UiT THE ARCTIC UNIVERSITY OF NORWAY

Faculty of Science and Technology
Department of Computer Science

Formal Verification of a Lock-free Split-order Hashmap

—

Åsmund Aqissiaq Arild Kløvstad

INF-2990 Bachelor's thesis in Informatics June 1, 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

List of Tables

1

Introduction

Concurrent and distributed systems are extremely important in modern software development. Due to the difficulty of developing ever smaller and more powerful CPUs the trend in hardware design since about 2005 has been to increase the number of cores to allow for high levels of parallelization [1]. Additionally important areas of computing such as image processing and machine learning lend themselves well to such parallelization. [citation needed, Phuong?] 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 [2].

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 [citations], but the overhead of lock management and difficulty of resizing often make these impractical or inefficient [3]. A lock free alternative is proposed by Shalev and Shavit in [3]. This approach has proven to be useful [4] and scale better with number of concurrent processes than lock-based approaches [5].

In both small- and large-scale computer systems it is important to ensure correctness. This is especially evident in critical infrastructure, but all scales and importance levels benefit from confidence in the correctness of their systems. [citation needed]

It is therefore troublesome that such systems are incredibly difficult to design, debug and reason about. The complexity of interactions between processes and sheer number of possible edge cases makes it infeasible for a person to determine correctness.

Early solutions to the problem of proving correctness include Hoare [6], Floyd [7] and Pnueli's [8] temporal logics and Leslie Lamport's Temporal Logic of Actions [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. One such model checker is the TLC model checker based on Lamport's TLA and incorporated in the TLA+ IDE.

1.1 Thesis

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

Furthermore it is possible to check this using a formal model checker, and the results of this will correspond to the properties proven by Shalev et al.

1.2 Method

In order to prove the correctness of the hashmap, its behavior will be implemented as a specification in TLA+ [11] and the TLC model checker will be used to test the claimed invariants.

The specification will be developed in stages of increasing granularity, assuming the atomicity of operations to begin with and gradually loosening assumptions.

1.3 Scope?

1.4 Outline

Chapter 2 discusses the motivation for formal verification and model checking, followed by a description of Temporal Logic in Section 2.1 and the TLA+ language in Section 2.2. Finally Shalev et al.'s hashmap design is described in Section 2.3.

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 thesis, and explores experiences and lessons lerned.

2

Background

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 develpment 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 [2] 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 inprecise 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.

2.1 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 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 exlusion 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 $s = \langle \pi_1, \pi_2, ..., \pi_n, u \rangle$ 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:

$$p \leadsto q : \forall t_1 \exists t_2 \ s.t \ t_1 \leq t_2, H(p, t_1) \implies H(q, t_2)$$

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 $\langle \pi = l_0, u = \phi \rangle \leadsto \langle \pi = l_m, u = \psi \rangle$ where ϕ denotes correct input, and ψ denotes correct output and l_0, l_m are the start and end labeles of the system, respectively.

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

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

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

$$[\forall s, s' p(s) \wedge R(s, s') \implies q(s')] \Rightarrow p \leadsto q \tag{A1}$$

$$(p \implies q) \Rightarrow p \leadsto q \tag{A2}$$

Figure 2.1: Pnueli's axioms

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.

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.

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

$$p \rightsquigarrow q, q \rightsquigarrow r \Rightarrow p \rightsquigarrow r \tag{R2}$$

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

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

Figure 2.2: Pnueli's inference rules

In a concurrent program we generalize q to hold when any π_i is updated by N and construct either a full attachment

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

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" **F**uture and **G**lobal on predicates such that at some time n

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

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

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 $p \rightsquigarrow q$)

2.2 TLA+

2.3 Split-Ordered List Hashmap

Maybe a description of hashmaps in general here to set up the later use of table, list, key etc. or is that assumed knowledge?

Shalev et al. [3] present 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 $\bmod 2^i$ 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 $\bmod 2^i$. This is because such an ordering corresponds to difference in the keys' ith least significant bit, which is equivalent to having a different remainder $\bmod 2^i$.

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 unitialized 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.

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.

