

Formal Verification of a Lock-free Split-order Hashmap

Åsmund Aqissiaq Arild Kløvstad

INF-2990 Bachelor's thesis in Informatics June 17, 2020



“You don’t need to understand everything at once. You understand one thing,
then you pat yourself on the back, have a cup of coffee, and understand one
more thing.”
–Nada Amin

Contents

| | |
|--|------------|
| List of Figures | v |
| List of Tables | vii |
| Abstract | ix |
| Acknowledgements | xi |
| 1 Introduction | 1 |
| 1.1 Problem Statement | 2 |
| 1.2 Method | 2 |
| 1.3 Scope | 3 |
| 1.4 Outline | 3 |
| 2 Background | 5 |
| 2.1 Model Checking | 5 |
| 2.2 Temporal Logics | 7 |
| 2.3 TLA+ | 10 |
| 2.4 Split-Ordered List Hashmap | 12 |
| 3 Specification | 17 |
| 3.1 Goal | 18 |
| 3.2 Non-Concurrent Specification | 18 |
| 3.2.1 Hashmap for Implementation | 18 |
| 3.2.2 Non-Concurrent Specification | 20 |
| 3.3 Concurrent Specification | 22 |
| 4 Results | 27 |
| 4.1 Method | 27 |
| 4.2 Technical Details | 28 |
| 4.3 Quantitative Analysis | 29 |
| 4.3.1 Non-concurrent specification | 29 |
| 4.3.2 Concurrent Specification | 30 |

| | |
|--|-----------|
| 5 Concluding Remarks | 35 |
| 5.1 The Results of Model Checking | 35 |
| 5.2 Lessons Learned | 36 |
| 5.3 The Usefulness of Formal Specification | 39 |
| 5.4 Conclusion | 39 |
| Bibliography | 43 |

List of Figures

| | | |
|------|--|----|
| 2.1 | Pnueli's axioms | 9 |
| 2.2 | Pnueli's inference rules | 9 |
| 2.3 | The layout of the split-ordered list | 13 |
| 2.4 | Insertion without bucket splitting | 14 |
| 2.5 | Expansion and bucket splitting | 15 |
| 3.1 | The hashmap specification | 19 |
| 3.2 | The initial state of the specification | 20 |
| 3.3 | The <i>Insert</i> operation | 21 |
| 3.4 | The Next action | 21 |
| 3.5 | The <i>SOFind</i> operation | 22 |
| 3.6 | Implementation of generic hashmap | 22 |
| 3.7 | The operation structures | 23 |
| 3.8 | Starting and stepping through operations | 23 |
| 3.9 | The concurrent Next action | 24 |
| 3.10 | The steps of a <i>Delete</i> operation | 25 |
| 4.1 | Invariant 2 as a TLA predicate | 28 |
| 4.2 | The claimed invariants as TLA predicates | 29 |
| 4.3 | The failure invariants as TLA predicates | 33 |

List of Tables

| | | |
|-----|--|----|
| 4.1 | Model checking results for SplitOrder | 30 |
| 4.2 | TLC model checking results for SOConcurrent | 31 |
| 4.3 | Failed model checks | 31 |
| 4.4 | Error Codes | 32 |
| 4.5 | Model checking SOConcurrent with operation IDs | 32 |

Abstract

Concurrent systems are increasingly important in a variety of application domains, but concurrent algorithms and data structures are difficult to design, debug and verify. Formal verification with model checkers is one way to verify concurrent structures by creating a specification of the structure and demonstrating properties by exhaustively checking possible states.

One important data structure is the extensible hashmap, and Shalev et al. [1] present a novel lock-free way to implement them.

We present a formal specification of Shalev et al.'s lock-free hashmap in a concurrent setting and demonstrate that their claimed invariants hold for a subset of the state space using a model checker. We also discuss the merits of this demonstration and lessons learned from the process.

We produce 4 specifications: a non-concurrent version of the hashmap structure, an idealized hashmap used to check the non-concurrent version, and two versions of the structure used to verify invariants in a concurrent setting. The specifications are written in TLA+ and checked using the TLC model checker.

We conclude that our formal verification gives increased confidence in the correctness of Shalev et al.'s hashmap, but that the method cannot be used as conclusive proof on its own. We also present three concrete lessons about formal verification motivated by experiences during development of the specification; decide on the purpose of a specification early, test assumed invariants, and choose step size early and deliberately.

Acknowledgements

I would like to thank

my advisor Håvard D. Johansen for fruitful discussions, actionable advice and follow-up above and beyond the call of duty;

the TLA+ google group¹ for quick help with technical problems; and

my office nieghbor Isak Ø. Hellemo for frequent rubber-ducking and idea-bouncing.

1. <https://groups.google.com/forum/#!forum/tlaplus>



1

Introduction

Concurrent and distributed systems are extremely important in modern software development. Due to the end of Moore's law and the power wall making it difficult to construct ever denser and higher clock-speed CPUs the trend in hardware design since about 2005 has been to increase the number of cores and use high levels of both instruction-level parallelism and concurrency to improve performance [2]. Another example are areas of computing such as image processing and machine learning which lend themselves well to parallelization. At the same time software as a service and the massive scale industry giants like Amazon require a complex network of distributed systems to provide their functionality robustly and efficiently [3]. It is therefore increasingly important to have access to concurrent data structures.

Hash tables are an important data structure for a variety of applications because they allow for data retrieval in constant time. Several lock-based hash tables for concurrent systems exist, but the overhead of lock management and difficulty of resizing often make these impractical or inefficient [1]. Lock-free approaches use atomic operations and novel structure to avoid this overhead. This approach has proven to be useful [4] and scale better with number of concurrent processes than lock-based approaches [5]. A lock free alternative is proposed by Shalev and Shavit in [1]. We will refer to their design as the Shalev Hash Table.

Shalev et al. present four invariants of their data structure and prove them inductively to demonstrate correct behavior.

1.1 Problem Statement

Verification of correct behavior in computer systems is often desirable, but can be difficult in concurrent settings, especially in lock-free designs where locks cannot provide linearization points.

Early solutions to the problem of proving correctness include Hoare [6], Floyd [7] and Pnueli's [8] temporal logics and Lamport's Temporal Logic of Actions (TLA) [9] which seek to formalize the execution of programs in order to reason about them with logic. These formal methods proved useful, but laborious [10].

Building on the work in temporal logics, model checkers seek to minimize the human labor and ingenuity needed to prove correctness. This is done by specifying a model using some system of logic and then letting a model checker exhaustively survey the possible states of the system. This automates the process of proving correctness.

We will apply these methods to the Shalev Hash Table with the thesis that

The Shalev Hash Table can be formally verified for concurrent settings.

We intend to show that a formal specification of the data structure can be checked using a model checker to increase our confidence in the claimed invariants.

Additionally we will use this example to explore and discuss the use of model checkers for formal verification of concurrent data structures.

1.2 Method

To demonstrate the correctness of the Shalev Hash Table, we specify its behavior using TLA+ [11] and use the TLC model checker in the TLA+ toolbox [12] to test the claimed invariants.

We construct four specifications:

1. a generic hashmap,
2. a non-concurrent specification of the split-order structure,
3. a concurrent specification of the split-order structure, and

4. a concurrent specification with operation ID's.

The first two are used to test the behavior of the structure in a non-concurrent setting which allows for a larger state space. The third is used to test a smaller state space in a concurrent setting, while the final specification is needed to test invariants relating to early termination of operations.

1.3 Scope

The validity of formal verification with a model checker relies on two premises:

1. that the specification of the structure is correct
2. that the behavior of the model checker is correct.

The focus of this thesis is premise 1. For a specification to be helpful in proving correctness of a system it is important that it specifies the system accurately, and that the properties checked by the model checker are precisely the properties we wish to show in the system. We explain the specification in Chapter 3 and the properties in Chapter 4. Both are also discussed in Chapter 5

Premise 2 is outside the scope of this project, but the TLC model checker is open source [12] and has been used in industry with success [3, 13].

This thesis is only concerned with formal verification of the Shalev Hash Table, not its implementation, performance or other proofs of correctness.

1.4 Outline

Chapter 2 discusses the motivation for formal verification and model checking through a review of related literature, followed by a description of Temporal Logic in Section 2.2 and the TLA+ language in Section 2.3. Finally Shalev et al.'s hashmap design is described in Section 2.4.

Chapter 3 describes the specification of the hashmap in TLA+ and the development of this specification.

Chapter 4 describes the method and results of model checking.

Chapter 5 discusses these results in relation to the problem statement, and

explores experiences and lessons learned.

/2

Background

This section presents relevant prior work and background on model checking, TLA+ and the Shalev Hash Table. Section 2.1 briefly describes the development and use of formal verification and model checkers, Section 2.2 recounts Pnueli's [8] early work on temporal logic, and Section 2.3 covers the syntax and structure of TLA+. Finally Section 2.4 describes Shalev et al.'s data structure.

