees (7+ nodes) and more than one process. The size of the model checking state space that would be necessary to process before reaching such paths, starting from an empty structure, would make model checking completely intractable. As such, for sufficiently complicated data structure it likely necessary to use initial configurations. This of course has a downside, namely the additional development cost required to implement generating initial configurations.

We implement initial configurations using TLCs operator overload feature. We use Java code to generate 'interesting' AVL trees, and an `initializer` driver process to select a tree for model checking. Listing 11 shows the `initializer` process for our BAVL tree models. Line 4 calls the overridden operator `SetOfInterestingAVLTrees`, with configurable arguments. Although not shown, we also implement an `InterestingAVLTree` operator which allows the selecting of a specific tree. The `reachable_at_start` variable stores the set of reachable nodes, which are used for linearizability checking.

The set of 'interesting' AVL trees is chosen to only contain trees which differ from one another in a way that makes model checking each tree in the set uniquely valuable. Precisely, no two trees are right/left reflections of each other, and also no two trees differ only by some scalar factor on the absolute values of the keys of their nodes. The result of this is a large reduction in the number of initial configurations that model checking is run from. For example, the total number of AVL trees with node keys in 1..10 is 2706, while the 'interesting' subset has only 123 trees.

```
fair process (initializer = "initializer")
{
init0:
    with (starter_state \in SetOfInterestingAvlTrees(TreeGenKeyMin, TreeGenKeyMax,
    ↪   TreeGenSizeMin, TreeGenSizeMax)){
        key := starter_state.key @@ key;
        val := starter_state.val @@ val;
        left := starter_state.left @@ left;
        rite := starter_state.rite @@ rite;
        height := starter_state.height @@ height;
        parent := starter_state.parent @@ parent;
        reachable_at_start := starter_state.reachable;
    }
}
```

LISTING 11: A BAVL model snippet showing the `initializer` driver process. The process has 1 action and chooses an arbitrary initial tree configuration (`starter_state`) from the set of configurations returned by the `SetOfInterestingAvlTrees` method.

4.3 Practical problems

While the previous section focused on conceptual problems related to modelling concurrent trees, this section is a comparatively shorter discussion of some of the most important practical aspects of applying our methodology.

Applying any of the steps of our methodology listed in Section 4.1.2 involves many practical considerations common to any programming project. In this section we focus on things that are particularly interesting and relate to model checking, TLA+ and TLC.

4.3.1 Abstraction and symmetry

The TLC model checker can make use of user defined *symmetry* to prune states from its search. A symmetric set is a set of model values (unique identifiers) that TLC will use to identify symmetric states in its queue. For example, if two processes p_0 and p_1 are members of the symmetry set $S = \{p_0, p_1\}$ then TLC will compare pairs of states and prune those which are identical up to a permutation of members of S. In the ideal case, for this example, half of all explored states could be pruned.

While applying symmetry is very effective at reducing the size of the state space, it does not always shorten the time needed for model checking. This is because there is a large overhead involved in comparing states, especially if the size of the symmetry set S is large or if elements of S appear many times in each state being processed.

We considered applying symmetry in two ways to our models of the BAVL and CAVL trees. The first type of symmetry that we considered was symmetry on the set of addresses used to model memory. In our earlier models we found this worsened performance rather than improving it. This happened because both the set of addresses used, and the number of occurrences of addresses in each state were large. The other type of symmetry that we considered was symmetry on the set of processes. This is a common technique when using TLA+. We found that applying symmetry to the set of processes used was effective at reducing model checking time for our models, for a small number of processes only (1-3).

While we found using symmetry on the set of processes to be effective for a small number of processes (1-3), there are some caveats involved in writing models to support symmetry. While TLA+ does not have types, TLC does, and can only check comparable types for symmetry while model checking. This means, for example, that if a model contains a function satisfying $p_0 \mapsto 42, p_1 \mapsto \{43\}$, then TLC will throw an exception when it tries to check symmetries over the function. We use Integers in many places in our models for this reason.

4.3.2 Writing the models in parallel with a source code implementation

Section 4.1.2 mentions the value of writing models together with a source code implementation. We used Java, which has many advantages. First of all, many implementations of the BAVL tree are written in Java, so it was useful to be able to directly adapt existing implementations. An additional major advantage of Java relates to the `volatile` keyword. A `volatile` variable in Java gives variables full-visibility and happens-before guarantees [3, 56]. This drastically simplifies writing Pluscal, in comparison to how it would have to be written in order to mimic C or C++ code.

A further property of the Java language itself, that helps to simplify modelling, is the fact that Java reference/pointer updates are atomic by default. This again reduces the complexity of modelling significantly, compared to the complexity that would be required to explicitly model a more complex pointer update mechanism.

Tips for using a Java mirror implementation

Ideally we would have developed a Java to Pluscal translator to assist with developing our models, but such a project was far outside of the scope of this thesis. We were able to maintain a Java mirror implementation of our models reasonably effectively by making the most of IDE features. In this subsection we briefly discuss a few tips that helped us develop quickly.

Mimic Pluscal in Java as closely as possible before making changes to the actual Pluscal code. Java code is much easier to change than Pluscal due to how feature rich IDEs for Java are. We found it useful to do the following, for example,

- Prefix variable names with method names in order to avoid name clashes in Pluscal.
- Structure logic to avoid `break` and `continue` statements, in order to streamline the translation to Pluscal.
- Explicitly initialize variables where possible, for example write `boolean x = false;` instead of `boolean x;`, and hoist variable declarations to the beginning of methods. Java language features which are implicit are easily overlooked when translating to Pluscal.

Judiciously simplify Pluscal logic to reduce state, but do not overdo it so as to hamper trace debugging. It can be wise when writing Pluscal to structure things in a way that reduces the number of unique states created. This can be done, for example, by combining actions if you judge them to be unimportant for assuring correctness. By doing this you can inadvertently give up the possibility of finding an error in your algorithm, but you can also make debugging traces more difficult. This can happen for example if actions become fragmented over the code. Listing 12 shows a small example of this.

```
1  label0:
2      if(var = 0){
3  label1:
4          var := 1;
5      }else if(var = 1){
6          done := TRUE;
7      };
```

LISTING 12: A snippet of Pluscal code that shows an action split over several lines. The action defined by `label0` consists of lines 2 and lines 5-7 and the action defined by `label1` consists of line 4 only.

4.3.3 Carefully choose and implement invariant operators

Section 4.1.2 mentions the value of using invariants to check structural properties of data structures, and Section 4.2.3 gives an example: the `IsCycleFree` invariant. It is important to note that complex recursive operators do not typically have good performance in TLC (certainly compared to say, optimized imperative C++ code). Listing 9 shows the invariants that we used in our models. We did not always use all of the invariants as we found that using all of the invariants could as much as double model checking time. In fact, the `IsCycleFree` invariant (Listing 10) that we use is an example of quite a slow operator. We accepted a performance hit by using this operator.

In addition to invariants, TLA+ allows *properties*, or temporal invariants. We disregard this feature completely as using temporal operators would make model checking completely intractable.

4.3.4 Running the model checker

We wrote our TLA+ models in a way that allows quick configuration of various aspects of the model. Those aspects include, for example, parameters used for initial state configuration, the number of processes modelled and which data structure operations they perform, and which invariants are checked. In addition to this style, we also utilized features of TLC to improve the performance of our model checking runs. These included tweaking the number of OS threads and amount of system memory to allocate to model checking – TLC scales particularly well with more memory. Other useful features are the ability to use depth-first model checking and simulation modes to run the models. Depth-first model checking will quickly explore sequences of actions all the way up to the linearizabilty check and termination. This is a good way to quickly check the model for any basic errors. When we ran depth-first mode, if no error had been found after a while then we usually stopped the run and switched to the default breadth-first mode.

Simulation mode does not exhaustively check all action interleavings so it can't be used for correctness assurance; however, it is a useful way to quickly check models for any simple mistakes in the model itself.

In addition to using various features of TLC to sanity check our models, we also used Lincheck [47], a JVM library for checking linearizability of data structures. Lincheck does not offer an error trace if an error is found, but it runs very fast and we usually used it on our Java implementations to rule out basic errors before running TLC.

4.3.5 Caching data structure initial configurations in Java

We cached the set of 'interesting' AVL trees that we generated using Java code. This is because the operator call shown in lines 4-5 of Listing 11 is executed more than once, in general.

4.3.6 Visualize error traces to quickly determine the cause of errors

As mentioned in Section 4.1.2, in this project we used a TLC error trace visualizer tool that we wrote, to quickly determine the cause of errors found during model checking. Figure 4.2 shows screenshots of our tool, which is web based and takes a JSON format error trace as input.