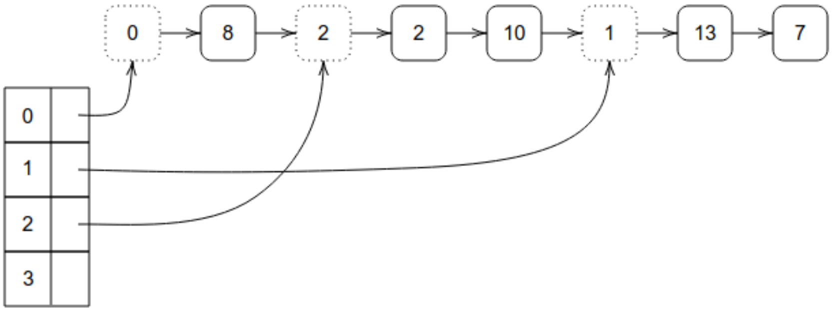
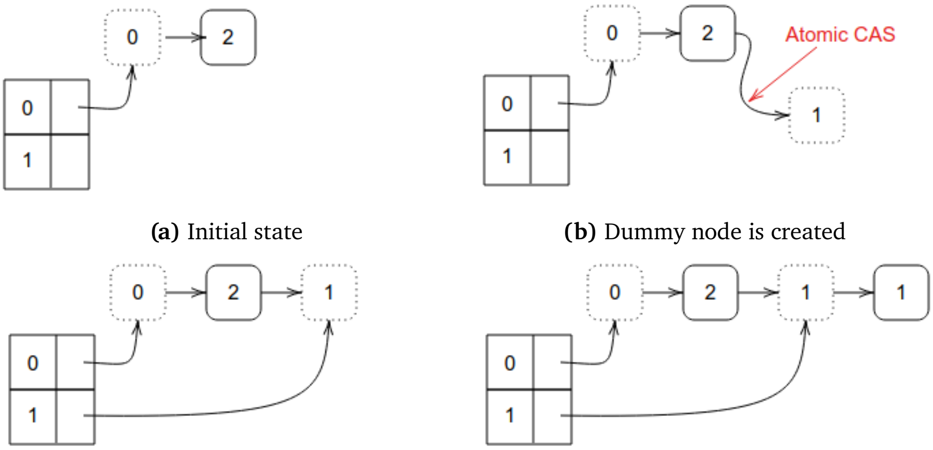


Figure 2.3: The layout of the split-ordered list



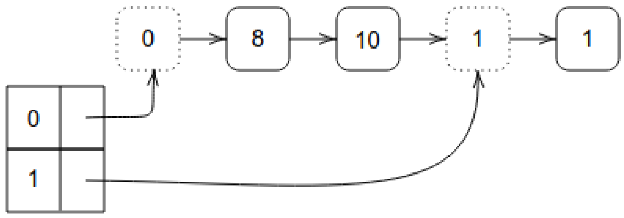
(a) Initial state

(b) Dummy node is created

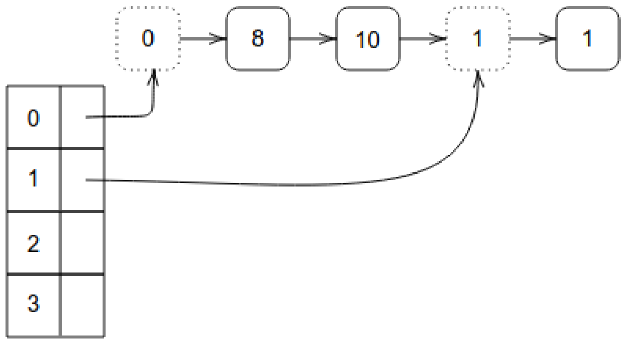
(c) Dummy node is inserted in bucket

(d) New node is inserted

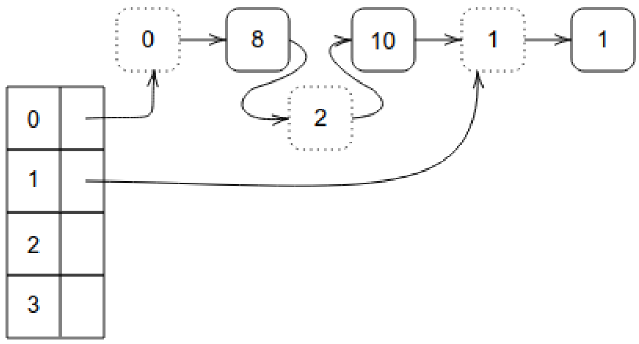
Figure 2.4: Insertion without bucket splitting