2.1 Model Checking

Model Checking: Algorithmic Verification and Debugging [10] In the Turing Lecture by the winners of the 2007 Turing Award, Edmund Clarke, Allen Emerson and Joseph Sifakis they describe the development and use of model checkers as a verification method for computer systems. Previous efforts to prove correctness had been focused on formal proofs which have three key shortcomings:

1. they require human ingenuity,
2. they are difficult to work with in concurrent and distributed systems,
3. they scale poorly with system size and complexity.

Instead, they propose algorithmic model checkers.

With this method a Temporal Logic is used to specify the correct behavior of a system and the model checker verifies that this behavior is not violated by exploring the state space of the model. Importantly, such model checkers produce a counter example – an example of incorrect behavior – which makes debugging and correcting the system easier. Key properties of a temporal logic are *expressiveness* and *efficiency*.

Model checking also scales poorly with system complexity, so several techniques are introduced to deal with "state space explosion"

- symbolic checking of ordered binary decision diagrams
- isolation of independent events in concurrent systems
- bounded checking by solving SAT
- reduce state space by increasing level of abstraction
 - if counterexamples are found a lower abstraction level is needed, but "good" properties hold through abstraction mappings

How Amazon Web Services Uses Formal Methods [3] Amazon's AWS services are all underpinned by large and complex distributed systems. This is necessary for high availability, growth and cost-effective infrastructure. Traditionally these systems have been tested by savvy engineers who know what to test and look for. However, some errors are very rare and will very likely slip through such testing. To catch these errors they employ model checking (with TLA+).

The PlusCal or TLA+ specifications work as a tool to bridge the gap between design and implementation. Designs are expressive, but imprecise while the implementation is precise, but hides overall structure. Through a choice of abstraction level, specifications can bridge this gap and provide both. An expressive specification also provides useful documentation of the system.

The key benefits of model checkers at Amazon are:

- a precisely specified design helps make changes and optimizations safely. This usage improves system understanding.
- they are faster than formal proofs

- a correct design and the understanding the specs provide promote better, more correct code.

In this section we have looked at the development of algorithmic model checkers, and their use in industry. We have seen that algorithmic model checker solve many of the issues with hand-written formal proofs, but still suffer from state space explosion. However, they have proven useful in the development process of distributed systems at Amazon Web Services by precisely describing systems and promoting understanding.

2.2 Temporal Logics

The Temporal Logic of Programs [8] In The Temporal Logic of Programs [8], Amir Pnueli proposes a unified approach to the verification of both sequential and concurrent programs. His work seeks to unify approaches to both, while also presenting a system that emulates the design intuition of programmers. The key concepts in this work are *invariance* – which covers partial correctness, clean behavior, mutual exclusion and deadlock freedom – and *eventuality* – which generalizes these notions to cyclic programs and provides a special case of total correctness.

A dynamic discrete system is generalized as a three-tuple $\langle S, R, s_0 \rangle$ where S is the set of possible states, R a transition relation, and s_0 the initial state of the system. In order to make later constructions easier we further specify

$$s = \langle \pi, u \rangle$$

where π is the control component specifying the location in the program and u is the data component describing the state of any variables and data structures, and

$$R(\pi, u) = N(\pi, u) \wedge T(\pi, u)$$

where N describes the control flow and T the change in data such that a step in the execution may be described by

$$R(\langle \pi, u \rangle, \langle \pi', u' \rangle) \iff \pi' = N(\pi, u) \wedge u' = T(\pi, u)$$

To reason about concurrent programs we let states have multiple control components $s = \langle \pi_1, \pi_2, \dots, \pi_n, u \rangle$ and randomly choose one control component to update in each step. Finally we let X be the set of all reachable states for the system. A predicate $p(s)$ is **invariant** if $p(s)$ is true $\forall s \in X$.

We can now start to define useful properties of the systems described in this way.

Partial correctness is the claim that given the correct input, a program produces the correct output. We let $\phi(x)$ be the statement "reaching the end state \implies (correct input \implies correct output)". Partial correctness is equivalent to saying ϕ is an invariant.

Clean execution means the program does not behave illegally, i.e it does not access illegal memory locations or divide by zero. We may define these restrictions as a predicate to make clean execution equivalent to this predicate being invariant.

Mutual exclusion. Given a critical section C , mutual exclusion of the processes π_1 and π_2 is described by the invariance of the predicate $\neg(\pi_1 \in C \wedge \pi_2 \in C)$.

In addition to these properties we wish to reason about *temporal* implications. We let time be described by a $t \in \mathbb{N}$ and $H(p, t)$ denote the value of the predicate p at time t . We then introduce the temporal operator $p \rightsquigarrow q$ to mean p eventually leads to q , or formally:

$$p \rightsquigarrow q : \forall t_1 \exists t_2 \text{ s.t } t_1 \leq t_2, H(p, t_1) \implies H(q, t_2)$$

For all times t_1 there is a later time t_2 such that if p holds at t_1 , q will hold at t_2 . Armed with eventuality we can define temporally useful properties of systems.

Total correctness is stronger than partial correctness because it also requires that the program reaches an end state. We can express total correctness as $\langle \pi = l_0, u = \phi \rangle \rightsquigarrow \langle \pi = l_m, u = \psi \rangle$ where ϕ denotes correct input, and ψ denotes correct output and l_0, l_m are the start and end labels of the system, respectively.

Accessibility is the guarantee that some segment S of a program can be reached. It can be expressed by $\pi = l_0 \rightsquigarrow \pi \in S$

Responsiveness. It is often desirable that some request r will be met by a response s . We call this responsiveness and describe it by $r \rightsquigarrow s$.

With these definitions under our belt, Pnueli defines the necessary axioms and inference rules to reason about the correctness of programs.

These axioms define two ways to establish eventuality. A2 says that any logical implication is also an eventuality. A1 is a little more involved, but states that if for all consecutive states p being true in the first implies q being true in the second, then p eventually leads to q .

$$[\forall s, s' p(s) \wedge R(s, s') \implies q(s')] \Rightarrow p \rightsquigarrow q \quad (\text{A1})$$

$$(p \implies q) \Rightarrow p \rightsquigarrow q \quad (\text{A2})$$

Figure 2.1: Pnueli's axioms

$$p \rightsquigarrow q, \forall s, s' r(s) \wedge R(s, s') \implies r(s') \Rightarrow (p \wedge r) \rightsquigarrow (q \wedge r) \quad (\text{R1})$$

$$p \rightsquigarrow q, q \rightsquigarrow r \Rightarrow p \rightsquigarrow r \quad (\text{R2})$$

$$p_1 \rightsquigarrow q, p_2 \rightsquigarrow q \Rightarrow (p_1 \vee p_2) \rightsquigarrow q \quad (\text{R3})$$

$$p \rightsquigarrow q \Rightarrow (\exists u p) \rightsquigarrow q \quad (\text{R4})$$

Figure 2.2: Pnueli's inference rules

Using these axioms and inference rules as well as first-order logic, Pnueli goes on to formalize invariance and eventuality for sequential programs and concurrent programs.

Invariance of the predicate $q(\pi, u)$ is described by the conjunction $\bigwedge_i \pi = l_i \implies q(l_i, u)$, asserting that q is true at all points of execution. This method is called an *attachment* of the predicate to the program. In a concurrent program we generalize q to hold when any π_i is updated by N and construct either a full attachment

$$\bigwedge_{i_1, i_2, \dots, i_n} (\pi_1 = i_1 \wedge \pi_2 = i_2 \wedge \dots \wedge \pi_n = i_n) \implies q(\pi_1, \dots, \pi_n, u)$$

shown here for n concurrent execution threads, or the partial attachment

$$\bigwedge_i \pi_1 = i \implies q(\pi_1 = i, \pi_2, u) \wedge \bigwedge_j \pi_2 = j \implies q(\pi_1, \pi_2 = j, u)$$

shown here for two threads π_1 and π_2 .

Eventuality is formulated as the temporal implication $\pi = l_1 \wedge p(u) \rightsquigarrow \pi = l_2 \wedge q(u)$. We can then describe the path between l_1 and l_2 by a finite sequence of steps and apply A1 to each step.

Finally, Pnueli introduces two new "tense operators" Future and Global on predicates such that at some time n

$$F(p) = \exists t \geq n \text{ s.t } H(t, p)$$

$$G(p) = \forall t \geq n H(t, p)$$

This lets us describe useful properties such as

$p \implies F(q)$ – if p is true now, then at some point in the future q will be true.

$G(p \implies F(q))$ – whenever p is true it will eventually be followed by a state in which q is true. (this is equivalent to $p \rightsquigarrow q$)

Pnueli gives us a powerful basic framework to describe and reason about both sequential and concurrent program execution which has informed many newer temporal logics, including TLA and TLA+.