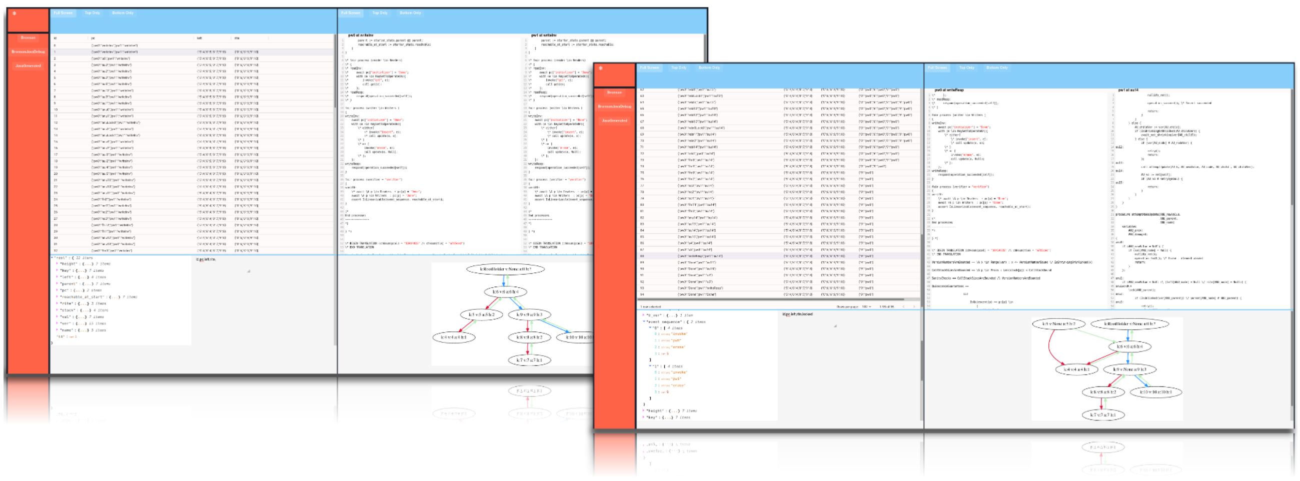

FIGURE 4.2: Web based visualization tool for debugging TLC state traces.

The tools interface shows four panels which each display different information useful for debugging. The top-left panel shows the error trace in a compact table form, and can be used to select particular states from the trace. The other panels always display a rendering of some aspect of the chosen state. The top-right panel shows the Pluscal source code of each processes next action at the chosen state. The bottom-right panel shows a visual representation of the tree graph, including information like the memory address, and the height, key and value of nodes. The last, bottom-left, panel shows the entirety of the selected state, which allows easy inspection of every single variable in the state. This is analogous to a program debugger which allowed you to see, for some breakpoint, every bit at every layer of the stack running the programs process.

Chapter 5

Results

In this chapter we present the major results of the thesis. The first major result is a methodology for designing concurrent tree data structures. The other major results are successful applications of the methodology. Namely, we used the methodology to develop a new concurrent AVL tree variant; the multi-reader single-writer (MRSW) CAVL tree. We present this CAVL tree in this chapter. Additionally, we applied the methodology to an existing concurrent AVL tree, the BAVL tree, and were able to find critical bugs in the tree's design, which we were able to successfully replicate in various implementations.

This chapter begins with a description of our work on the MRSW CAVL tree, and is followed by a discussion on applying the methodology to the BAVL tree. We wrap up by discussing scalability aspects of the methodology, as well as limitations and lessons learned. Elaborating

Section 5.1 contains a presentation of our MRSW CAVL tree algorithms, including a discussion of the TLA+ models and implementations.

Section 5.2 describes some of the bugs that we found in the BAVL tree algorithm. The section also includes a list of implementations for which we have either successfully replicated a bug, or have been able to identify incorrect code.

As Chapter 4 contains a detailed description of the methodology itself, Section 5.3 focuses on our findings in terms of the scalability of the methodology.

Section 5.4 contains a description of major limitations of our methodology. The section also discusses things that we learned over the course of this work, and gives two points of major advice on applying the methodology.

5.1 Concurrent multi-reader single-writer CAVL trees

In this section we present a multi-reader single-writer version (MRSW) of the CAVL tree. The MRSW CAVL tree has the following traits

- The tree implements a Map.
- The tree is external: only the leaf nodes have handles to values.
- The tree is a concurrent AVL tree: at any time there can be an arbitrary number of processes executing *get* operations concurrently with each other and 0 or 1 processes executing *insert* and *erase* operations.
- The tree provides the blocking progress guarantees *deadlock freedom* and *starvation freedom* according to the definitions of Herlihy *et al.* [36].
- Due to the use of version number locks, the tree does not provide any non-blocking progress guarantees (*obstruction freedom*, *lock freedom* or *wait freedom*).
- The tree does not use any traditional locks offered by the operating system or language, for example, Java reentrant locks (`synchronized` block) [3] or any POSIX Threads locks [59].

In this section we first discuss the algorithm at a high level. This includes a proof outline of deadlock freedom and starvation freedom but does not discuss a proof of linearizability. Following that we discuss our models and implementations of the tree. The section is concluded by a discussion of the limitations of the tree, models and algorithms. Possible avenues for future work are discussed in Chapter 6.

5.1.1 The MRSW CAVL Tree algorithm

We designed two MRSW CAVL trees that differ only in the implementation of the *get* operation. As discussed in Section 4.2.1 it can be useful to eliminate recursion from algorithms to improve model checking. Our two trees differ in the type of recursion used in the *get* operation. The difference is explained in full shortly. First of all we give a very high level description of the core algorithm common to both variants. Following that we explain the difference in our two variants.

A high level overview of the core algorithm

The MRSW CAVL tree is similar to the original sequential CAVL tree due to Fynn *et al.* [31] but it supports optimistic concurrent readers using the techniques used in the BAVL tree. This integration required some modifications to the sequential CAVL algorithm, which made it possible to safely add concurrency. We believe that these modifications will also make the tree more straightforward to adapt into a multi-reader multi-writer version in the future. Chapter 6 discusses such a possible extension in more detail.

The algorithm works by augmenting the classic external AVL tree with chunks. A chunk is an instance of the Chunk data structure. The Chunk of the sequential CAVL is introduced in Section 3.3.2. Everything discussed in Section 3.3.2 also holds for the MRSW CAVL tree Chunk, but the Chunk in the MRSW CAVL tree is used in a conceptually different way and is implemented completely differently.

The concurrent *get* operation of our algorithm is almost exactly the same as the *get* operation in the BAVL tree – the only difference is that the *get* must traverse all the way to a leaf before it can read a value. This is because the MRSW CAVL is external while the BAVL tree is partially external.

We first of all discuss the *get* operation. Following that we discuss the Chunk structure.

Concurrent *get* operations

In the canonical BAVL tree algorithm, the *get* operation uses hand-over-hand locking which can rollback in a step-by-step manner (see Section 3.2). As briefly mentioned in Section 4.2.1, it is possible, and can be useful for model checking, to alter the *get* operation so that rollbacks do not rollback only a single transaction but instead restart the entire operation. Brown [12] finds that the performance impact of this kind of change is difficult to predict. On the one hand, rollbacks are usually a rare occurrence, and hence restarting the *get* operation is likely to only rarely be necessary. Therefore the cost of restarting is usually only actualized infrequently. On the other hand, step-by-step rollback may better utilize processor caches: nodes that are likely to be read by a traversing process have a higher chance of occupying the cache when using step-by-step rollback [12].

```
1      Optional<Optional<V>> getImpl(K k, Node node, char dirToC, long nodeOVL) {
2          while (true) {
3              Node child = child(node, dirToC);
4              if (child == null) {
5                  if (node.ver != nodeOVL) {
6                      return Optional.empty();
7                  }
8                  return Optional.of(Optional.empty());
9              }
10             K childKey = child.k;
11             if (childKey.equals(k) && child.isLeaf()) { // Check if leaf
12                 return Optional.of(child.v);
13             }
14             long childOvl = child.ver;
15             if (isShrinking(childOvl)) {
16                 if (node.ver != nodeOVL) {
17                     return Optional.empty();
18                 }
19             } else if (child != child(node, dirToC)) {
20                 if (node.ver != nodeOVL) {
21                     return Optional.empty();
22                 }
23             } else if (node.ver != nodeOVL) {
24                 return Optional.empty();
25             } else {
26                 Optional<Optional<V>> res = getImpl(k, child, appropriateDirection(childKey
27                         , k), childOvl);
28                 if (res.isPresent()) {
29                     return res;
30                 }
31             }
32         }
33     }
```

LISTING 13: A snippet showing the `getImpl` method of the recursive Map based MRSW CAVL tree that is used in the implementation of *get*. The method is very similar to the `attemptGet` method of the canonical BAVL tree implementation due to Bronson [11]. Line 11 contains an important difference, namely, checking that a child is a leaf node before returning its value.

