Faculty of Electrical Engineering,
Mathematics & Computer Science

Verification of distributed locks: a case study

Joël Ledelay
M.Sc. Thesis
June 2023

Supervisors:
Prof. dr. M. Huisman
E. Ruijters (BetterBe)
R.B. Rubbens
Examiner:
Prof. dr. ir. A.L. Varbanescu

Formal Methods and Tools group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

In this thesis, VerCors will be used to verify functional correctness of a lock implementation provided by BetterBe. This implementation is a distributed, reentrant, read-write lock, in which the resource which the lock locks on is a database table row. This implementation has been used in practice and any problems which have previously turned up have been solved. BetterBe is fairly confident that the implementation works as intended, as the implementation has run for an extended amount of time without major defects. In an effort to ensure that no more major defects will appear down the line, BetterBe wishes to employ formal verification to close the gap between the perceived correctness of the system and its actual correctness.

Scenarios which are highly dependent on timing, availability of database services or other specific conditions such as system load or connectivity issues may not have been encountered, but that does not give any guarantee that these scenarios never lead to faults in system behaviour.

This thesis formally verifies the functional correctness of a simplified version of the implementation, and proves properties such as data race freedom. Annotations are added to the program to specify handling of ownership and functional correctness of the program with regards to its expected behaviour. The annotated code is then analysed by VerCors to verify these properties.

Table of Contents

Abstract 2
1 Introduction 8
1.1 Motivation 8
1.2 Formal Verification through VerCors 8
1.3 Research Goal 9
1.4 Overview 9
2 Background 12
2.1 Concurrency 12
2.2 Locks 13
2.3 Reentrancy 13
2.4 Readers-Writers problem 15
2.5 Permission-based Separation Logic 16
2.6 Pre- and Postconditions 18
2.7 Lock Invariants 19
2.8 Predicates 20
3 Related Work 22
3.1 Alternative locking mechanisms 22
3.2 Verification of other lock implementations 22
3.3 Alternate logics for reasoning about lock operations 22
4 Implementation 24
4.1 Named locks 24
4.2 Distributed access 24
4.3 Reentrancy 24
4.4 Read/Write Lock 24
4.5 Obtaining a lock 25
4.6 Releasing a lock 25
4.7 Code structure 25
4.8 FailFastLock optimization 27
4.9 Further implementation details 28
5 Assumptions 30
5.1 No deadlocks 30
5.2 Database reachability 30
5.3 Limited number of threads per DbLockManager 30
5.4 Unique UUIDs 30
6 Methodology 32

6.1 Code transformations 32
6.1.1 Watchdog functionality 32
6.1.2 Enums 32
6.1.3 MutableInt 32
6.1.4 Generics 32
6.1.5 Bounded unique locks 34
6.1.6 Strings 34
6.1.7 ThreadLocal and Map 34
6.1.8 Database support 35
6.1.9 Miscellaneous (minor syntactical changes left out) 38
6.2 Preparing intermediate steps 38
6.2.1 Intermediate 1 39
6.2.2 Intermediate 2 40
6.2.3 Intermediate 3 40
6.2.4 Intermediate 4 40
6.3 Verifying the intermediate steps 41
7 Verification 42
7.1 Intermediate 1 42
7.1.1 AtomicInteger 42
7.1.2 Subject 43
7.1.3 DbNamedLock 43
7.1.4 Pair 48
7.1.5 DbLockManager 49
7.2 Intermediate 2 55
7.2.1 DbNamedReadWriteLock 55
7.2.2 DbNamedLock 56
7.2.3 DbLockManager 60
7.3 Intermediate 3 61
7.3.1 DbRow 61
7.3.2 ListDbRow 61
7.3.3 DbModel 66
7.3.4 DbNamedLock 71
7.3.5 DbNamedReadWriteLock 74
7.3.6 DbLockManager 74
7.4 Intermediate 4 79
7.4.1 ListDbRow 79