2.3 TLA+

This section introduces the structure and syntax of TLA+ necessary to follow the description of specifications in the following chapter.

TLA+ is a formal specification language that describes a series of **actions** acting on **variables**. Each action statement is a logical predicate on a pair of states called a **step** and a sequence of steps – called a **behavior** – adheres to a specification if this predicate holds for every step of the behavior.

The basic form of a specification is

$$\text{Spec} \triangleq \text{Init} \wedge \square[\text{Next}]_{\langle \text{iterator}, \text{reverseIterator} \rangle}$$

which reads "Init and always Next" and means the statement *Init* is true for the initial state and the values of the variables *iterator* and *reverseIterator* make the statement *Next* true on all steps.

Each action is a predicate on primed and un-primed variables, where primed variables denote the value of a variable in the second state. For example

$$\text{Next} \triangleq \text{iterator}' = \text{iterator} + 1 \wedge \text{UNCHANGED reverseIterator}$$

describes a variable *iterator* which is increased by one in each step. This is not an assignment, but a logical predicate whose value can be true or false on each step. $=$ is a statement of mathematical equality, so the order of variables does not matter. Also note that each action must include all variables, so **UNCHANGED** is used to specify that the value of *reverseIterator* does not

change in this step.

When an action includes multiple variables, their behavior can be specified by a conjunction or disjunction of predicates. For example

$$\text{Next} \triangleq$$

$$\wedge \text{iterator}' = \text{iterator} + 1$$

$$\wedge \text{reverseIterator}' = \text{reverseIterator} - 1$$

is a conjunction describing a step in which *iterator* is increased by one and *reverseIterator* is decreased by one, while

$$\text{Next} \triangleq$$

$$\vee \text{iterator}' = \text{iterator} + 1$$

$$\vee \text{reverseIterator}' = \text{reverseIterator} - 1$$

is a disjunction in which either *iterator* is incremented or *reverseIterator* is decremented.

The values of variables can be simple constants like numbers and strings, sets, or functions from one set to another. Functions are used to represent associative arrays and more complicated structures.

$$\text{list} \triangleq [0 \dots 42 \mapsto \text{Nat}]$$

Specifies a list with indexes from 0 to 42 (inclusive) and values in the natural numbers and

$$\text{student} \triangleq [\text{ID} : 0 \dots 1000, \text{major} : \{\text{"CS"}, \text{"math"}, \text{"gardening"}\}]$$

describes a structure with two fields: *ID* and *major* and the possible values of these fields. The values of functions are updated with the EXCEPT construct where

$$\text{list}' = [\text{list EXCEPT } ![6] = 28]$$

means that the new value list is the same as the old, except the value at index 6 is now 28. This notation is included because many readers find it strange or confusing.

Additionally TLA+ includes the temporal operators \square and \diamond meaning always and eventually, which correspond to Pnueli's Global and Future operators as well as $\diamond\square$ which means the predicate eventually becomes true and then stays true forever.

Finally, implementation in TLA+ is implication. If specification A implements specification B this is exactly equivalent to $A \implies B$. If A is true on a behavior, then B is also true on that behavior.

2.4 Split-Ordered List Hashmap

This section provides a description of Shalev et al.'s split-order hashmap structure [1] – the first lock-free extensible hash table implemented using only loads, stores and atomic Compare and Swap (CAS).

Hashmaps are a key building block in many important systems [citation needed], but are difficult to implement concurrently. In particular, the resizing (extending) of the table is difficult to do atomically because at the very least a node must be moved from one list to another. In order to avoid conflicts and loss of data in this process some overhead is required which impacts performance.

The key insight of Shalev et al. is to flip the process upside down. Instead of moving nodes between buckets, they suggest moving the buckets among a statically ordered list of nodes. This requires an ordering of the list in which a bucket can always be split into two new buckets while their contents remain correct. A node should always reside in the bucket corresponding to its key $\text{mod } 2^i$ where 2^i is the current size of the table.

Split-Ordered Lists are introduced to make resizing of the map possible without introducing locks. By sorting the keys according to their reversed binary representation Shalev et al. obtain a list which can be always be split into buckets $\text{mod } 2^i$. This is because such an ordering corresponds to difference in the keys' i th least significant bit, which is equivalent to having a different remainder $\text{mod } 2^i$.

To deal with the problems caused by removing nodes pointed to by hash table entries dummy nodes with the bucket value are introduced. These nodes signify the start of a bucket and are recursively initialized when an item is inserted into an uninitialized bucket. To distinguish dummy nodes from regular nodes in the list, regular node keys have their most significant bit set to 1 before being reversed. This order and the structure of the map can be seen in Figure 2.3.

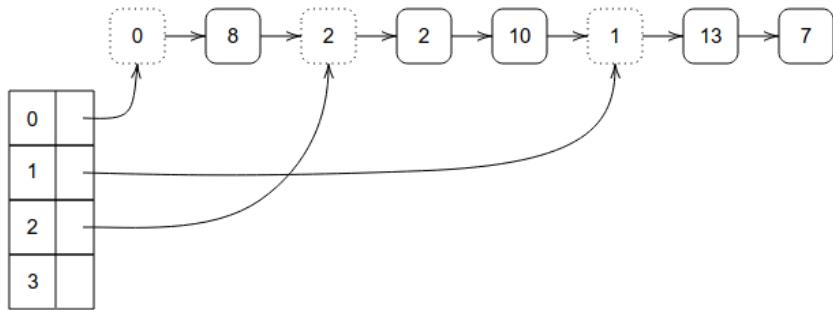


Figure 2.3: The layout of the split-ordered list

Insertion in to the map is done through atomic CAS instructions on the list. If a bucket is not initialized, a dummy node is created and inserted into the list before the new value is added as shown in Figure 2.4. The map is expanded by doubling the number of buckets and inserting new dummy nodes. Because of the split-ordering of the list, it is always possible to insert a new bucket by splitting an existing one. This process is shown in Figure 2.5.

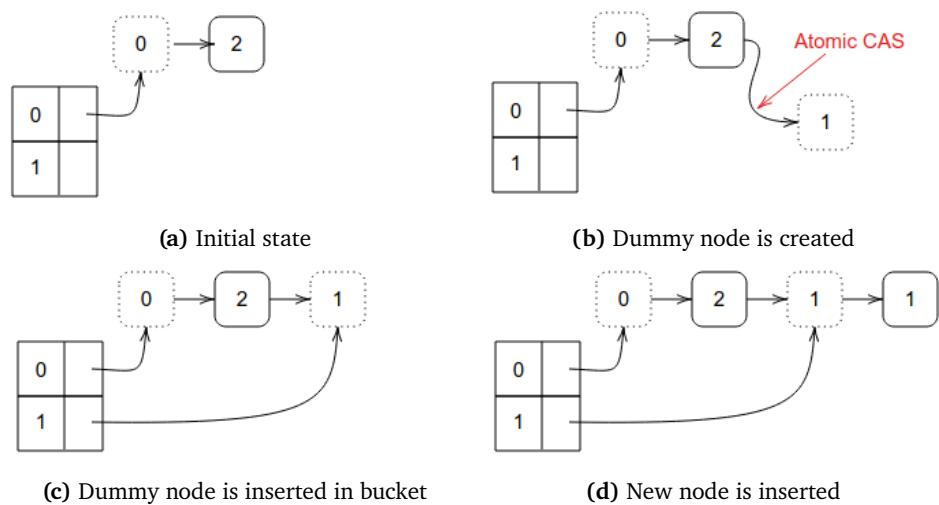
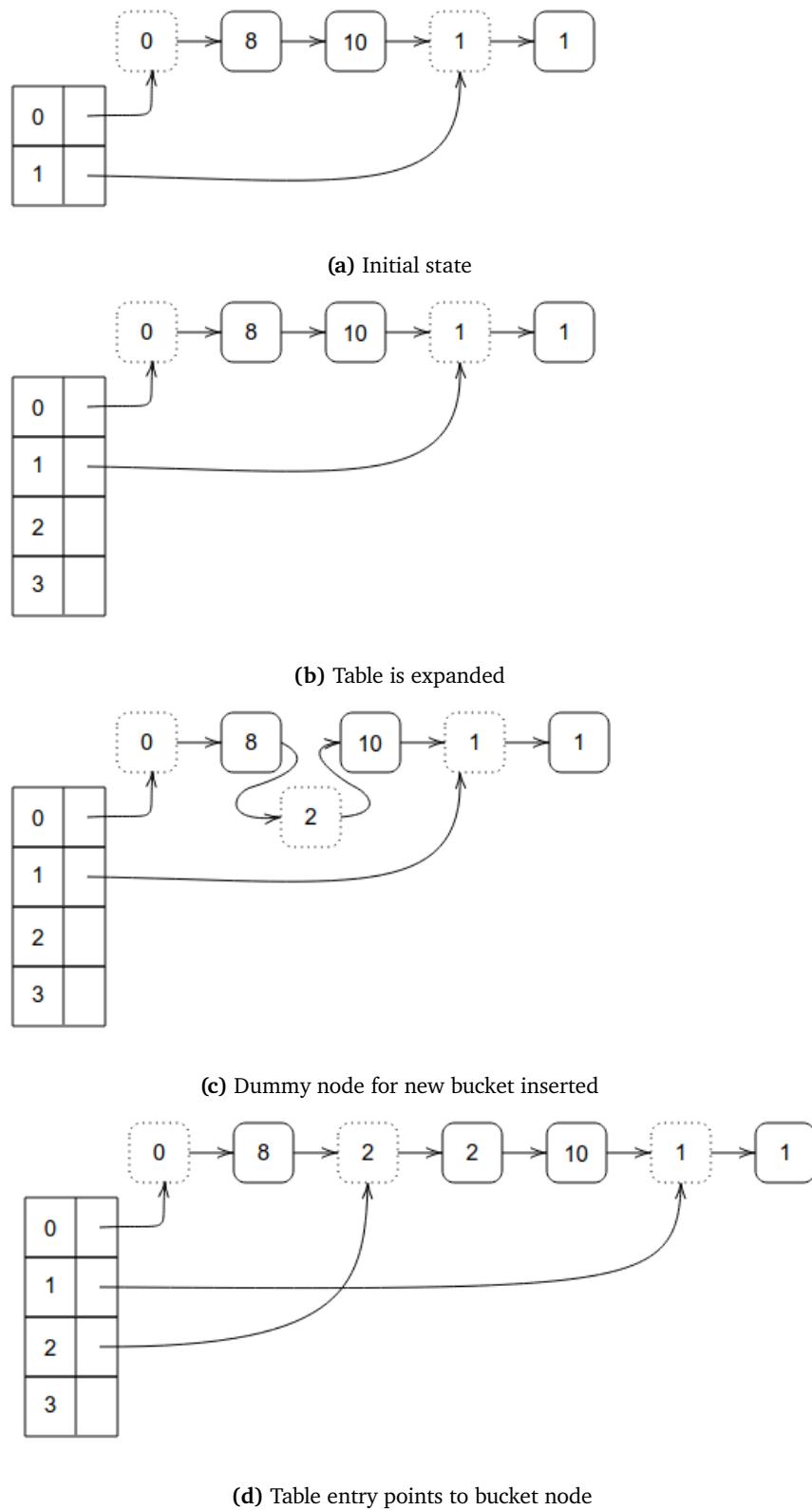