Our two variants of the MRSW CAVL tree differ only in that one implements step-by-step rollback using recursion, while the other is iterative and restarts the entire *get* operation when optimistic progress fails. Aside from this change, both variants are exactly the same, and the rest of the *get* operation that does not deal with rollback is the same as that used in the BAVL tree, with one minor difference: the process executing *get* must traverse all the way to a leaf before it can read a value, because the MRSW CAVL is external while the BAVL tree is only partially external.

Listing 13 shows Java code for the `getImpl` method of the recursive MRSW CAVL tree that uses step-by-step rollback.

```
1     void rotateLeft(Node pivot, Node pivotParent, char directionFromParent) {
2
3         assert (!pivot.rite.isLeaf());
4
5         if (pivot.isChunkRoot()) {
6             pivot.chunk.shiftLeft(pivot.rite.k);
7             pivot.chunk.root = pivot.rite;
8             pivot.rite.chunk = pivot.chunk;
9             pivot.chunk = null;
10        } else if (pivot.rite.isChunkRoot()) {
11            splitLeftWingIntoNewChunk(pivot.rite);
12        }
13
14        long pivotOvl = pivot.ver;
15
16        // Mark pivot to indicate the beginning of a structural rotate
17        pivot.ver = beginChange(pivotOvl);
18
19        Node newPivot = pivot.rite;
20        Node childOfNewPivot = newPivot.left;
21        newPivot.left = pivot;
22
23        if (directionFromParent == 'L') {
24            pivotParent.left = newPivot;
25        } else {
26            pivotParent.rite = newPivot;
27        }
28
29        pivot.parent = newPivot;
30        newPivot.parent = pivotParent;
31        pivot.rite = childOfNewPivot;
32        childOfNewPivot.parent = pivot;
33
34        // Increment the version number of pivot to indicate the end of a structural rotate
35        pivot.ver = endChange(pivotOvl);
36
37        adjustNodeHeight(pivot.left);
38        adjustNodeHeight(pivot);
39    }
```

LISTING 14: A snippet showing the `rotateLeft` method used in both variants of the MRSW CAVL tree. Lines 17 and 35 mark the beginning and end of the structural rotate by marking and increment the version number of the pivot, which shrinks.

In order to safely implement optimistic readers it is necessary to modify the implementation of write operations to use version number locks, so as to avoid the *shrinking* problem described in Section 3.2. The important change is in the `rotateLeft` and `rotateRite` operations. Listing 14 shows Java code for the `rotateLeft` method. Lines 5-12 of the method are related to operations on chunks, while lines 14-38 implement the structural rotate and associated bookkeeping updates. The actual rotate is guarded by a version number lock transaction beginning at line 17 and ending at line 35. Concurrent readers rollback if certain steps of the traversing algorithm conflict with the transaction. They may also be blocked while lines 17-35 are being executed. This is the reason that the tree does not offer non-blocking progress guarantees.

Chunks

We first explain chunks from a conceptual point of view and then we explain the implementation that we used for our Java MRSW CAVL trees. Figure 5.1 shows a visual representation of the tree, including chunks.

Conceptually a Chunk is a structure that holds leaf nodes from the tree. The tree is then partitioned into a collection of such *chunks*. In implementations in languages with automatic memory management, such as Java, the tree and the chunk may both hold owning references to leaves in the chunk. In implementations using languages without automatic memory management, such as C++, the leaves are owned strictly by either the chunk or the tree. In the rest of the discussion we assume that the chunks *own* the leaves, while the tree just points into the chunk's leaves from internal nodes, as is the case in our non-concurrent C++ implementation of the tree.

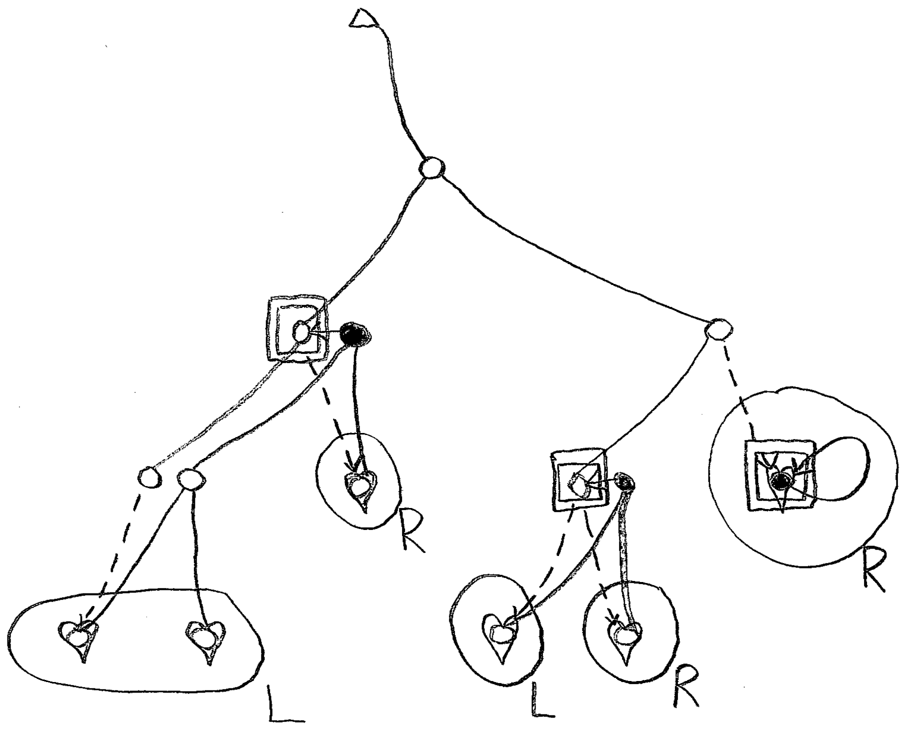

FIGURE 5.1: An MRSW CAVL tree with three chunks. The triangle represents the sentinel root holder of the entire tree and the root of the tree proper is its right child. Hearts represent leaves. Boxed nodes mark chunk roots and shaded nodes are the actual root nodes that chunks use to access leaves. The dotted lines represent pointers from nodes owned by the MRSW CAVL tree itself into nodes owned by chunks. Leaves which are grouped together and marked with 'L' or 'R' make up the left and rite wings of chunks.

Furthermore, we assume that the Chunk implements a Map, with some additional operations. This is not strictly necessary but it simplifies the discussion. In this discussion we illustrate chunks as tree based Maps, although any kind of Map implementation could be used. The Chunk structure also references a left and right side. We name these sides 'wings' and refer to the 'left wing' and 'rite wing'[1] of the Chunk. Recalling Section 3.3.2, the Chunk structure has a reference to some node in the whole tree that is the root of the subtree that contains all its leaves. The root is called the 'chunk root'. The left wing of the chunk

[1]We use 'rite' in the place of 'right' all across our implementations and models as using a word that is the same length as 'left' makes editing source code easier.

contains leaves whose key is strictly less than the key of the chunk root while the rite wing contains leaves whose key is greater than or equal to the key of the chunk root.

The breaking of chunks into left and rite wings allows the convenient implementation of rotates. In addition to rotate operations rebalancing the tree, they must maintain the chunk invariants described in Section 3.3.2. By dividing chunks into a left and rite wing, it is possible to implement rotates using only operations which transfer nodes between the left and rite wings of chunks, and an operation that creates a new chunk. The operation that creates a new chunk is the same as is used in the *insert* operation when a chunk has reached its maximum size.