7.4.2 DbModel 80
7.4.3 MyReentrantReadWriteLock 80
7.4.4 Pair 81
7.4.5 DbNamedLock 82
7.4.6 DbLockManager 82
8 Limitations 90
8.1 Intermediate 2 90
8.1.1 Usage of the DbNamedReadWriteLock 90
8.1.2 Subject's invariant for read locks 90
8.1.3 Shared permission for DbNamedReadWriteLock's lock type flag 90
8.1.4 Each lock instance has their own Subject 91
8.2 Intermediate 3 91
8.2.1 Usages of database model predicates are incorrectly specified 91
8.2.2 Missing postcondition in DbLockManager.releaseLock() 91
8.2.3 DbLockManager.next_uuid is not unique 91
8.3 Intermediate 4 92
9 Results 94
9.1 Intermediate 1 94
9.2 Intermediate 2 94
9.3 Intermediate 3 94
9.4 Intermediate 4 94
9.5 Miscellaneous 95
9.6 Conclusion 95
10 Bibliography 96
Appendix A Implementation details of abstracted parts 98
Appendix B Overview of changes from Capita Selecta code until Intermediate 3 100

1 Introduction

1.1 Motivation

In a world where many critical systems rely on the correctness of software, it is insufficient to give an intuition on whether a piece of software should work or not. More robust measures are required in order to inspire confidence in the correctness of the program.

Many testing methods exist to give some measure of confidence in the correctness of a program. These testing methods fall into the category of dynamic analysis. Dynamic analysis is a technique where the program is executed with some set of input values. Then, the behaviour and internal state of the system is observed. This is a useful way to collect information to address a particular problem that is encountered [4].

However, traditional testing methods are limited in what they can show. This class of testing can only show the presence of errors in a program, not guarantee the absence of them. Specific test cases may succeed, but that does not give a guarantee that all inputs will lead to a valid result. Furthermore, not all inputs can be tested for, as this is completely infeasible for anything other than the most simple of programs.

Concurrent software is even more difficult to properly test. Their behaviour depends on timing, and while certain interleavings of commands may cause the test to succeed, others cause it to fail. This is exacerbated by certain types of bugs which are difficult to recreate, such as databases being unresponsive, certain processes which happen to not get scheduled often enough by the processor, other threads hanging or crashing, etc.

In order to prove the correctness of the program even in these hard to recreate situations, testing, however rigorous, does not suffice. Creating test cases for all edge cases is not feasible, and furthermore takes significant time and effort to do. There will most likely be edge cases which are overlooked or cannot be reproduced, and then conclusions can still not be drawn about the absence of errors.

1.2 Formal Verification through VerCors

This is where formal verification comes into play. Formal verification is used to reason about the program in a more abstract manner. Formal methods work by reasoning mathematically about models of system behaviour. By doing this, stronger guarantees can be given on system behaviour. One such formal method is deductive verification, which will be used in this thesis.

Deductive verification is a technique through which correctness with respect to a specification is proven. The program does not need to be executed, but instead, mathematical reasoning is used to determine whether a given program conforms to its specification. There has been a lot of research into deductive verification techniques, and many different tools which employ deductive verification techniques exist for many different languages [10].

Many formal verification tools exist, but most of them are concerned mostly with sequential programs. VerCors [1] is a formal verification tool which employs deductive verification to reason about concurrent software, proving properties such as data-race freedom, memory safety and functional correctness. It does this through specifications, which are added to the program as annotations. These annotations specify the expected behaviour of the program. The logic it is built upon has a strong notion of ownership. Variables stored on the heap can only be accessed if the correct permissions are held. More details on how this works is given in Section 2.5.

1.3 Research Goal

In this thesis, VerCors will be used to verify functional correctness of a lock implementation provided by BetterBe[1]. This implementation is a distributed, reentrant, read-write lock, in which the resource which the lock locks on is a database table row. More details on the implementation can be found in Chapter 4. This implementation has been used in practice and any problems which have previously turned up have been solved. BetterBe is fairly confident that the implementation works as intended, as the implementation has run for an extended amount of time without major defects. In an effort to ensure that no more major defects will appear down the line, BetterBe wishes to employ formal verification to close the gap between the perceived correctness of the system and its actual correctness.

Scenarios which are highly dependent on timing, availability of database services or other specific conditions such as system load or connectivity issues may not have been encountered, but that does not give any guarantee that these scenarios never lead to faults in system behaviour.