Figure 2.4: Insertion without bucket splitting

**Figure 2.5:** Expansion and bucket splitting

/3

Specification

This chapter describes the structure of the TLA+ specification and the process of writing the specification. First we introduce the goals and scope of the specification, then describe the definitions of necessary data structures and operations.

We construct two specifications: a non-concurrent and a concurrent version.

The non-concurrent specification consists of two main parts:

1. a generic hashmap and
2. *SplitOrder* implementing this map.

The generic specification describes the workings of a hashmap with insert and remove operations and ensures that this structure behaves as specified. The *SplitOrder* specification describes Shalev et al.'s specific structure and algorithms for implementing hashmap functionality.

The concurrent specification *SOConcurrent* describes the Shalev Hash Table in a concurrent setting. This is accomplished by splitting operations into steps and letting the steps proceed independently of each other.

3.1 Goal

The purpose of our specification is to verify the claims about correctness made by Shalev et al. Because TLA+ is designed for checking liveness and safety properties [14] we will not check claims about the performance of the implementation. This leaves 4 invariants:

1. the list beginning at bucket o is always sorted
2. if a bucket is initialized, then it points to a dummy node which is in the list beginning at bucket o
3. if a key k is in the map, then `insert(k)` fails. Otherwise k is added to the map
4. if a key k is in the map, then `remove(k)` removes it from the map. Otherwise it fails.

The non-concurrent specification demonstrates correctness by implementing a generic hashmap specification which describes correct behavior, while the concurrent specification looks directly at the above invariants. Additionally we will check *type safety*: at every point of execution, all keys and values are members of predefined key- and value-sets, respectively.

3.2 Non-Concurrent Specification

3.2.1 Hashmap for Implementation

The generic hashmap specification seen in Figure 3.1 describes how a hashmap *should* behave. It maintains a set of keys $keys$ and a mapping map from keys to values. Additionally it describes two operations: insert and remove.

Insert adds a key to the set of keys and changes one entry in the map. The key is added with a set union, which preserves key uniqueness. The insert action is always enabled since there will always be keys and values in PossibleKeys and PossibleValues.

Remove removes one key from the set and changes its mapping to NULL. This is done through the set difference operator. This action is idempotent, so attempting to remove a key that is not in the map will result in no change.

The specification also contains temporal formulae *TypeOK* and *KeyHasValue*

MODULE *hashmap*

This module describes a hashmap to be used for testing with Shalev et al.'s split-ordered list implementation of the data structure

EXTENDS *Integers*

CONSTANTS *NULL, PossibleKeys, PossibleValues*

VARIABLES *keys, map*

Initial state has empty map and no keys

$$\begin{aligned} \text{HashmapInit} &\stackrel{\Delta}{=} \wedge \text{keys} = \{\} \\ &\wedge \text{map} = [k \in \text{PossibleKeys} \mapsto \text{NULL}] \end{aligned}$$

Insert changes exactly one mapping of the hashmap and adds one key to the set of keys

$$\begin{aligned} \text{Insert} &\stackrel{\Delta}{=} \exists k \in \text{PossibleKeys} : \\ &\exists v \in \text{PossibleValues} : \\ &\wedge \text{keys}' = \text{keys} \cup \{k\} \\ &\wedge \text{map}' = [\text{map EXCEPT } ![k] = v] \end{aligned}$$

Remove sets exactly one mapping to NULL

$$\begin{aligned} \text{Remove} &\stackrel{\Delta}{=} \exists k \in \text{PossibleKeys} : \\ &\wedge \text{keys}' = \text{keys} \setminus \{k\} \\ &\wedge \text{map}' = [\text{map EXCEPT } ![k] = \text{NULL}] \end{aligned}$$

Next is either an insert, a remove or a find $\text{Find}(k)$ returns NULL if not in map, otherwise a value

$$\begin{aligned} \text{HashmapNext} &\stackrel{\Delta}{=} \vee \text{Insert} \\ &\vee \text{Remove} \end{aligned}$$

TypeOK asserts all keys and values are of the right type

$$\begin{aligned} \text{TypeOK} &\stackrel{\Delta}{=} \forall k \in \text{keys} : \\ &\wedge k \in \text{PossibleKeys} \\ &\wedge \text{map}[k] \in \text{PossibleValues} \end{aligned}$$

KeyHasValue asserts that every key is mapped to a value

$$\text{KeyHasValue} \stackrel{\Delta}{=} \forall k \in \text{keys} : \neg(\text{map}[k] = \text{NULL})$$

The hashmap specification as a temporal formula

$$\text{HashmapSpec} \stackrel{\Delta}{=} \text{HashmapInit} \wedge \square[\text{HashmapNext}]_{\langle \text{keys}, \text{map} \rangle}$$

Figure 3.1: The hashmap specification

```

EXTENDS Integers

CONSTANTS NULL, PossibleKeys, PossibleValues, LoadFactor, MaxSize

VARIABLES keys, AuxKeys, list, buckets, size, count, map

ASSUME
   $\wedge \text{PossibleKeys} \subseteq 0..15$ 
   $\wedge \text{NULL} \notin \text{PossibleKeys}$ 
   $\wedge \text{NULL} \notin \text{PossibleValues}$ 

```

The Init for split-order keys is initially empty the map maps every possible key to NULL The list initially contains only the o dummy node

```

SOInit  $\triangleq$   $\wedge \text{keys} = \{\}$ 
   $\wedge \text{AuxKeys} = \{\}$ 
   $\wedge \text{list} = [n \in 0..255 \mapsto \text{IF } n = 0 \text{ THEN } \text{SODummyKey}(0) \text{ ELSE } \text{NULL}]$ 
   $\wedge \text{buckets} = [m \in \text{PossibleKeys} \mapsto \text{IF } m = 0 \text{ THEN } \text{SODummyKey}(0) \text{ ELSE } \text{NULL}]$ 
   $\wedge \text{size} = 1$ 
   $\wedge \text{count} = 0$ 
   $\wedge \text{map} = [k \in \text{PossibleKeys} \mapsto \text{NULL}]$ 

```

Figure 3.2: The initial state of the specification

that specify correct behavior. These have been checked with the TLC model checker and holds for up to 16 possible values and 4 possible keys.

Finally the entire hashmap is described by *HashmapSpec*. This temporal formula states that the hashmap specification is fulfilled if the initial state fulfills *HashmapInit* and each action fulfills *HashmapNext*. This formula is used to prove a more complex specification implements the semantics of Hashmap.