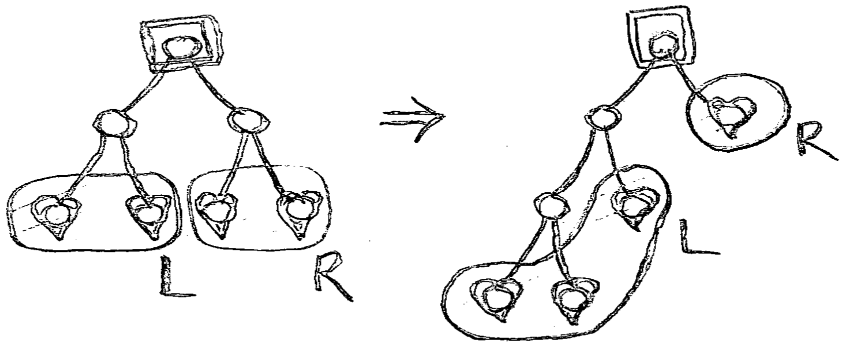

FIGURE 5.2: Illustration of the `rotateLeft` method for the MRSW CAVL in the case that the pivot node of the rotate is a chunk root. Boxed nodes are chunk roots and grouped nodes belong to the same wing of the chunk they belong to. Leaves are drawn only for illustrative purposes; leaves may be far away in the tree from the site of the rotation. The `rotateLeft` method shift leaves from the rite wing of the pivot into the left wing. The chunk root also changes: after the operation it is the root of the subtree.

Listing 14 shows Java code for the `rotateLeft` operation of the MRSW CAVL tree. Lines 5-12 implement the part of the rotate that is related to reorganizing chunks. There are three possible cases: either the pivot node is a chunk root, the right child of the pivot is a chunk root, or neither is the case. The cases are mutually exclusive, as no chunk root is a descendant of another.

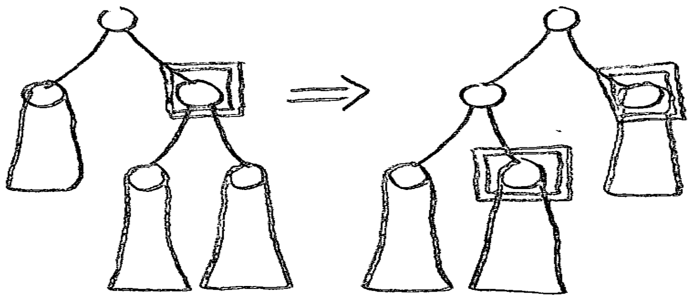

FIGURE 5.3: Illustration of the `rotateLeft` method for the MRSW CAVL in the case that the right child of the pivot node is a chunk root. Boxed nodes are chunk roots, and only chunk roots are marked explicitly. Columns represent arbitrary descendant trees. In the case that the right child of the pivot is a chunk root, the chunk is split. The resulting two chunks have roots which are the left and right child of the original right child of the pivot.

The handling of the cases ensures that all chunk invariants hold after the rotate. Figure 5.2 illustrates the first case (in which the pivot node is the chunk root). In Figure 5.2 leaves are close to the chunk roots only for illustrative purposes. The case in which the right child of the pivot is a chunk root is illustrated in Figure 5.3. In that illustration only the chunk roots are marked explicitly. The final case, that neither the pivot nor its right child are chunk roots at the beginning of the rotate, needs no explicit handling: all chunk invariants are maintained in this case.

Aside from rotates, the *insert* and *erase* operations of the tree must account for chunks. For brevity, we only give a few key points about how *insert* and *erase* work.

The *insert* and *erase* algorithms carefully account for concurrent readers and do not use locks (traditional or version number based) at all, except for during rotations, where they use a version number lock. The physical *insert* and *erase* are more straightforward for the MRSW CAVL tree than they are for the BAVL tree as the MRSW CAVL tree is external. This means that structural erasures and inserts always occur at the leaves or near the leaves. Our algorithm relies on a delayed reclamation algorithm, as most concurrent data structures do. We have implemented the tree in Java and use the garbage collector.

Deadlock freedom and starvation freedom

Following the definitions of Herlihy *et al.* [36], the MRSW CAVL tree provides deadlock and starvation freedom. These guarantees rely on a fair scheduler – one that does not indefinitely halt any process. The tree is deadlock free as the only locks are the version number locks which are only controlled by one process. The tree is starvation free as readers may only block in the case that a writer has (version) locked a node for rotation and not yet unlocked it – the fair scheduler will at some time allow the writer to finish the rotate at which point any blocked reader will resume. It would be possible for readers to continuously block only if the number of writes on the data structure was unbounded.

5.1.2 TLA+ models and implementations

We implemented the MRSW CAVL tree in Java and model checked it using our methodology.

TLA+ models

We created TLA+ models for the MRSW CAVL variant that uses step-by-step rollback as well as a variant that uses the same Chunk design, but does not support concurrent reads (this is essentially a different implementation of the sequential CAVL). Listing 5 shows the Pluscal procedure that represents the `chunkShiftRite` method of the MRSW CAVL tree. Much of the model code is the same between the variants, and both variants use a lot of the same code that the BAVL tree models use.

We model checked the sequential version of the tree using integer keys from 1..5 and maximum Chunk sizes in 2..4. The checking returned no errors. We also model checked the MRSW version of the tree using keys in 1..4 and maximum Chunk sizes in 2..4. The checking also returned no errors. Note that since the Chunked AVL tree is external, the size of the tree is larger than the the size of the set of keys used. The largest tree that we checked for the MRSW CAVL algorithm had 7 nodes, excluding the `RootHolder` sentinel node.

Both model checking runs had full action coverage, meaning that each TLA+ action was taken at some time during model checking. The time required to run the model checking is discussed in Section 5.3.

Implementations

We implemented both the step-by-step rollback (recursive) and the iterative variants of the MRSW CAVL tree in Java and tested them using the linearizable data structure testing framework Lincheck [48]. We tested for correctness only and did not evaluate performance.

5.1.3 Limitations of the tree, models and implementations

The tree uses standard programming language features and does not require a specialized library or tool of any kind. The tree does not support concurrent writers, but multiple writers can still use the tree if they coordinate using an external mutual exclusion system. Future work to develop an MRMW CAVL tree is discussed in Chapter 6.

The TLA+ models that we created are not as optimized as the models that we developed of the BAVL tree. Critically, we did not implement initial data structure configurations, as described in Section 4.1.2. This limits the size of the models that can be model checked tractably, especially as the chunking aspect of the tree expands the state space significantly. Despite this, the model checking runs did have full action coverage.

The Java implementations are not extensively optimized or micro-optimized, this is discussed in Chapter 6.

5.2 Bugs found in BAVL trees

Using our methodology we were able to model the BAVL tree and obtain model checking error traces that demonstrate violations of the relaxed balance property. Recalling Section 3.2.5, the relaxed balance property is the property that the tree is always balanced during any quiescent interval.

We found scenarios, consisting of a number of write operations, that result in an unbalanced tree. In fact, we found that the canonical implementation of the BAVL tree due to Bronson violated the relaxed balance property with only one process. We fixed this error, using an existing unpublished but publicly available fix due to Trevor Brown, who is an expert on concurrent trees and author of [2, 12, 13, 14].

Following this, we implemented a model of the tree with the fix of Brown, that did not exhibit an error with a single process only. Using this model, we still found new error traces for executions using two processes. We have replicated the errors in several open-source Java implementations of the BAVL tree, and identified obvious errors in the source code of several more implementations in Java and other languages. In addition to the errors with two processes for which we have error traces, we were also able to trigger a violation of the relaxed balance property using three processes and a smaller number of operations than are required to trigger errors with only two processes. We detected the error using the Lincheck [48] linearizability testing framework, but we were not able to run a large enough model checking run using three processes to obtain an error trace for this violation.

Here we describe the errors found in more detail. For one of the errors that we found an error trace for, we present an explanation of the error.

5.2.1 Relaxed balance violation with one process in the canonical BAVL implementation

The relaxed balance violation that occurs with one process only, seems likely to be due to a bug introduced by Bronson as a byproduct of an overzealous optimization attempt. In the BAVL tree, processes that rebalance the tree also try to physically unlink logically erased nodes from the tree, if they can. This makes sense as all the locks needed to unlink a node included in a rotate are held by the rotating process. Bronson includes statements which alter the rotate operation, in order be able to do this kind of unlinking, however some of the statements are not necessary, and lead to rotate operations that are required for correctness not being performed.

The fix due to Brown consists only of removing 21 lines of unnecessary code. We discovered the bug by running a property based test in Java, on one process, checking for balance in-between operations. We were able to debug the cause of the failing test and link lines of code that seemed suspicious to a fix in a Java implementation in a public code repository belonging to Trevor Brown. We had read Brown's code during the research phase of working on this thesis and had not, at the time, been able to understand the reasoning for the changes Brown had made from the canonical implementation. When debugging the failing test we recalled the changed lines and realized that Brown had implemented a fix. The changes are undocumented.

5.2.2 Relaxed balance violations with two processes in the partially fixed BAVL tree

Following the discovery of the balance violation with one process in the canonical BAVL tree, we created a model of the version due to Brown that fixes the problem. By model checking the tree with two processes we have been able to obtain error traces using TLC that illustrate issues with the design of the tree. In this subsection we describe an error trace.