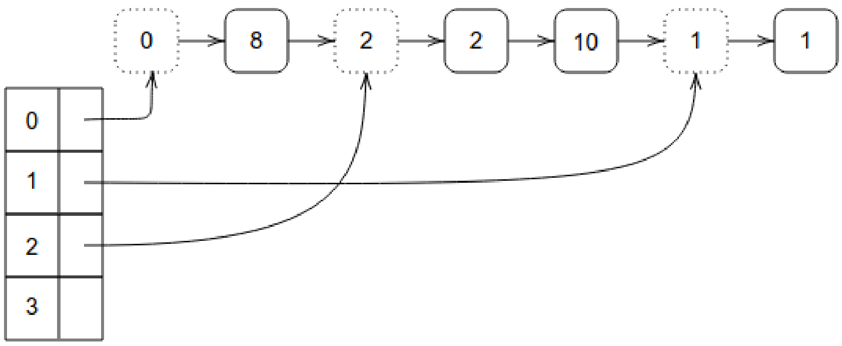
(a) Initial state



(b) Table is expanded



(c) Dummy node for new bucket inserted



(d) Table entry points to bucket node

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.

The specification consists of two main parts:

1. a generic hashmap and
2. an [insert name of structure here] *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 [insert name of structure] specification describes Shalev et al.'s specific structure and algorithms for implementing hashmap functionality.

We present two versions of the specification at two levels of granularity. The first assumes the atomicity of hashmap operations, while the second describes entirely concurrent operations.

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 [12] we will not check claims about the performance of the implementation. This leaves 4 invariants:

1. the list beginning at bucket 0 is always sorted

2. if a bucket is initialized, then it points to a dummy node which is in the list beginning at bucket 0

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 corerct 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. Updating the map uses the `EXCEPT` construct, to say "the map is the same except the new key maps to the new value". 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 and the same `EXCEPT` construct as in insert. Both operations are idempotent, so attempting to remove a key that is not in the map will result in no change.

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

$$
\begin{aligned}
&HashmapInit \triangleq \quad \land\ keys = \{\} \\
&\quad \land\ map = [k \in PossibleKeys \mapsto NULL]
\end{aligned}
$$

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

$$
\begin{aligned}
&Insert \triangleq \quad \exists\, k \in PossibleKeys : \\
&\qquad \exists\, v \in PossibleValues : \\
&\qquad\qquad \land\ keys' = keys \cup \{k\} \\
&\qquad\qquad \land\ map' = [map \text{ EXCEPT } ![k] = v]
\end{aligned}
$$

Remove sets exactly one mapping to NULL

$$
\begin{aligned}
&Remove \triangleq \quad \exists\, k \in PossibleKeys : \\
&\qquad \land\ keys' = keys \setminus \{k\} \\
&\qquad \land\ map' = [map \text{ EXCEPT } ![k] = NULL]
\end{aligned}
$$

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

$$
\begin{aligned}
&HashmapNext \triangleq \quad \lor\ Insert \\
&\quad \lor\ Remove
\end{aligned}
$$

TypeOK asserts all keys and values are of the right type

$$
\begin{aligned}
&TypeOK \triangleq \quad \forall\, k \in keys : \\
&\qquad \land\ k \in PossibleKeys \\
&\qquad \land\ map[k] \in PossibleValues
\end{aligned}
$$

KeyHasValue asserts that every key is mapped to a value

$$KeyHasValue \triangleq \forall\, k \in keys : \neg(map[k] = NULL)$$

The hashmap specification as a temporal formula

$$HashmapSpec \triangleq HashmapInit \land \Box[HashmapNext]_{\langle keys,\, map\rangle}$$

Figure 3.1: The hashmap specification

The specification also contains temporal formulae *TypeOK* and *KeyHasValue* that specify correct behavior. These have been checked with the TLC model checker [insert results here maybe?].