3.2.2 Non-Concurrent Specification

The specification consists of three main parts:

1. the data structures,
2. operations on the map, and
3. the abstract specification tying them together.

This section describes the construction and function of each in turn.

The necessary data structures are a list of nodes and an array of buckets

$$\begin{aligned}
 SOInsert &\stackrel{\Delta}{=} \wedge \exists k \in PossibleKeys : \\
 &\exists v \in PossibleValues : \\
 &\quad BucketInsert(k, v) \\
 BucketInsert(k, v) &\stackrel{\Delta}{=} \\
 &\text{Either a bucket needs to be initialized} \\
 &\quad \vee \wedge buckets[k \% size] = NULL \\
 &\quad \wedge BucketInit(k \% size) \\
 &\quad \wedge ListInsert(SORegularKey(k), v) \\
 &\quad \wedge AuxKeys' = AuxKeys \cup \{k\} \\
 &\quad \text{Or the bucket is already initialized} \\
 &\quad \vee \wedge buckets[k \% size] \neq NULL \\
 &\quad \wedge ListInsert(SORegularKey(k), v) \\
 &\quad \wedge AuxKeys' = AuxKeys \cup \{k\} \\
 &\quad \wedge \text{UNCHANGED } \langle buckets \rangle \\
 ListInsert(k, v) &\stackrel{\Delta}{=} \text{IF } list[k] = NULL \\
 &\quad \text{THEN } list' = [list \text{ EXCEPT } ![k] = v] \wedge count' = count + 1 \\
 &\quad \text{ELSE } \text{UNCHANGED } \langle list, count \rangle
 \end{aligned}$$
Figure 3.3: The *Insert* operation
$$\begin{aligned}
 SONext &\stackrel{\Delta}{=} \vee SOInsert \wedge BucketGrow \wedge \text{UNCHANGED } \langle map, keys \rangle \\
 &\vee SORemove \wedge \text{UNCHANGED } \langle size, map, keys \rangle \\
 &\vee SOFind \wedge \text{UNCHANGED } \langle size, count, list \rangle
 \end{aligned}$$
Figure 3.4: The Next action

pointing to nodes. These are both specified as TLA+ functions. The list is a function from the set of list-keys to the set of values, while bucekts is a function from keys to list-keys. Additionally the specification includes a set of keys and a map from keys to values that correspond to the keys and map in the generic specification. The initial state of the map, showing these structures and their initial values is shown in Figure 3.2

The *Insert* and *Remove* operations are written in three layers. Figure 3.3 shows the Insert operation. First, *SOInsert* describes a very high-level view: there exists some key and some value, and we insert the value with that key. *BucketInsert* is the most involved, corresponding to the insert operation described by Shalev et al.'s pseudocode [1]. Here the correct bucket is found or initialized and the value is inserted into the list pointed to by this bucket with the split-order key. Finally, *ListInsert* inserts the value into the list, or ignores it if the node already has a value.

Finally, the *Next* action shown in Figure 3.4 describes the transition of the

$$\begin{aligned}
 SOFind &\triangleq \exists k \in AuxKeys : \\
 &\wedge keys' = keys \cup \{k\} \\
 &\wedge AuxKeys' = AuxKeys \setminus \{k\} \\
 &\wedge \text{IF } buckets[k\%size] = \text{NULL} \\
 &\quad \text{THEN } \wedge \text{BucketInit}(k\%size) \\
 &\quad \wedge map' = [map \text{ EXCEPT } ![k] = \text{ListFind}(k\%size, SORRegularKey(k))] \\
 &\quad \text{ELSE } \wedge \text{UNCHANGED } buckets \\
 &\quad \wedge map' = [map \text{ EXCEPT } ![k] = \text{ListFind}(k\%size, SORRegularKey(k))]
 \end{aligned}$$

Figure 3.5: The *SOFind* operation

$$\begin{aligned}
 SOSpec &\triangleq SOInit \wedge \square[SONext]_{\langle keys, AuxKeys, list, buckets, size, count, map \rangle} \\
 &\text{INSTANCE } hashmap \\
 &\text{THEOREM } SOSpec \implies HashmapSpec
 \end{aligned}$$

Figure 3.6: Implementation of generic hashmap

system from one state to another. It states that such a transition consists of either inserting and growing the map, removing, or the *SOFind* operation.

The *SOFind* operation shown in Figure 3.5 corresponds to the find operation in Shalev et al.'s pseudocode and is responsible for updating the map. Because we wish to show that this specification implements the generic hashmap it is necessary to update the set of keys and the map itself at the same time and so a set of auxiliary keys is maintained and only added to the "real" map once a find operation looks for them. Using the "working variables" trick, the specification can be shown to implement the generic hashmap through implication as seen in Figure 3.6.

The theorem $SOSpec \Rightarrow HashmapSpec$ seen in Figure 3.6 states that any program following the split-order specification also follows the hashmap specification. Hence, any program following the split-order specification is a correct implementation of a hashmap. It is tested by checking the property *Hashmap-Spec* in a model checking *SOSpec*.

3.3 Concurrent Specification

The specification described in Subsection 3.2.2 describes the working of the split-order structure. However, it is not useful for proving the correctness of the structure in a concurrent setting. This is because its structure presupposes the linearizability of its operations by making them action statements. This

$OperationStates \triangleq$

Set of all possible active operations

They can be of type insert, delete or bucket_init

Step denotes the current step of the operation

$[type : \{"insert"\}, step : 1 \dots 4, key : PossibleKeys, value : PossibleValues]$

\cup

$[type : \{"delete"\}, step : 1 \dots 3, key : PossibleKeys]$

\cup

$[type : \{"bucket_init"\}, step : 1 \dots 3, bucket : 0 \dots (MaxSize - 1)]$

Figure 3.7: The operation structures

Begin an insert operation

$Insert(k, v) \triangleq activeOps' = activeOps \oplus SetToBag(\{[type \mapsto "insert", step \mapsto 1, key \mapsto k, value \mapsto v]\})$

Begin a delete operation

$Delete(k) \triangleq activeOps' = activeOps \oplus SetToBag(\{[type \mapsto "delete", step \mapsto 1, key \mapsto k]\})$

Begin a bucket_init operation

$BucketInit(b) \triangleq activeOps' = activeOps \oplus SetToBag(\{[type \mapsto "bucket_init", step \mapsto 1, bucket \mapsto b]\})$

$NextStep(op) \triangleq activeOps' = (activeOps \ominus SetToBag(\{op\})) \oplus SetToBag(\{[op \text{ EXCEPT } !["step \mapsto 1"]]\})$

$End(op) \triangleq activeOps' = activeOps \ominus SetToBag(\{op\})$

Figure 3.8: Starting and stepping through operations

means each state transition consists of a complete operation, rather than a single instruction.

To specify concurrent working, a more granular specification is needed. In a complete description each step would be a single machine instruction, but this is both infeasible and counterproductive as such a specification is no more useful for verification than the program itself.

Instead, we identify the steps in the each operation that change the shared state of the system and make those the steps of our specification. This results in each operation (*Insert*, *Remove*, and *Bucket Init*) being split into several steps. We then maintain a multiset (called a "bag") of active operations of the form shown in Figure 3.7 and step through them as shown in Figure 3.8. The result is the next-step action shown in Figure 3.9. This disjunction specifies that each next step can either start an insert or remove operation, or perform a single step in an active operation. To show the structure of a single operation in the concurrent specification, we will look at *Delete* (Figure 3.10). *Delete1* checks if the relevant bucket is initialized and starts a *bucket_init* operation if it is not. Because starting a new operation and advancing the current operation both modify the state of *activeOps*, a union of both changes is needed. Also note that this step is enabled iff there exists an operation with type "delete" and step 1

$$\begin{aligned}
 SOnext &\triangleq \\
 &\vee \wedge \exists k \in PossibleKeys : \\
 &\quad \exists v \in PossibleValues : \\
 &\quad \quad Insert(k, v) \\
 &\quad \wedge BagCardinality(activeOps) < MaxActiveOps \\
 &\quad \wedge \text{UNCHANGED } \langle buckets, count, list, size \rangle \\
 &\vee \wedge \exists k \in PossibleKeys : \\
 &\quad Delete(k) \\
 &\quad \wedge BagCardinality(activeOps) < MaxActiveOps \\
 &\quad \wedge \text{UNCHANGED } \langle buckets, count, list, size \rangle \\
 &\vee Insert1 \\
 &\vee Insert2 \\
 &\vee Insert3 \\
 &\vee Insert4 \\
 &\vee Delete1 \\
 &\vee Delete2 \\
 &\vee Delete3 \\
 &\vee BucketInit1 \\
 &\vee BucketInit2 \\
 &\vee BucketInit3
 \end{aligned}$$

Figure 3.9: The concurrent Next action

in the set of active operations. *Delete2* removes a node from the list. This step is assumed to be atomic by our assumption that an atomic list implementation is available. Finally *Delete3* decrements count. Note that the previous step did not advance if a node was not removed, so the decrement step will always be taken if a delete operation has reached step 3.