Error trace with two concurrent *erase* operations

One error occurs when running two concurrent erase operations concurrently, starting from a correct tree with 8 nodes. The error appears because the processes make incorrect decisions on whether or not to rotate the tree.

Figure 5.4 consists of four frames, and illustrates major steps involved in the erroneous execution. Frame 1 of Figure 5.4 shows the correct starting state with 8 nodes.

Consider processes p_0 and p_1 executing $erase(b)$ and $erase(g)$. First of all, both processes traverse to the locations where they will perform the erase. Following that, p_0 actually erases b and traverses to c where it decides to rotate (left) in order to rebalance the tree. Frame 2 shows this state of the tree.

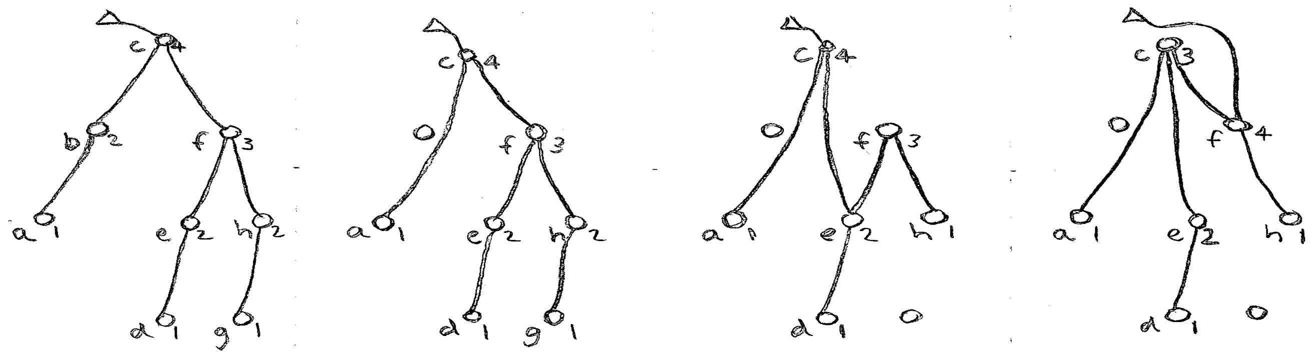

FIGURE 5.4: Four panels illustrating an erroneous execution of the BAVL tree with two concurrent *erase* operations. The sentinel root holder is denoted with a triangle. The letter at the bottom of left of each node is its key while the number at the bottom right of each node is the value of its `height` field in the data structure (not necessarily the true height).

Before p_1 has taken any more steps, p_0 partially performs its rotate, setting the right child of c to point to e. p_1 then erases g. The state at this time is shown in frame 3. Now, following its erasure of g, p_1 traverses to f and decides not to rotate the subtree rooted at f. Subsequently, p_0 completes its rotate, leading to an unbalanced tree, shown in frame 4. Both processes traverse back up the root holder and the final result is that the tree is unbalanced.

The other trace that we found reveals a similarly complex bug, with a similar root cause, that occurs with a concurrent *insert* and *erase*. That bug appears with a starting tree size of only 7 nodes, rather than 8.

The error traces are both over 100 steps long, and are rich with information. Using our trace visualizer, as described in Section 4.1.2, we were able to understand each trace and the cause of the errors quickly. It took us about 25 minutes to understand each error.

5.2.3 Relaxed balance violation with three processes in the the partially fixed BAVL tree

Using the Lincheck linearizability testing library for Java we were able to find a relaxed balance violation error using three concurrent processes executing insert operations only. The error is interesting because only 6 operations are needed in total, beginning from the empty tree, to cause the imbalance. Model checking 8 operations or fewer, with two processes rather than three, did not result in finding an error. The fewest number of operations needed to trigger an error with two processes is 9 operations, as described in Section 5.2.2.

5.2.4 Erroneous implementations

Using Lincheck we were able to replicate the relaxed balance violations, which we initially found using model checking, in three implementations of the BAVL tree. These are the canonical implementation due to Bronson, the version of Brown described in this chapter, and also the implementation used in the Synchrobench benchmark suite [33].

We were also able to identify erroneous code in all (26) forks of the canonical implementation on Github [6], as well as the C version in the ASCYLIB concurrent data structure library [22]. We did not run the implementations but rather inspected the code and found the same faulty lines that are found in the canonical implementation.

5.3 Scalability of concurrent tree model checking

In this section we discuss the scalability of the model-checking component of our methodology.

All the model checking runs run for the results in this chapter were run on a desktop computer with the following hardware specification and software configuration: OS: Ubuntu 64 bit; Architecture: x86_64; CPU: 6 cores with 12 threads, 3.8 GHz clock and 35MiB cache; Memory: 32 GiB at 3600 MHz; TLC worker threads: 8; TLC fraction allocated memory: 82% (26.2 GiB).

5.3.1 Model checking the CAVL and MRSW CAVL trees

We checked the sequential CAVL tree with integer keys in 1..5 and maximum Chunk sizes in 2..4 in 9 minutes (per chunk size parameter). We checked the MRSW version of the tree using keys in 1..4 and maximum Chunk sizes in 2..4 in 23 minutes (also per chunk size parameter).

TREE	AVAILABLE KEYS	NUMBER OF WRITES	CHUNK SIZE	MINS REQUIRED
CAVL	1,2,3,4,5	5	2	9
CAVL	1,2,3,4,5	5	3	9
CAVL	1,2,3,4,5	5	4	9
MRSW CAVL	1,2,3,4	4	2	23
MRSW CAVL	1,2,3,4	4	3	23
MRSW CAVL	1,2,3,4	4	4	23

TABLE 5.1: The number of minutes required to model check the CAVL and MRSW CAVL trees for a range of model configurations. The available keys is the set of keys that processes can use as arguments to *get*, *insert* or *erase* operations. The number of writes is the total number of write operations that are performed. The chunk size is a parameter that sets the maximum size of the chunk structures. Mins required is the total time taken to run model checking.

5.3.2 Model checking the BAVL trees

We ran models of the BAVL tree version of Brown using several combinations of initial tree configurations and model parameters. By using generated initial tree configurations as a starting point to model two concurrent operations we were able to check trees with a varying number of nodes in total.

Our run checking trees with 0-8 nodes did not find an error, and took 6 minutes. Our run using trees with 0-9 nodes did find an error, and took 20 minutes. We found another error requiring 10 nodes, beginning from configurations with at least 8 nodes, in 16 minutes.

AVAILABLE KEYS	SIZES OF INITIAL TREE	OPERATION 1	OPERATION 2	MINS REQUIRED	ERROR
1,2,3,4,5,6,7,8	0,1,2,3,4,5,6	*insert*, *get*, *erase*	*insert*, *get*, *erase*	6	No
1,2,3,4,5,6,7,8,9	0,1,2,3,4,5,6,7	*insert*, *get*, *erase*	*insert*, *get*, *erase*	20	Yes
1,2,3,4,5,6,7,8,9,10	8	*insert*, *get*, *erase*	*insert*, *get*, *erase*	16	Yes
1,2,3,4,5,6,7,8,9	7	*insert*	*erase*	1	Yes
1,2,3,4,5,6,7,8,9,10	8	*erase*	*erase*	1	Yes

TABLE 5.2: The number of minutes required to model check the BAVL tree with the fix of Brown, for a range of model configurations. The available keys is the set of keys that processes can use as arguments to operations. The size of the initial tree is the number of nodes in the tree at the start of execution, but each model checking run checks a set of initial trees. Mins required is the number of minutes required for model checking to either find an error trace or to terminate.

After we had found the errors, we ran models configured in a targeted way, with the goal of reproducing them, in 1 minute. We targeted the configurations to use 1 *insert* and 1 *erase*, and 2 *erase*s. This shows that our methodology is useful, not just to design new structures or to check existing ones believed to be correct, but to obtain an error trace for known bugs in existing structures. Error traces are invaluable for understanding the root cause of bugs.

5.3.3 Summary

Our findings on the scalability of our methodology indicate that it is strongly desirable to both eliminate recursion from data structure models (Section 4.2.1) and to use initial structure configurations (Section 4.2.4). Combining both of these techniques offers the possibility to model check much larger trees than would be possible otherwise, and in shorter time.

5.4 Limitations and lessons learned

In this section we discuss some of the limitations of our methodology, and also some of the lessons that we learned when developing this thesis, in terms of approaches that we tried but that were not fruitful.

We believe that our methodology has two central limitations. The first limitation is the limitation on the complexity and size of the models that can be tractably checked. The second limitation is the cost of our methodology in terms of model development time. A third limitation, common to all model checking techniques, is that model checking is not formal proof, and can never completely guarantee correctness.