This thesis formally verifies the functional correctness of a simplified version of the implementation, and proves properties related to data race freedom. A simplified version of the implementation was fully specified and verified, which consists of a distributed, reentrant, read-write lock. An optimization designed to alleviate database load during locking operations was not fully verified, though significant progress was made towards a successful verification.

Annotations are added to the program to specify handling of ownership and functional correctness of the program with regards to its expected behaviour. The annotated code is then analysed by VerCors to verify these properties. More details on the methodology used to perform this research can be found in Chapter 6.

Finally, the scalability of the VerCors toolset is investigated, as VerCors has mostly been used on small examples thus far. The conclusion drawn from this thesis is that VerCors is a toolset which has some hurdles which are yet to be overcome. However, great leaps are being taken by the VerCors team to overcome these hurdles and make applying VerCors to industrially sized projects more and more feasible.

For now, applying VerCors to such projects is only feasible under several assumptions and limitations, and requires some simplification and rewriting of the code in order to be possible at all. However, since this thesis was started, many of the features which were missing are now implemented at least to some extent, so usability of the tool is rapidly improving.

1.4 Overview

This thesis achieves a successful verification of a distributed, reentrant, read-write lock, under some assumptions and with some limitations. The final optimization which was present in the implementation given by BetterBe was not successfully verified, though significant steps were taken towards a successful verification.

Chapter 0 introduces some of the background information for this research, such as basic information on concurrency, locks and variants thereof, as well as some of the theoretical background on how VerCors works. Chapter 4 introduces the implementation, as provided by BetterBe. In Chapter 5, the assumptions that were necessary to conduct this research are given. Chapter 6 discusses the methodology used in this research, including code transformations and intermediate results, and Chapter 7 describes the verification of each of these intermediate results.

[1] Home · BetterBe. – https://www.betterbe.com/ (Accessed: Mar. 29, 2023)

The limitations of the work presented in this thesis are discussed in Chapter 8, and finally, the results of this research are presented in Chapter 9.

2 Background

VerCors is a deductive verification tool developed by the University of Twente [1]. This thesis uses VerCors to write specification for a Java project. VerCors also has support for several other languages, including C, OpenCL and Prototypal Verification Language (PVL), a language designed for use with VerCors, in order to showcase and test its capabilities for verifying concurrent software. All of the features described in this chapter are also supported for PVL, but may have different syntax. All of the VerCors-specific information presented here can also be found on the VerCors wiki, accessible through GitHub [1].

In this chapter, some background is given on concurrency, the issues that arise when concurrency is improperly managed, and some of the constructs existing in Java that are used to manage these problems.

It is key to understand when a concurrent piece of software is written correctly. The techniques used in VerCors to verify the correctness of concurrent software are also presented in this chapter.

2.1 Concurrency

In concurrent software, many parallel processes operate concurrently to achieve a goal. Such processes are called threads in Java. Threads are used to perform multiple tasks at the same time. Threads are very versatile and can be used for a variety of tasks, ranging from performing multiple complex background calculations to responding to user input. In order to synchronize and manage the tasks performed by the threads, communication between threads is crucial.

One way in which such threads communicate with each other is through access to a shared resource, for instance a database or shared memory. These threads can access and update the shared resource, but this must be handled in a proper manner. If multiple threads attempt to read and update the same piece of memory at the same time, faults may occur. For instance, suppose there are two threads. Both threads want to increment the value of this shared variable.

Suppose the starting value of this variable is 5. Then, two possible executions of thread 1, concurrently with thread 2, are shown in Table 1.

Thread 1 reads the current value: 5	Thread 1 reads the current value: 5
Thread 2 reads the current value: 5	Thread 1 writes 5 + 1 = 6 to the variable
Thread 1 writes 5 + 1 = 6 to the variable	Thread 2 reads the current value: 6
Thread 2 writes 5 + 1 = 6 to the variable	Thread 2 writes 6 + 1 = 7 to the variable

Table 1: Two possible interleavings of Threads 1 and 2

In these two possible scenarios, both thread 1 and thread 2 have performed their task, but the result of performing the two operations is non-deterministic. This is called a race condition. Race conditions cause problems, because the result of performing any operation can no longer be guaranteed. This makes it difficult to guarantee anything about the piece of software which the race condition exists in. The automated detection of race conditions and data races has been studied extensively over the years [2, 3, 13].