Steps of a delete operation

$$\text{Delete1} \stackrel{\Delta}{=} \begin{array}{l} \text{Start a bucket, } nit \text{ if necessary} \\ \exists op \in \text{BagToSet}(activeOps) : \\ \quad \wedge op.type = "delete" \\ \quad \wedge op.step = 1 \\ \quad \wedge \text{IF } buckets[op.key \% size] = NULL \\ \quad \quad \text{THEN } activeOps' = (activeOps \ominus \text{SetToBag}(\{op\})) \\ \quad \quad \oplus \text{SetToBag}(\{[op \text{ EXCEPT } !["step"] = op.step + 1]\}) \\ \quad \quad \oplus \text{SetToBag}(\{[type \mapsto "bucket_init", step \mapsto 1, bucket \mapsto op.key \% size]\}) \\ \quad \text{ELSE } NextStep(op) \\ \quad \wedge \text{UNCHANGED } \langle list, buckets, size, count \rangle \end{array}$$

$$\text{Delete2} \stackrel{\Delta}{=} \begin{array}{l} \text{If the key is not there, end operation. Else, remove it} \\ \exists op \in \text{BagToSet}(activeOps) : \\ \quad \wedge op.type = "delete" \\ \quad \wedge op.step = 2 \\ \quad \wedge \text{IF } list[SORRegularKey(op.key)] = NULL \\ \quad \quad \text{THEN } \wedge End(op) \\ \quad \quad \wedge \text{UNCHANGED } list \\ \quad \text{ELSE } \wedge list' = [list \text{ EXCEPT } ![SORRegularKey(op.key)] = NULL] \\ \quad \quad \wedge NextStep(op) \\ \quad \wedge \text{UNCHANGED } \langle buckets, size, count \rangle \end{array}$$

$$\text{Delete3} \stackrel{\Delta}{=} \begin{array}{l} \text{Decrement count} \\ \exists op \in \text{BagToSet}(activeOps) : \\ \quad \wedge op.type = "delete" \\ \quad \wedge op.step = 3 \\ \quad \wedge count' = count - 1 \\ \quad \wedge End(op) \\ \quad \wedge \text{UNCHANGED } \langle buckets, list, size \rangle \end{array}$$
Figure 3.10: The steps of a Delete operation

4

Results

In this section we present the process and results of model checking the specification. First we describe the method of checking and the properties checked by the model. Then we present the quantitative results along with commentary on these results. Section 4.2 covers the technical details of this process.

4.1 Method

To demonstrate the invariants claimed by Shalev et al. [1] we use the TLC model checker to test the specifications outlined in Section 3.1. The first specification (Subsection 3.2.2) was tested by setting *HashmapSpec* as a property of the model, thus showing that *SplitOrder* implements the generic hashmap specification.

SOConcurrent is more granular than the abstract specification which makes it difficult to demonstrate implementation. Instead the invariants were written as predicates in TLA such that they can be checked by the TLC model checker. The following paragraphs describe these invariants in turn.

The first invariant states that the list beginning at bucket o is always sorted. Because our list is specified as a function from positions to values, this is trivially true for the specification.

The second invariant asserts that if a bucket is initialized, then it points to a dummy node in the list. This is captured by *BucketsInitialized* (Figure 4.1) which asserts each bucket is either NULL, or points to an initialized node in the list. This is an invariant on a single state and can be checked straightforwardly.

Each bucket is either uninitialized or points to a node with the dummy key

$$\text{BucketsInitialized} \triangleq$$

$$\forall i \in 0 \dots (\text{size} - 1) :$$

$$\vee \text{buckets}[i] = \text{NULL}$$

$$\vee \text{buckets}[i] = \text{SODummyKey}(i) \wedge \text{list}[\text{SODummyKey}(i)] = i$$

Figure 4.1: Invariant 2 as a TLA predicate

Invariant 3 and 4 are both statements about a behavior, rather than a state. This makes them liveness properties and therefore more difficult to check. Figure 4.2 shows half of each invariant, namely that *Insert(k)* succeeds if *k* is not present and *Delete(k)* succeeds if *k* is present. To do this they make use of the \diamond operator meaning "eventually" to claim "if an insert is initiated and the key is not in the map, then eventually the value will be inserted", and the opposite for *Delete*. Also note that both properties are written in terms of the constant set of all possible operation states because TLC cannot check liveness properties which include existential qualifiers (\exists and \forall) over variables.

The second halves of invariant 3 and 4 (Figure 4.3) introduce another complication. To show that an operation fails, it is necessary to differentiate them, so that we can make claims about the failure of a specific operation. The differentiation is done by introducing an additional ID element in each operation. Additionally the properties make use of the \square operator meaning "always" to claim "if an insert operation is initiated and the key is already in the map, that operation never reaches step 3".

Having formalized these invariants we are left with one safety property and four liveness properties, as well as the non-concurrent specification.

4.2 Technical Details

Tests were run with version 2.15 of the TLC model checker on a Intel(R) Core i5-8250U CPU with 8 GiB of memory and 250 GiB of hard drive space running Ubuntu 18.04. TLC was set to use 8 worker threads and 6 GiB of memory.

An insert with key not in map will succeed
 $InsertSucceeds \triangleq$

```

 $\forall op \in OperationStates :$ 
 $\text{IF } op.type = \text{"insert"}$ 
    This test is needed to avoid checking fields that do not exist in other types of operations
 $\text{THEN}$ 
     $\wedge op \in BagToSet(activeOps)$ 
     $\wedge op.step = 1$ 
     $\wedge list[SORRegularKey(op.key)] = \text{NULL}$ 
     $\implies \diamond(list[SORRegularKey(op.key)] = op.value)$ 
 $\text{ELSE TRUE}$ 

```

A delete with key in map will succeed
 $DeleteSucceeds \triangleq$

```

 $\forall op \in OperationStates :$ 
 $\text{IF } op.type = \text{"delete"}$ 
 $\text{THEN}$ 
     $\wedge op \in BagToSet(activeOps)$ 
     $\wedge op.step = 1$ 
     $\wedge list[SORRegularKey(op.key)] \neq \text{NULL}$ 
     $\implies \diamond(list[SORRegularKey(op.key)] = \text{NULL})$ 
 $\text{ELSE TRUE}$ 

```

Figure 4.2: The claimed invariants as TLA predicates

Unless otherwise noted, the maximum size of the map was equal to the number of possible keys and the load factor was set to half of the maximum size.

4.3 Quantitative Analysis

This section covers the results of model checking for three different specifications. Firstly the coarse non-concurrent specification and then the concurrent specification with and without operation IDs.

4.3.1 Non-concurrent specification

By setting *HashmapSpec* as a property while checking *SplitOrder*, the model checker tests if the latter implements the former. The implementation was checked for increasing sizes of the map, with results reported in Table 4.1. While the results of *SplitOrder* do not prove much about the structure in a concurrent setting, showing this implementation holds – at least for maps with

| Keys | Values | Diameter | Distinct States | Time (hh:mm:ss) |
|------|--------|----------|-----------------|-----------------|
| 2 | 4 | 10 | 2,523 | 00:00:01 |
| 2 | 8 | 10 | 23,763 | 00:00:08 |
| 2 | 16 | 10 | 279,075 | 00:01:02 |
| 4 | 2 | 17 | 39,827 | 00:00:09 |
| 4 | 4 | 17 | 1,790,067 | 00:03:44 |
| 4 | 8 | 17 | 147,266,723 | 07:14:45 |

Table 4.1: Model checking results for SplitOrder

up to 4 keys and 8 values – helps to increase our confidence that it does in fact implement an abstract hashmap structure.

It is also noteworthy that even this relatively simple specification ran out of hard-drive space (using roughly 150 GB) trying to test a map with 4 possible keys and 16 possible values.

4.3.2 Concurrent Specification

In the concurrent specification each invariant was checked in isolation, again increasing the size of the state space incrementally.

Table 4.2 shows the results of checking *InsertSucceeds* and *DeleteSucceeds*. Both exhaust the available memory attempting to check a map with 4 possible keys and 3 possible values in approximately 30 minutes. However, both hold for a smaller map with only 2 possible values and this completed in only a few minutes. Also note that *DeleteSucceeds* runs out of memory having explored all most 5 times the number of unique states. The discrepancy in space explored is likely because an insert operation takes 4 steps, while a delete takes only 3 leading to shorter behavior traces for deletes.