5.4.1 Model size and complexity

The foremost problem for anyone doing exhaustive model checking is tractability. Model checking requires a lot of computational power and the scope of the models that can be checked will always be limited.

Without eliminating recursion or using initial structure configurations, we were only able to check the MRSW CAVL algorithm using trees up to 7 nodes in size. These runs did still give full action coverage. For the BAVL tree models, we model checked 2 processes only. We needed to eliminate recursion and use initial structure configurations in order to make model checking larger trees possible. We found that this was critical as two-process errors in the tree are only triggered for larger trees (8-10).

Inherent complexity

Concurrent data structures are inherently complex and any execution is very stateful, with a huge number of possible interleavings of bytecode or assembly instructions. Full exhaustive model checking at the instruction level is not tractable for data structures as complex as the BAVL or (MRSW) CAVL trees. Our methodology model checks at a granularity that is far coarser than instruction level, and the level of correctness assurance given is dependent on both the model author choosing the right abstractions, and the number of processes and the structure size that can be checked given the available computational resources. We note though that our models are far more detailed than any TLA+ models that model concrete structure implementations that we are aware of. Lamport's model of a concurrent Queue [50] discussed in Section 2.3.4, the work with a model most similar to any of ours, is 243 lines of Pluscal and TLA+, and does not check linearizability or structural invariants. Our model of the BAVL tree is 1295 lines long, and checks both.

Run time predictability

We found, for the models we created, that it was extremely difficult to predict how long model checking would take, as a function of the model and computational resources used. Every part of the model has an effect on size of the state space, as well as the time required to process a single state. Due to the size of the models and complexity of the algorithms, we were rarely able to accurately predict the resources and time required for model checking.

5.4.2 Development cost

Applying our methodology to developing models of the trees required a lot of time. Some parts of the process require thought, and will inherently take time, while other parts are mechanical and could be automated by programs in the future.

Inherently time consuming aspects of the methodology

Reasoning about abstracting a structure, and defining the invariants and properties to be checked all require time and are not likely to be completely automated. Implementing the generation of initial structure configurations could also be difficult and time consuming, depending on the data structure. The actual running of the models is also time consuming, and that is unlikely to change without further progress on model checking.

Mechanical work required

We initially spent time writing a lot of Pluscal code by hand, and found the process to be error prone and time consuming. This is due to Pluscal being quite unusual for a programming language; it has several inconvenient restrictions that are necessary to ensure it can be easily translated to TLA+. For example, there are no `break` statements. Writing the models and tweaking the models or code is all mechanical work that is not ultimately valuable to the end goal of assuring correctness.

5.4.3 Model checking is not proof

Model checking is not full formal verification: the degree of correctness assurance attainable using any model checking technique is limited by the fact that the model is only a finite-state approximation of real running system behavior. Our methodology could be used to lay a groundwork for a full formal verification approach. However, a general formal proof would require substantially more work, including developing inductive invariants relating successive valid tree states to each other, and a proof script that details the steps of the formal proof.

5.4.4 Lessons learned

Making the development cycle more productive

The most effective thing that we did to improve productivity was to use a mirror implementation of the data structure being modelled. This is explained in detail in Section 4.1.2. Using a mirror implementation means that you can take advantage of the benefits available for developers of popular, mainstream programming languages. It is also easy to run tests, using Lincheck [48] in the JVM case, for example. This is a good way to check a design idea for obvious mistakes before creating a corresponding model.

Avoid non terminating model checking

Our early models had mistakes that made model checking impossible. One model exhibited the kind of infinite looping explained in Section 4.2.1, others looped due to typos in variable names. It is difficult to tell if a long running model is looping because of a basic error, or if the model is correct but the model checking configuration used was too ambitious. This is another motivation to use initial structure configurations, as it allows you to model check structures in a targeted way, allowing you to debug particular situations.

Chapter 6

Conclusions

In this thesis we presented a methodology for modelling and model checking concurrent tree data structures with TLA+. Model checking provides a higher degree of correctness assurance than testing – we demonstrated the value and practicality of our methodology by using it both to design and verify a novel data structure, a multi-reader single-writer Chunked AVL Tree, and to discover critical bugs in a popular state of the art concurrent AVL tree algorithm [11]. In this conclusion, we summarize our major results, and highlight possible directions for future work.

6.1 Summary

6.1.1 A pragmatic methodology

We presented concrete steps that developers of concurrent trees and other concurrent data structures can apply to their projects. Our methodology covers conceptual and practical challenges involved in modelling concurrent trees and explains how they can be overcome.

6.1.2 The Multi-Reader Single-Writer Chunked AVL Tree

We applied the methodology to the development of a concurrent version of an existing data structure, the Chunked AVL Tree. We provide a version of the tree which allows an arbitrary number of reading processes to operate concurrently with a writing process. We modelled the tree in TLA+ and were able to model check a large enough tree to ensure full action coverage.

6.1.3 Finding bugs in the Practical Concurrent Binary Search Tree

In addition to applying the methodology to the development of a new structure, we also applied it to the verification of a popular existing concurrent tree from the literature. We found the tree to exhibit critical violations of its *relaxed balance* property. Moreover, we obtained error traces for two errors caused by concurrent interleavings of processes, which show the behavior of each process. The error traces contain the entire state of the system over the course of an erroneous execution. This made determining the cause of the errors easy.

6.1.4 Broader contributions of the thesis

We demonstrate that our methodology is both practical and effective. Data structures designed using our methodology can be trusted to a higher degree than testing alone would allow. TLA+ models of data structures produced using our methodology are unambiguous can be communicated with absolute clarity. Additionally, we demonstrate that by applying our methodology, it is possible to gain a deep understanding of complex bugs in concurrent structures very quickly, should they exist.

6.2 Future work

There are many directions that future works building on this thesis could go in. Here we briefly discuss a few of them.

6.2.1 CAVL Tree variant implementations

It would be useful to rigorously evaluate the performance of the MRSW CAVL trees that we presented. This would require creating optimized implementations in one or more programming languages.

6.2.2 A Multi-Reader Multi-Writer CAVL Tree

Developing a MRMW (multi-reader multi-writer) tree would allow for the tree to be used in many more applications. Adding full high-performance multi-writer concurrency, competitive with state of the art MRMW AVL trees, is a considerable challenge due to the inherent non-local nature of the Chunk structure. Our work could provide a basis for such an algorithm, as the Chunk implementation using *shift* operations, that we introduced, substantially reduces the size of the graph that needs to manipulated, per operation, compared to the original CAVL algorithm.

6.2.3 Automating model creation

Source code to TLA+ translators would be appreciated by anyone applying our methodology. A Java to Pluscal or Java to TLA+ translator would be immediately useful, and it would drastically shorten the time required to implement our methodology.

6.2.4 Proof of correctness

Our methodology could be used as a basis for a full formal verification approach: many parts of our models should be able to be reused to form the foundation of an inductive correctness proof. This would require defining inductive invariants relating successive valid tree states to each other, and a proof script detailing the steps of the proof.

Appendix A

A TLA+ and Pluscal whirlwind tour; modelling a re-entrant lock

This section serves as a short, bare-bones tour of the Pluscal language and TLA+ ecosystem. We will use a component of our models of the BAVL tree; a re-entrant lock, to motivate the discussion. A re-entrant lock is a lock that can only be held by one process, but for which holders of the lock don't block if calling $lock()$. In Java `synchronized` blocks are implemented using re-entrant locks.

A.1 The structure of a TLA+ and Pluscal model

Listings 15 and 16 contain a complete TLA+ and Pluscal model split into two parts. A TLA+ module always begins and ends with a header and footer, the header containing the module name (line 1 of Listing 15). A Pluscal algorithm is always written entirely inside a TLA+ comment (denoted by `(**)`). In Listing 15 the Pluscal algorithm is between lines 6 and 41. Line 3 can be thought of as being similar to a list of import statements in other languages. In this case line 3 declares that the module `EXTENDS` four TLA+ standard library modules. Line 4 declares a symbol Procs, which can be configured to take any TLA+ value when running the model checker. We use Procs to denote a set of process identifiers.

A Pluscal model is translated to pure TLA+ using a translator tool and the translation is written in-between two lines (lines 1 and 45 of Listing 16). The model checker, in our case TLC, reads only the lines of pure TLA+ and ignores all comments, including the one that the Pluscal algorithm is contained in. Therefore you develop the model by possibly writing some pure TLA+ above and below the Pluscal comment, but mainly by writing Pluscal inside the comment.