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.

The theorem $SOSpec \implies HashmapSpec$ 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 *HashmapSpec* in a model checking SOSpec.

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 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 [3]. 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

EXTENDS $Integers$

CONSTANTS $NULL$, $PossibleKeys$, $PossibleValues$, $LoadFactor$, $MaxSize$

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

ASSUME
$\land\ PossibleKeys \subseteq 0\,..\,15$
$\land\ NULL \notin PossibleKeys$
$\land\ NULL \notin PossibleValues$

The Init for split-order keys is initially empty the map maps every possible key to NULL The list initially contains only the 0 dummy node

$SOInit \triangleq\ \land\ keys = \{\}$
$\land\ AuxKeys = \{\}$
$\land\ list = [n \in 0\,..\,255 \mapsto \textsc{if}\ n = 0\ \textsc{then}\ SODummyKey(0)\ \textsc{else}\ NULL]$
$\land\ buckets = [m \in PossibleKeys \mapsto \textsc{if}\ m = 0\ \textsc{then}\ SODummyKey(0)\ \textsc{else}\ NULL]$
$\land\ size = 1$
$\land\ count = 0$
$\land\ map = [k \in PossibleKeys \mapsto NULL]$

Figure 3.2: The initial state of the specification

$SOInsert \triangleq\ \land\ \exists\, k \in PossibleKeys :$
$\exists\, v \in PossibleValues :$
$BucketInsert(k, v)$

$BucketInsert(k, v) \triangleq$
Either a bucket needs to be initialized
$\lor\ \land\ buckets[k\%size] = NULL$
$\land\ BucketInit(k\%size)$
$\land\ ListInsert(SORegularKey(k), v)$
$\land\ AuxKeys' = AuxKeys \cup \{k\}$
Or the bucket is already initialized
$\lor\ \land\ buckets[k\%size] \neq NULL$
$\land\ ListInsert(SORegularKey(k), v)$
$\land\ AuxKeys' = AuxKeys \cup \{k\}$
$\land\ \textsc{unchanged}\ \langle buckets \rangle$

$ListInsert(k, v) \triangleq\ \textsc{if}\ list[k] = NULL$
$\textsc{then}\ list' = [list\ \textsc{except}\ ![k] = v] \land count' = count + 1$
$\textsc{else}\ \textsc{unchanged}\ \langle list, count \rangle$

Figure 3.3: The *Insert* operation

$$
\begin{aligned}
SONext \triangleq\ & \lor SOInsert \land BucketGrow \land \text{UNCHANGED } \langle map,\ keys \rangle \\
& \lor SORemove \land \text{UNCHANGED } \langle size,\ map,\ keys \rangle \\
& \lor SOFind \land \text{UNCHANGED } \langle size,\ count,\ list \rangle
\end{aligned}
$$

Figure 3.4: The Next action

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

Figure 3.5: The *SOFind* operation

has a value.

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

3.2.3 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

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

Figure 3.6: Implementation of generic hashmap

$OperationStates \triangleq$

Set of all possible active operations

They can be of type insert, delete or bucket_init

Step denotes the current step of the operation

$[type : \{\text{"insert"}\}, step : 1 \,..\, 4, key : PossibleKeys, value : PossibleValues]$
$\cup$
$[type : \{\text{"delete"}\}, step : 1 \,..\, 3, key : PossibleKeys]$
$\cup$
$[type : \{\text{"bucket\_init"}\}, step : 1 \,..\, 3, bucket : 0 \,..\, (MaxSize - 1)]$

Figure 3.7: The operation structures

Begin an insert operation

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

Begin a delete operation

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

Begin a bucket_init operation

$BucketInit(b) \triangleq activeOps' = activeOps \cup \{[type \mapsto \{\text{"bucket\_init"}\}, step \mapsto 1, bucket \mapsto b]\}$

$NextStep(op) \triangleq activeOps' = (activeOps \setminus \{op\}) \cup \{[op \text{ EXCEPT } ![\text{"step"}] = op.step + 1]\}$
$End(op) \triangleq activeOps' = activeOps \setminus \{op\}$