Table 4.2 also shows the result of checking *BucketsInitialized* on the same specification. Since *BucketsInitialized* is a safety property, memory is not consumed as quickly as with the earlier liveness properties and *BucketsInitialized* was shown to hold for up to 4 keys and 5 values. Additionally this property was checked for larger numbers of concurrent operations. Note that a map with 4 keys and 2 values was fully explored for 2 concurrent operations in 22 seconds, but took 14 minutes with 3 concurrent operations. 4 concurrent operations forced a reboot after 26 minutes.

Table 4.3 and 4.4 describe model checking runs which failed to complete due to memory or disk space constraints.

| Invariant | Keys | Values | Conc. Ops | Diameter | Distinct States | Time (mm:ss) |
|---------------------------|------|--------|-----------|----------|-----------------|--------------|
| <i>InsertSucceeds</i> | 2 | 2 | 2 | 38 | 10,083 | 00:02 |
| | 2 | 4 | 2 | 38 | 66,901 | 00:13 |
| | 2 | 4 | 3 | 45 | 1,351,453 | 01:50 |
| | 4 | 2 | 2 | 62 | 1,627,390 | 02:15 |
| <i>DeleteSucceeds</i> | 2 | 2 | 2 | 38 | 10,083 | 00:01 |
| | 2 | 4 | 2 | 38 | 66,901 | 00:07 |
| | 2 | 4 | 3 | 45 | 1,351,453 | 00:31 |
| | 4 | 2 | 2 | 62 | 1,627,390 | 00:45 |
| <i>BucketsInitialized</i> | 2 | 2 | 2 | 38 | 10,083 | 00:02 |
| | 2 | 4 | 2 | 38 | 66,901 | 00:16 |
| | 2 | 4 | 8 | 38 | 624,645 | 00:16 |
| | 2 | 4 | 16 | 38 | 7,371,157 | 01:55 |
| | 4 | 2 | 2 | 62 | 1,627,390 | 00:22 |
| | 4 | 3 | 2 | 62 | 8,368,282 | 01:39 |
| | 4 | 4 | 2 | 62 | 29,973,646 | 06:10 |
| | 4 | 5 | 2 | 62 | 85,419,916 | 18:41 |
| | 4 | 2 | 3 | 75 | 61,353,460 | 13:59 |

Table 4.2: TLC model checking results for SOConcurrent

| Invariant | Keys | Values | Conc. Ops | Diameter | Distinct States | Time (hh:mm:ss) | Error |
|---------------------------|------|--------|-----------|----------|-----------------|-----------------|-------|
| Non-Concurrent | 4 | 16 | | 17 | 84,587,043 | 02:54:11 | 3 |
| <i>InsertSucceeds</i> | 4 | 3 | 2 | 27 | 1,426,888 | 00:33:30 | 2 |
| <i>DeleteSucceeds</i> | 4 | 3 | 2 | 39 | 6,067,795 | 00:30:30 | 1 |
| <i>BucketsInitialized</i> | 4 | 2 | 4 | 57 | 120,175,837 | 00:26:49 | 4 |

Table 4.3: Failed model checks

For the final two invariants it was necessary to use a slightly different specification in which operations are associated with an ID. Because previous model resulted in a diameter of 62, this model was set up to use a maximum of 62 unique IDs. Differentiating between individual operations leads to a significant increase in both diameter and distinct states because equivalent sets of active operations are now seen as distinct states.

Table 4.5 shows the results of checking this model with operation IDs. The depth of the state space even for minimal key/value sets makes it infeasible to check liveness properties for this specification.

| Code | Error |
|------|----------------------------|
| 1 | GC Overhead limit exceeded |
| 2 | Out of heap space |
| 3 | No space left on device |
| 4 | Unknown |

Table 4.4: Error Codes

| Invariant | Keys | Values | Conc. Ops | Diameter | Distinct States | Time (hh:mm:ss) | Error |
|---------------------------|------|--------|-----------|----------|-----------------|-----------------|-------|
| <i>BucketsInitialized</i> | 2 | 2 | 2 | 406 | 59,272,902 | 00:07:08 | |
| | 2 | 4 | 2 | 406 | 387,645,300 | 00:54:33 | |
| | 4 | 2 | 2 | 153 | 665,993,030 | 04:22:12 | |
| <i>InsertFails</i> | 2 | 2 | 2 | >20 | >64,000 | >00:01:07 | 2 |
| <i>DeleteFails</i> | 2 | 2 | 2 | >25 | >153,370 | >00:02:12 | 1 |

Table 4.5: Model checking SOConcurrent with operation IDs

An insert with key in map will not reach step 3

$InsertFails \triangleq$

```

 $\forall op \in OperationStates :$ 
   $\text{IF } op.type = \text{"insert"}$ 
     $\text{THEN}$ 
       $\wedge op \in (activeOps)$ 
       $\wedge op.step = 1$ 
       $\wedge list[SORRegularKey(op.key)] \neq NULL$ 
       $\implies \diamond\Box(\neg([op \text{ EXCEPT } !["step"] = 3] \in (activeOps)))$ 
   $\text{ELSE } \text{TRUE}$ 

```

A delete with key not in map will not reach step 3

$DeleteFails \triangleq$

```

 $\forall op \in OperationStates :$ 
   $\text{IF } op.type = \text{"delete"}$ 
     $\text{THEN}$ 
       $\wedge op \in (activeOps)$ 
       $\wedge op.step = 1$ 
       $\wedge list[SORRegularKey(op.key)] = NULL$ 
       $\implies \diamond\Box(\neg([op \text{ EXCEPT } !["step"] = 3] \in (activeOps)))$ 
   $\text{ELSE } \text{TRUE}$ 

```

$OperationStates \triangleq$

Set of all possible active operations

They can be of type insert, delete or bucket_init

Step denotes the current step of the operation

$[type : \{"insert"\}, step : 1 .. 4, key : PossibleKeys, value : PossibleValues, id : 0 .. MaxID]$
 \cup

$[type : \{"delete"\}, step : 1 .. 3, key : PossibleKeys, id : 0 .. MaxID]$
 \cup

$[type : \{"bucket_init"\}, step : 1 .. 3, bucket : 0 .. (MaxSize - 1), id : 0 .. MaxID]$

Figure 4.3: The failure invariants as TLA predicates

/ 5

Concluding Remarks

In this chapter we first discuss the results of model checking in relation to the problem statement, then follows a discussion of the experience of writing and testing the specification, and the lessons learned in this process. Finally we conclude our work and answer the problem statement.

5.1 The Results of Model Checking

The results of model-checking the non-concurrent specification show that it does implement the higher level abstract specification. While it does not show the correctness of the structure in a concurrent setting, this result gives us reason to believe the overall structure is sound if the operations are linearizable.

The concurrent model shows similarly promising results for all but the *InsertFails* and *DeleteFails* properties. Due to memory constraints, the model could not be used to check very large state spaces, but the positive results on small subsets give some credence to the correctness of the hashmap.

In the case of *InsertFails* and *DeleteFails* the need to differentiate individual operations lead to an explosion in the size of the state space so large it became impossible to check these properties. In particular the diameter is a problem because they are both liveness properties and so require storing the very long execution trace.

One issue with model checking specifications as a way to prove the correctness of programs is the correctness of the model checking. The problem is two-fold: how certain can we be that the model checker does its job correctly; and how certain can we be that the specification is really specifying what we want it to?

The final specification behaves as expected along several axes. Type correctness holds across all the tested invariants (though it was not checked along with liveness properties to reduce computational load), the size of the bucket array and the set of active operations both stay within bounds, and additional possible values increases the number of unique states without increasing the depth of the state space while an increase in the number of possible keys increases the depth.

A surprising feature is the sheer size of the state space. Even for two possible keys and two possible values, the state space of the concurrent specification is surprisingly large (10,083 unique states). One way to evaluate the specification would be a calculation of the expected state space of the system with the same parameters. If such a calculation agreed with the results of model checking it would increase our confidence in the specification accurately describing the system.

The results of model checking do not conclusively prove correctness because they must necessarily be finite in scope. In the case where limited computing power and memory is available, they are definitely finite and indeed quite small. One could argue that correctness on a small sample space implies total correctness, but such an argument can not be based on model checking alone. Again, some human ingenuity is needed. However, successful model checking on a small sample space should increase our confidence in the correctness of the structure.

Model checking with TLC can be used to exhaustively show the correctness of a structure on a small subset of the sample space. The results on this subset correspond to the properties formally proven by Shalev et al. with the exception of operation failures which proved impossible to check with the computational resources available.

5.2 Lessons Learned

In this section we will explain some of the choices made in the development of the specification and discuss the experiences and lessons learned from the process. We present three specific lessons from this project along with concrete experiences and possible solutions.

The three specific lessons:

1. decide on the purpose of the specification as early as possible
2. test assumed invariants and try to break them
3. choose step size early and deliberately

1. It is useful to clearly decide on the purpose of the specification as early as possible. On several occasions we discovered that the specification was not suitable for its purpose. First, the non-concurrent spec proved difficult to map to an abstract hashmap spec. Then, the concurrent spec turned out to be unsuitable for showing the failure of operations because it did not differentiate between individual operations and so could not show that a specific operation was aborted. In both cases it would have been useful to be clear about precisely what we wished to show at an earlier state and plan the specification accordingly.