Recalling: TLA+ is the name given to the specification language itself. It's also used to refer to the ecosystem as a whole. TLC is a model checker which interprets TLA+ specifications. Pluscal is a pseudocode-like language that's written inside TLA+ comments and is automatically translated to TLA+.

```
 1  ------------------- MODULE reentrant_lock -------------------
 2
 3  EXTENDS Naturals, FiniteSets, TLC, Sequences
 4  CONSTANTS Procs
 5
 6  (* --algorithm algo {
 7
 8  variables
 9      locked = [e \in {0,1} |-> FALSE];
10
11  define{
12
13      AtMostOneProcInCritical ==
14          \A a,b \in Procs:
15          a # b => ~(pc[a] = "CS" /\ pc[b] = "CS")
16
17      Liveness == [] (\A a \in Procs : <> (pc[a] = "CS"))
18
19  }
20
21  \* Re-entrant
22  macro lock(addr){
23      await locked[addr] = FALSE \/ locked[addr] = self;
24      locked[addr] := self;
25  }
26
27  macro unlock(addr){
28      locked[addr] := FALSE;
29  }
30
31  fair process (proc \in Procs){
32  m0:
33      while(TRUE){
34  m1:+
35          lock(0);
36  CS:     skip;
37  m2:     unlock(0);
38      }
39  }
40
41  }*)
```

LISTING 15: Lines 1-5: the first lines of a TLA+ module. Lines 6-41: a TLA+ comment containing the Pluscal code for a re-entrant lock.

The Pluscal code itself uses a mixture of regular TLA+ syntax and its own syntax. Pluscal models are split into five sections: a `variables` block, a `defines` block, `macros`, `procedures` (not shown) and `processes`. All sections are optional, except for that there must be at least one process. The `variables` section declares global variables, a `defines` block contains *operator* definitions (pure functions), `macros` are for code inlining (like C macros), `procedures` are similar to traditional functions, except for that they don't return a value, and `processes` define the 'main' function for a process or set of processes.

A.2 Modelling a re-entrant lock

A re-entrant lock is often used to implement mutual exclusion around a critical section. We model this use case. Our model is broken into a 'main' function (Listing 15, 31-39). All processes in `Procs` use this 'main' function. The main function simply loops, repeatedly trying to take a lock, enter a critical section, and release the lock.

There are a few things to consider about our model. How do we define and assure correctness? How do we represent the address space of the program? How do we model the OS scheduler?

A.2.1 Defining and assuring correctness

Simplifying somewhat: a program that guards a critical section using a re-entrant lock wants at least one, or possibly two guarantees. The first is that no two processes are ever concurrently inside the critical section. The second guarantee that is often hoped for in practice is that all processes eventually (or *always* eventually) enter the critical section. We model these requirements using operators (Listing 15, 13-17). `AtMostOneProcInCritical` is an *invariant*, while `Liveness` is a *property*, to follow TLC terminology.

`AtMostOneProcInCritical` is a predicate over the entire model state. `pc` (Listing 15, 15) is a global Map created by Pluscal, which maps processes to their program counter. A Pluscal program is divided into atomic regions, called *actions*, defined by *labels*. The program counter is always the label of the next action to be taken by a process. `m0:` (Listing 15, 32) is an example of a label in Pluscal syntax.

`Liveness` is an example of a temporal formula. The property states: always all processes eventually execute the critical section.

We can use TLC to check invariants and properties as the model is running.

A.2.2 Representing an address space

We model object handles or memory addresses as integers. Our model has an address space which is the Set $\{0, 1\}$. Line 9 of Listing 15 defines a global Map, initially mapping each address to `FALSE` (not locked).

A.2.3 Modelling a scheduler

Operating systems typically offer configurable schedulers. For instance, a Java program may assume that each process will always eventually be scheduled. A Pluscal `process` does not have obey such behavior by default, so we must prepend the `fair` keyword to the process definition (Listing 15, 31). The fair keyword ensures that a process will eventually execute an action if it is continuously able to. In the case of a fair process, the + character (Listing 15, 34) promotes an action to give it 'strongly fair' scheduling. A strongly fair action is eventually executed if it is infinitely often possible to do so.

A.3 Translation to TLA+

Before running TLC on a Pluscal algorithm, the Pluscal comment should be translated to TLA+. The translation of our model is in-between lines 1 and 45 of Listing 16. The TLA+ translation consists of variables and operators. Many of the operators use primed variables (an ′ is appended to the identifier). These are called actions and they represent a change to some variables values. Unprimed variables hold values as evaluated in the old state and primed variables hold values as evaluated in the new state. The `SPEC` operator on line 42 of Listing 16 is the operator that is used by TLC to check the model. The `SPEC` operator follows the standard form accepted by TLC, which is a conjunction of an initializer action, a `[][Next]_vars` statement, and an optional temporal formula. `[][Next]_vars` means that there is some valid action that changes a variable, or no variables' value changes. In our case we do have an optional temporal formula, which declares weak/strong fairness of actions.

TLC will exhaustively check the state space explored by SPEC and declare that either no type error, invariant violation, or property violation has been found, or that indeed one has been found. TLC by default also provides a short sequence of states that lead to an error or violation, if one is found.

```
\* BEGIN TRANSLATION (chksum(pcal) = "e8abe34f" /\ chksum(tla) = "78f1874e")
VARIABLES locked, pc

(* define statement *)
AtMostOneProcInCritical ==
\A a,b \in Procs:
a # b => ~(pc[a] = "CS" /\ pc[b] = "CS")

Liveness == [] (\A a \in Procs : <> (pc[a] = "CS"))


vars == << locked, pc >>

ProcSet == (Procs)

Init == (* Global variables *)
        /\ locked = [e \in {0,1} |-> FALSE]
        /\ pc = [self \in ProcSet |-> "m0"]

m0(self) == /\ pc[self] = "m0"
        /\ pc' = [pc EXCEPT ![self] = "m1"]
        /\ UNCHANGED locked

m1(self) == /\ pc[self] = "m1"
        /\ locked[0] = FALSE \/ locked[0] = self
        /\ locked' = [locked EXCEPT ![0] = self]
        /\ pc' = [pc EXCEPT ![self] = "CS"]

CS(self) == /\ pc[self] = "CS"
        /\ TRUE
        /\ pc' = [pc EXCEPT ![self] = "m2"]
        /\ UNCHANGED locked

m2(self) == /\ pc[self] = "m2"
        /\ locked' = [locked EXCEPT ![0] = FALSE]
        /\ pc' = [pc EXCEPT ![self] = "m0"]

proc(self) == m0(self) \/ m1(self) \/ CS(self) \/ m2(self)

Next == (\E self \in Procs: proc(self))

Spec == /\ Init /\ [][Next]_vars
        /\ \A self \in Procs : WF_vars(proc(self)) /\ SF_vars(m1(self))

\* END TRANSLATION
=============================================================================
```

LISTING 16: The TLA+ translation of a Pluscal re-entrant lock.

Bibliography

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 23–34. ACM Press, 1995.

[2] M. Arbel-Raviv, T. Brown, and A. Morrison. Getting to the root of concurrent binary search tree performance. In H. S. Gunawi and B. Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 295–306. USENIX Association, 2018.

[3] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, Third Edition*. Addison-Wesley, 2000.

[4] S. A. Asadollah, D. Sundmark, S. Eldh, H. Hansson, and E. P. Enoiu. A study of concurrency bugs in an open source software. In K. Crowston, I. Hammouda, B. Lundell, G. Robles, J. Gamalielsson, and J. Lindman, editors, *Open Source Systems: Integrating Communities - 12th IFIP WG 2.13 International Conference, OSS 2016, Gothenburg, Sweden, May 30 - June 2, 2016, Proceedings*, volume 472 of *IFIP Advances in Information and Communication Technology*, pages 16–31. Springer, 2016.

[5] M. Barr. *Programming embedded systems in C and C++ - thinking inside the box*. O'Reilly, 1999.

[6] P. Bell and B. Beer. *Introducing GitHub - A Non-Technical Guide*. O'Reilly, 2015.

[7] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.

[8] H. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In R. Gupta and S. P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 68–78. ACM, 2008.

[9] L. Bougé, J. Vallés, X. M. Peypoch, and N. Schabanel. Height-relaxed avl rebalancing: a unified, fine-grained approach to concurrent dictionaries. 1998.

[10] S. Braithwaite, E. Buchman, I. Konnov, Z. Milosevic, I. Stoilkovska, J. Widder, and A. Zamfir. Tendermint blockchain synchronization: Formal specification and model checking. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I*, volume 12476 of *Lecture Notes in Computer Science*, pages 471–488. Springer, 2020.