Figure 3.8: Starting and stepping through operations

$SONext \triangleq$
$\lor \quad \land \exists k \in PossibleKeys :$
$\quad \exists v \in PossibleValues :$
$\quad Insert(k, v)$
$\quad \land \text{ UNCHANGED } \langle buckets, count, list, size \rangle$
$\lor \quad \land \exists k \in PossibleKeys :$
$\quad Delete(k)$
$\quad \land \text{ UNCHANGED } \langle buckets, count, list, size \rangle$
$\lor Insert1$
$\lor Insert2$
$\lor Insert3$
$\lor Insert4$
$\lor Delete1$
$\lor Delete2$
$\lor Delete3$
$\lor BucketInit1$
$\lor BucketInit2$
$\lor BucketInit3$

Figure 3.9: The concurrent Next action

Steps of a delete operation

$$
\begin{array}{l}
Delete1 \triangleq \\
\quad \text{Start a bucket}_{i}nitifnecessary \\
\quad \exists\, op \in activeOps : \\
\quad\quad \land\ op.type = \{\text{"delete"}\} \\
\quad\quad \land\ op.step = 1 \\
\quad\quad \land\ \textsc{if}\ buckets[op.key \% size] = NULL \\
\quad\quad\quad \textsc{then}\ activeOps' = (activeOps \setminus \{op\}) \\
\quad\quad\quad\ \cup \{[op\ \textsc{except}\ ![\text{"step"}] = op.step + 1]\} \\
\quad\quad\quad\ \cup \{[type \mapsto \{\text{"bucket\_init"}\},\ step \mapsto 1,\ bucket \mapsto op.key \% size]\} \\
\quad\quad\quad \textsc{else}\ \ NextStep(op) \\
\quad\quad \land\ \textsc{unchanged}\ \langle list,\ buckets,\ size,\ count \rangle
\end{array}
$$

$$
\begin{array}{l}
Delete2 \triangleq \\
\quad \text{If the key is not there, end operation. Else, remove it} \\
\quad \exists\, op \in activeOps : \\
\quad\quad \land\ op.type = \{\text{"delete"}\} \\
\quad\quad \land\ op.step = 2 \\
\quad\quad \land\ \ \textsc{if}\ list[SORegularKey(op.key)] = NULL \\
\quad\quad\quad \textsc{then}\ \ \land\ End(op) \\
\quad\quad\quad \land\ \textsc{unchanged}\ list \\
\quad\quad\quad \textsc{else}\ \ \ \land\ list' = [list\ \textsc{except}\ ![SORegularKey(op.key)] = NULL] \\
\quad\quad\quad \land\ NextStep(op) \\
\quad\quad \land\ \textsc{unchanged}\ \langle buckets,\ size,\ count \rangle
\end{array}
$$

$$
\begin{array}{l}
Delete3 \triangleq \\
\quad \text{Decrement count} \\
\quad \exists\, op \in activeOps : \\
\quad\quad \land\ op.type = \{\text{"delete"}\} \\
\quad\quad \land\ op.step = 3 \\
\quad\quad \land\ count' = count - 1 \\
\quad\quad \land\ End(op) \\
\quad\quad \land\ \textsc{unchanged}\ \langle buckets,\ list,\ size \rangle
\end{array}
$$

Figure 3.10: The steps of a *Delete* operation

the structure in a concurrent setting. This is because its structure presupposes the linearizability of its operations by making them action statements. This 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 set 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). *Delete*1 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 in the set of active operations. *Delete*2 removes a node from the list. This step is assumed to be atomic by our assumption that an atomic list implementation is available. Finally *Delete*3 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.

4 Results

4.1 Method

To demonstrate the invariants claimed by Shalev et al. [3] 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 0 is always sorted. Because our list is specified as a function from positions to values, this is trivialy 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.

Invariant 3 and 4 are both statements about a behavior, rather than a state.

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

$$
\begin{aligned}
&BucketsInitialized \triangleq \\
&\quad \forall\, i \in 0\,..\,(size - 1) : \\
&\qquad \lor\ buckets[i] = NULL \\
&\qquad \lor\ buckets[i] = SODummyKey(i) \land list[SODummyKey(i)] = i
\end{aligned}
$$

Figure 4.1: Invariant 2 as a TLA predicate

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 (∃ and ∀) over variables.