The clearest example of this error was our attempt to show the correct behavior of a non-concurrent specification through implementation. The idealized hashmap specification proved impossible to directly map to the SplitOrder specification because such a mapping would involve a *find* operation which may result in a bucket having to be initialized and this sort of two-level action is not permitted in refinement mappings in TLA+. Instead we introduced a set of auxillary variables [15] which maintain an idealized map that is updated in atomic steps. While auxillary variables are a possible solution, they introduced additional complexity and made it more difficult to see that the specification accurately describes the structure.

Additionally this initial non-concurrent specification was not well suited to showing the precise invariants claimed by Shalev et al. and was difficult to adapt to a concurrent version. Because the structure of the specification was modeled after pseudocode, and therefore divided into presumed linearizable functions, adapting it to describe a concurrent setting required a full rewrite.

Finally, neither specification was well suited to showing failure invariants: the claim that certain operations would always abort. In the non-concurrent case this was because each operation was a single action and so could not abort at all, and in the concurrent case it was because an operation was not distinguishable from a later operation with the same arguments. While the concurrent specification was relatively easy to expand with operation IDs, it was still unable to demonstrate these properties due to state space explosion. It

is possible that a specification that was designed from the beginning with these properties in mind would have been better suited to demonstrate them.

- 2.** A useful tool when writing specifications in TLA+ is making and testing assumptions. If you think something is true about the specification, the assumption can be written as a property or invariant and checked using the model checker. Both positive results – when invariants believed to be true hold, or invariants believed to be false are shown to be false – and negative results – when invariants believed to hold are violated – can be extremely useful. It is also useful to purposefully attempt to violate assumptions by changing the specification.

One such case occurred in early testing of the *DeleteFails* and *InsertFails* properties. An early attempt was tautologically true because it described operations terminating at **some** later stage, which was true whether the operation aborted or completed. This error was discovered by removing the terminating clause from the specification and observing that the property still appeared to hold. False positives like this error are very difficult to find without making changes to the specification and attempting to violate assumptions.

Another error occurred with an early version of the concurrent specification. Because the active operations were kept in a set, there could never be two identical operations active at the same time. In practice this meant one operation would "disappear" if another identical operation was initiated. The error was discovered because it led to an ever increasing *count* when *Delete* operations disappeared. By checking the invariant $count \leq 4$ on a model with only four possible values, an error trace showing this error was obtained. As a result, the specification was changed to use a multiset (Bag) to allow for multiple copies of the same operation.

- 3.** Finally, it is advisable to decide on the "step size" of the specification as early as possible. [16]. By step size we mean roughly the number of variables that change in each action. For the concurrent specification this was done by identifying the variables that constitute the state of the map: the list, the buckets, size, and count; and then making each change to one of these a single step. Making this choice deliberately and clearly made it much easier to write the concurrent specification than the non-concurrent one where no such considerations were made.

The choice of step size carries with it assumptions of atomicity. By allowing *list* and *size* to change in one action we assume that elements can be inserted and removed from the list and that an integer can be fetched and incremented atomically.

In this section we have described three concrete lessons learned from the project that are applicable to future work on specifications in TLA+. These were motivated by concrete experiences during the project.

5.3 The Usefulness of Formal Specification

Finally, a note on the usefulness of writing and model checking specifications: Sometimes formal specification can uncover rare or complex errors in algorithms and protocols [14, 17], but in this case it did not. Instead, our result may slightly increase confidence in the correctness of Shalev et al.’s hashmap, but arguably no more than a working implementation passing tests. That increase is a poor return on investment, especially if the user has to learn the tools.

It seems that formal specification and model checking has two uses:

1. as a way to produce minimal examples of a suspected error, and
2. as a development tool for algorithms and protocols

The first case is useful because it lets developers find and show errors in the algorithms, rather than in programs or implementations. In the second case, a specification is used to test and improve an algorithm or protocol under development without resorting to implementation. This is especially useful when an implementation requires hardware [13] or is prohibitively large in scale [3] and lets developers iterate on designs while ensuring their correctness in a similar way to test-driven development for implementation.

5.4 Conclusion

In this project a formal specification of Shalev et al.’s lock free hashmap implementation has been written in TLA+ and the TLC model checker has been used to demonstrate their claimed invariants on state spaces with up to 4 keys, 16 possible values and 3 concurrent processes.

We recall the problem statement from Section 1.1:

Shalev et al.’s split-ordered list design is a correct extensible hashmap for concurrent systems.

The results of model checking confirmed the claimed invariants of Shalev et al.'s design for maps with up to four keys and two possible values, with the exception of failure conditions for *Delete* and *Insert*. The testing was limited by the memory required to test liveness properties.

Additionally the use of formal specification in TLA+ and the TLC model checker were discussed and three lessons were formulated based on concrete experiences during the process.

We conclude that model checking gives increased confidence in the correctness of Shalev et al.'s split-order hashmap, but that it is poorly suited to showing liveness properties in the absence of significant memory resources.

Finally, we note that model checking may have more value as a development tool for protocols, data structures and algorithms than as a way to prove correctness of existing programs.

Bibliography

- [1] Ori Shalev and Nir Shavit. “Split-Ordered Lists: Lock-Free Extensible Hash Tables.” In: *J. ACM* 53.3 (2006), pp. 379–405. ISSN: 0004-5411. DOI: 10.1145/1147954.1147958. URL: <https://doi.org/10.1145/1147954.1147958>.
- [2] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th. USA: Prentice Hall Press, 2014. ISBN: 013359162X.
- [3] Chris Newcombe et al. “How Amazon Web Services Uses Formal Methods.” In: *Commun. ACM* 58.4 (Mar. 2015), pp. 66–73. ISSN: 0001-0782. DOI: 10.1145/2699417. URL: <https://doi.org/10.1145/2699417>.
- [4] Daniel Cederman et al. *Lock-free Concurrent Data Structures*. 2013. arXiv: 1302.2757 [cs.DC].
- [5] Rodrigo Medeiros Duarte et al. “Concurrent Hash Tables for Haskell.” In: *Programming Languages*. Ed. by Fernando Castor and Yu David Liu. Cham: Springer International Publishing, 2016, pp. 110–124. ISBN: 978-3-319-45279-1.
- [6] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming.” In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: <https://doi.org/10.1145/363235.363259>.
- [7] Robert W. Floyd. “Assigning Meanings to Programs.” In: *Proceedings of Symposium on Applied Mathematics* 19 (1967), pp. 19–32. URL: <http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf>.
- [8] Amir Pnueli. “The Temporal Logic of Programs.” In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS ’77. USA: IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32. URL: <https://doi.org/10.1109/SFCS.1977.32>.
- [9] L Lamport. “Proving the Correctness of Multiprocess Programs.” In: *IEEE Trans. Softw. Eng.* 3.2 (1977), pp. 125–143. ISSN: 0098-5589. DOI: 10.1109/TSE.1977.229904. URL: <https://doi.org/10.1109/TSE.1977.229904>.
- [10] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. “Model Checking: Algorithmic Verification and Debugging.” In: *Commun. ACM* 52.11 (Nov. 2009), pp. 74–84. ISSN: 0001-0782. DOI: 10.1145/1592761.1592781. URL: <https://doi.org/10.1145/1592761.1592781>.

- [11] Leslie Lamport et al. “Specifying and Verifying Systems with TLA+.” In: *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. EW 10. New York, NY, USA: ACM, 2002, pp. 45–48. DOI: 10.1145/1133373.1133382. URL: <http://doi.acm.org/10.1145/1133373.1133382>.
- [12] Microsoft. *TLA+ Toolbox*. <https://github.com/tlaplus/tlaplus>. 2003.
- [13] Brannon Batson and Leslie Lamport. “High-Level Specifications: Lessons from Industry.” In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 242–261. ISBN: 978-3-540-39656-7.
- [14] Jørgen Aarmo Lund. “Verification of the Chord protocol with TLA+.” In: (2019). URL: <https://munin.uit.no/bitstream/handle/10037/15613/thesis.pdf?sequence=2%7B%5C%7DisAllowed=y>.
- [15] Leslie Lamport. “Hiding , Refinement , and Auxiliary Variables.” In: July (2019).
- [16] Leslie Lamport. *The TLA+ Video Course*. Online Video. Mar. 2018.
- [17] Pamela Zave. “Using Lightweight Modeling to Understand Chord.” In: *SIGCOMM Comput. Commun. Rev.* 42.2 (Mar. 2012), pp. 49–57. ISSN: 0146-4833. DOI: 10.1145/2185376.2185383. URL: <https://doi.org/10.1145/2185376.2185383>.