[11] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In R. Govindarajan, D. A. Padua, and M. W. Hall, editors, *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 257–268. ACM, 2010.

[12] T. Brown. Techniques for constructing efficient lock-free data structures. *CoRR*, abs/1712.05406, 2017.

[13] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. *CoRR*, abs/1712.06687, 2017.

[14] T. Brown, F. Ellen, and E. Ruppert. Pragmatic primitives for non-blocking data structures. *CoRR*, abs/1712.06688, 2017.

[15] Z. Cai. Verification of concurrent data structures with tla. 2018.

[16] M. Castro and B. Liskov. Practical byzantine fault tolerance. In M. I. Seltzer and P. J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999.

[17] B. Chatterjee, N. N. Dang, and P. Tsigas. Efficient lock-free binary search trees. *CoRR*, abs/1404.3272, 2014.

[18] X. Chen. Formal verification of a concurrent binary search tree. 2013.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[20] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly binary search tree. In F. Wolf, B. Mohr, and D. an Mey, editors, *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, volume 8097 of *Lecture Notes in Computer Science*, pages 229–240. Springer, 2013.

[21] T. Crain, V. Gramoli, and M. Raynal. A fast contention-friendly binary search tree. *Parallel Process. Lett.*, 26(3):1650015:1–1650015:17, 2016.

[22] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 631–644. ACM, 2015.

[23] D. Deng, G. Jin, M. de Kruijf, A. Li, B. Liblit, S. Lu, S. Qi, J. Ren, K. Sankaralingam, L. Song, Y. Wu, M. Zhang, W. Zhang, and W. Zheng. Fixing, preventing, and recovering from concurrency bugs. *Sci. China Inf. Sci.*, 58(5):1–18, 2015.

[24] D. Detlefs, P. A. Martin, M. Moir, and G. L. S. Jr. Lock-free reference counting. *Distributed Comput.*, 15(4):255–271, 2002.

[25] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In M. L. Soffa and M. J. Irwin, editors, *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, pages 157–168. ACM, 2009.

[26] D. Drachsler, M. T. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In J. E. Moreira and J. R. Larus, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 343–356. ACM, 2014.

[27] D. Drachsler-Cohen, M. T. Vechev, and E. Yahav. Practical concurrent traversals in search trees. In A. Krall and T. R. Gross, editors, *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, pages 207–218. ACM, 2018.

[28] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In A. W. Richa and R. Guerraoui, editors, *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 131–140. ACM, 2010.

[29] Y. M. Y. Feldman, A. Khyzha, C. Enea, A. Morrison, A. Nanevski, N. Rinetzky, and S. Shoham. Proving highly-concurrent traversals correct. *Proc. ACM Program. Lang.*, 4(OOPSLA):128:1–128:29, 2020.

[30] C. C. Foster. A generalization of AVL trees. *Commun. ACM*, 16(8):513–517, 1973.

[31] E. Fynn and F. Pedone. Fast blockchain state synchronization via chunked avl trees (manuscript). 2021.

[32] A. Gonzalez. Trends in processor architecture. *CoRR*, abs/1801.05215, 2018.

[33] V. Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In A. Cohen and D. Grove, editors, *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 1–10. ACM, 2015.

[34] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with linux. *IBM Syst. J.*, 47(2):221–236, 2008.

[35] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In D. Malkhi, editor, *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings*, volume 2508 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2002.

[36] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[37] M. Herlihy and N. Shavit. On the nature of progress. In A. F. Anta, G. Lipari, and M. Roy, editors, *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, volume 7109 of *Lecture Notes in Computer Science*, pages 313–328. Springer, 2011.

[38] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[39] R. C. Holt. Some deadlock properties of computer systems. In E. J. McCluskey, N. A. Fortis, B. W. Lampson, and T. H. Bredt, editors, *Proceedings of the Third Symposium on Operating System Principles, SOSP 1971, Stanford University, Palo Alto, California, USA, October 18-20, 1971*, pages 64–71. ACM, 1971.

[40] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[41] S. V. Howley and J. Jones. A non-blocking internal binary search tree. In G. E. Blelloch and M. Herlihy, editors, *24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Pittsburgh, PA, USA, June 25-27, 2012*, pages 161–171. ACM, 2012.

[42] ISO. *ISO/IEC CD 1989.2: Information technology — Programming languages, their environments and system software interfaces — Programming language COBOL*. Third edition, 2021.

[43] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. R. Tuttle, and Y. Yu. Checking cache-coherence protocols with tla$^+$. *Formal Methods Syst. Des.*, 22(2):125–131, 2003.

[44] B. W. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[45] G. Klein, J. Andronick, M. Fernandez, I. Kuz, T. C. Murray, and G. Heiser. Formally verified software in the real world. *Commun. ACM*, 61(10):68–77, 2018.

[46] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[47] N. Koval, M. Sokolova, A. Fedorov, D. Alistarh, and D. Tsitelov. Testing concurrency on the JVM with lincheck. In R. Gupta and X. Shen, editors, *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, pages 423–424. ACM, 2020.

[48] N. Koval, M. Sokolova, A. Fedorov, D. Alistarh, and D. Tsitelov. Testing concurrency on the JVM with lincheck. In R. Gupta and X. Shen, editors, *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, pages 423–424. ACM, 2020.

[49] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[50] L. Lamport. Checking a multithreaded algorithm with $^{+}$cal. In S. Dolev, editor, *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, volume 4167 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 2006.

[51] L. Lamport. The pluscal algorithm language. In M. Leucker and C. Morgan, editors, *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer, 2009.

[52] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In T. Brecht and C. Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 162–180. ACM, 2019.

[53] B. H. Liskov and S. N. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–59, 1974.

[54] Y. Liu, J. Sun, and J. S. Dong. Developing model checkers using PAT. In A. Bouajjani and W. Chin, editors, *Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*, volume 6252 of *Lecture Notes in Computer Science*, pages 371–377. Springer, 2010.

[55] G. Lowe. Testing for linearizability. *Concurr. Comput. Pract. Exp.*, 29(4), 2017.

[56] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 378–391. ACM, 2005.

[57] B. Meyer. *Object-Oriented Software Construction, 1st edition*. Prentice-Hall, 1988.

[58] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distributed Syst.*, 15(6):491–504, 2004.

[59] F. Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*, pages 29–42. USENIX Association, 1993.

[60] C. A. Muñoz and R. A. Demasi. Advanced theorem proving techniques in PVS and applications. In B. Meyer and M. Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 96–132. Springer, 2011.

[61] M. Nofer, P. Gomber, O. Hinz, and D. Schiereck. Blockchain. *Bus. Inf. Syst. Eng.*, 59(3):183–187, 2017.

[62] B. Norris and B. Demsky. Cdschecker: checking concurrent data structures written with C/C++ atomics. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 131–150. ACM, 2013.

[63] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[64] A. Ramachandran and N. Mittal. CASTLE: fast concurrent internal binary search tree using edge-based locking. In A. Cohen and D. Grove, editors, *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 281–282. ACM, 2015.

[65] A. Ramachandran and N. Mittal. A fast lock-free internal binary search tree. In S. K. Das, D. Krishnaswamy, S. Karkar, A. Korman, M. J. Kumar, M. Portmann, and S. Sastry, editors, *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*, pages 37:1–37:10. ACM, 2015.

[66] N. Shavit and D. Touitou. Software transactional memory. In J. H. Anderson, editor, *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, pages 204–213. ACM, 1995.

[67] K. Tai. Definitions and detection of deadlock, livelock, and starvation in concurrent programs. In K. C. Tai, editor, *Proceedings of the 1994 International Conference on Parallel Processing, North Carolina State University, NC, USA, August 15-19, 1994. Volume II: Software*, pages 69–72. CRC Press, 1994.

[68] P. Thomson. *Practical systematic concurrency testing for concurrent and distributed software*. PhD thesis, Imperial College London, UK, 2016.

[69] A. Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer, 1996.

[70] M. T. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In C. S. Pasareanu, editor, *Model Checking Software, 16th International SPIN Workshop, Grenoble, France, June 26-28, 2009. Proceedings*, volume 5578 of *Lecture Notes in Computer Science*, pages 261–278. Springer, 2009.

[71] E. Verhulst, G. G. de Jong, and V. Mezhuyev. An industrial case: Pitfalls and benefits of applying formal methods to the development of a network-centric RTOS. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 411–418. Springer, 2008.

[72] P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Memory Management, International Workshop IWMM 92, St. Malo, France, September 17-19, 1992, Proceedings*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer, 1992.