An insert with key not in map will succeed

$$
\begin{aligned}
&InsertSucceeds \triangleq \\
&\quad \forall\, op \in OperationStates : \\
&\qquad \text{IF } op.type = \text{“insert”} \\
&\qquad \text{This test is needed to avoid checking fields that do not exist in other types of operations} \\
&\qquad \text{THEN} \\
&\qquad\quad \land\ op \in BagToSet(activeOps) \\
&\qquad\quad \land\ op.step = 1 \\
&\qquad\quad \land\ list[SORegularKey(op.key)] = NULL \\
&\qquad\quad \implies \Diamond(list[SORegularKey(op.key)] = op.value) \\
&\qquad \text{ELSE TRUE}
\end{aligned}
$$

A delete with key in map will succeed

$$
\begin{aligned}
&DeleteSucceeds \triangleq \\
&\quad \forall\, op \in OperationStates : \\
&\qquad \text{IF } op.type = \text{“delete”} \\
&\qquad \text{THEN} \\
&\qquad\quad \land\ op \in BagToSet(activeOps) \\
&\qquad\quad \land\ op.step = 1 \\
&\qquad\quad \land\ list[SORegularKey(op.key)] \neq NULL \\
&\qquad \implies \Diamond(list[SORegularKey(op.key)] = NULL) \\
&\qquad \text{ELSE TRUE}
\end{aligned}
$$

Figure 4.2: The claimed invariants as TLA predicates

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 □ 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".

An insert with key in map will not reach step 3

$$
\begin{array}{l}
\textit{InsertFails} \triangleq \\
\quad \forall\, op \in \textit{OperationStates} : \\
\quad\quad \text{IF } op.type = \text{“insert”} \\
\quad\quad \text{THEN} \\
\quad\quad\quad \land\; op \in (\textit{activeOps}) \\
\quad\quad\quad \land\; op.step = 1 \\
\quad\quad\quad \land\; list[\textit{SORegularKey}(op.key)] \neq \textit{NULL} \\
\quad\quad \implies \quad \Diamond\Box(\neg([op \text{ EXCEPT } ![\text{“step”}] = 3] \in (\textit{activeOps}))) \\
\quad\quad \text{ELSE TRUE}
\end{array}
$$

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

$$
\begin{array}{l}
\textit{DeleteFails} \triangleq \\
\quad \forall\, op \in \textit{OperationStates} : \\
\quad\quad \text{IF } op.type = \text{“delete”} \\
\quad\quad \text{THEN} \\
\quad\quad\quad \land\; op \in (\textit{activeOps}) \\
\quad\quad\quad \land\; op.step = 1 \\
\quad\quad\quad \land\; list[\textit{SORegularKey}(op.key)] = \textit{NULL} \\
\quad\quad \implies \Diamond\Box(\neg([op \text{ EXCEPT } ![\text{“step”}] = 3] \in (\textit{activeOps}))) \\
\quad\quad \text{ELSE TRUE}
\end{array}
$$

$\textit{OperationStates} \triangleq$

Set of all possible active operations

They can be of type insert, delete or bucket$_i$ nit

Step denotes the current step of the operation

$$
\begin{array}{l}
[type : \{\text{“insert”}\},\ step : 1 \,..\, 4,\ key : \textit{PossibleKeys},\ value : \textit{PossibleValues},\ id : 0 \,..\, \textit{MaxID}] \\
\quad \cup \\
[type : \{\text{“delete”}\},\ step : 1 \,..\, 3,\ key : \textit{PossibleKeys},\ id : 0 \,..\, \textit{MaxID}] \\
\quad \cup \\
[type : \{\text{“bucket\_init”}\},\ step : 1 \,..\, 3,\ bucket : 0 \,..\, (\textit{MaxSize} - 1),\ id : 0 \,..\, \textit{MaxID}]
\end{array}
$$

Figure 4.3: The failure invariants as TLA properties

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.

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 Results

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

Keys	Values	Diameter	Distinct States	Time (hh:mm:ss)	Error
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	
4	16	17	84,587,043	02:54:11	No space left on device

Table 4.1: Model checking results for SplitOrder

concurrent setting, showing this implementation holds – at least for maps with up to 4 keys and 8 values – helps to increase our confidence that is 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.

Invariant	Keys	Values	Conc. Ops	Diameter	Distinct States	Time (mm:ss)	Error
InsertSucceeds	4	2	2	62	1,627,390	02:15	
	4	3	2	27	1,426,888	33:30	Out of memory
DeleteSucceeds	4	2	2	62	1,627,390	00:45	
	4	3	2	39	6,067,795	30:30	GC limit exceeded
BucketsInitialized	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	
	4	2	4	57	120,175,837	26:49	???

Table 4.2: TLC model checking results for SOConcurrent

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 descrepancy 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 *BucektsInitialized* is a safety property, memory is not consumed as quickly as with the earlier liveness properties and *BucektsInitialized* 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.

Invariant	Keys	Values	Conc. Ops	Diameter	Distinct States	Time (hh:mm:ss)	Error
BucketsInitialized	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	
InsertFails	2	2	2	20	64,000	00:01:07	Out of heap space
DeleteFails	2	2	2	25	153,370	00:02:12	GC overhead limit exceeded

Table 4.3: Model checking SOConcurrent with operation IDs

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

Lessons learned

- state space explosion is **real**
- liveness properties tend to quickly exceed Java's GC limit

5

Discussion

In this chapter we first discuss the results of model checking in relation to the thesis: that Shalev et al.'s design is a correct extensible hashmap for concurrent systems and that a model checker can show this.

The latter half of the chapter is concerned with the experiences of writing and testing a specification, and the lessons learned in the process.

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 correctness of the TLC model checker is outside the scope of this text, but the software is open source [13] and has been used in industry with success [2, 14].

On the other hand, the specification is very much within the scope of this project. 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 – both require some human ingenuity. While developing the specification, I often tested properties or invariants in order to check that the specification adhered to expectations. 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 – helped to further refine the specification or correct errors.

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 Implementation Issues aka. "dumb things I did"

In this section I will explain some of the choices made in the development of the specification and discuss the experiences and lessons learned from the process. The first part discusses the non-concurrent specification and its shortcomings, and is followed by a discussion of experiences from the process as a whole.

The first attempt at writing a specification served as a learning tool for the TLA+ language, but did not produce a useful specification for the purpose of proving correctness in a concurrent setting. The structure of the specification was based on the structure of an implementation in that actions correspond to the *insert*, *delete* and *bucket init* functions in Shalev et al.'s pseudocode. Such a structure brings with it the implicit assumption that the functions are linearizable since one action corresponds to one state transition.

The attempt to show correctness through implementation also posed difficulties. The idealized hashmap specification proved impossible to directly map to the split-order specification because a *find* operation in may result in a bucket needing to be initialized and thus can not be used directly in a refinement mapping. Instead we introduced a set of auxiliary variables [15] and maintained an idealized map that was updated in atomic steps to implement the Hashmap specification. The updates loosely correspond to a *find* operation and the correctness of this idealized map does seem to correspond to a correct underlying structure (albeit at an earlier state), but introduces additional complexity to the spec and does not resolve the underlying issue of assumed linearizability.

- it's easy to mis-state what you want to check, and to make subtle spec errors if not careful (e.g my original insert/delete-fails which did always held because the op terminated, and then never held because there was no ID on ops)

- careful with infinities (e.g my infinite set of activeOps)
- sometimes it is hard to tell if a model is doing what you think (use assertions of assumed invariance)
- choose step-size early and clearly
- consider what we want to show and make sure the spec is capable of showing that. (Uniqueness in fail properties)
- formal specification/model checking is probably most useful as a development tool for algorithms and protocols the same way test-driven development is used for implementations.

Bibliography

[1] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th. USA: Prentice Hall Press, 2014. ISBN: 013359162X.

[2] 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.

[3] 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.

[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] 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.

[13] Microsoft. *TLA+ Toolbox*. https://github.com/tlaplus/tlaplus. 2003.

[14] 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.

[15] Leslie Lamport. "Hiding , Refinement , and Auxiliary Variables." In: July (2019).