In the first example, thread 2 is accessing the variable while thread 1 is in the middle of writing to it. This is called a data race. Data races can cause crashes, reading corrupted or incorrect data or even corruption of the resource that’s being written to.

2.2 Locks

In Java, concurrency problems are often handled through the use of locks. Locks protect a piece of shared memory, such that nobody may access the memory without holding the lock.

In Java, locks are represented by the Lock object. It contains (among others) methods to obtain a lock (tryLock, lock) and release a lock (unlock). The tryLock() method attempts to obtain the lock a single time. The lock() method repeatedly calls tryLock() until it succeeds. The unlock() method releases the lock. When the lock is obtained through the tryLock() or lock() methods, the lock is considered to be held. Other threads may no longer obtain the lock, until the lock is unlocked by the current holder.

An example of a situation where concurrent behaviour was potentially problematic was given in Table 1. If a lock were to protect the variable that was being written to, such that each increment action was atomic from the perspective of the other thread, then the leftmost interleaving shown would no longer be possible, and the final result would always be that the variable holds the value 7. There are still multiple execution paths possible (either thread 1 or thread 2 may increment the variable first), but data races may no longer occur.

In Java, there is also another way to obtain exclusive access to an Object. This is the synchronized keyword. An internal lock on a class instance is used by declaring a code block or method as synchronized. This marks the code block or method as a critical section, which only one thread may access at any one time. Prior to entering the synchronized block, an internal lock is automatically obtained, and upon exiting the synchronized block, the lock is released again.

An example of a synchronized block is given in Listing 1. Here, lines 2-4 are a synchronized block which synchronizes on `this`. Upon reaching line 2, the synchronized block is entered and the internal lock on the `this` object is obtained. Then, the count field is incremented. Upon reaching line 4, the synchronized block is released and the lock is released again.

If a thread is already inside a synchronized block, other threads are forced to wait until the synchronized block is left before proceeding.

```
[1]public void incr() {
[2] synchronized (this) {
[3]  count++;
[4] }
[5]}
```

Listing 1: An example of the synchronized keyword

2.3 Reentrancy

Reentrancy refers to the capability of re-obtaining a lock while already holding it. In a normal non-reentrant lock, attempting to obtain a lock which is already held by the thread will result in a deadlock, as the thread will wait until the lock is released before advancing. In a reentrant lock, however, obtaining the lock while already holding it increments an internal counter. When releasing the lock, this counter is decremented. When this count reaches zero, the lock is finally released.

This can be useful in situations where keeping track of whether or not a lock is held is difficult or inconvenient. An example of this is given in Listing 2. Here, we have a class Counter, which has a value field and a ReentrantLock which protects access to the val field. The incr() method obtains the lock (line 7), increments the val field (line 8) and releases the lock again (line 10).

The incrTwice() method must increment the val field twice, without any other threads gaining access to the val field inbetween the two calls of incr(). Because of this, the lock is obtained before the first

incr() call (line 16). Then, the incr() method is entered, obtaining the lock again in line 7. If the lock were not reentrant, this would result in a deadlock. However, because the lock is reentrant, it is now held twice. The value is incremented once (line 8) and the lock is unlocked (line 10). We exit the first incr() call, still holding the lock once, and enter the second incr() call on line 18. Again, the lock is reentrantly obtained (line 7), the val field is incremented (line 8) and the lock is released (line 10). Now, the value of the val field is incremented twice, and we still hold the lock. Finally, the lock is fully released (line 20).

While this example is slightly contrived, this technique may be useful for a variety of applications. These applications include recursive method calls, traversing non-linear data structures and other applications, where it is not immediately clear whether or not the given lock is already held upon entering a critical section.

```
[ 1]class Counter {
[ 2] int val;
[ 3] ReentrantLock lock;
[ 4]
[ 5] void incr() {
[ 6]  try {
[ 7]   lock.lock();
[ 8]   val++;
[ 9]  } finally {
[10]   lock.unlock();
[11]  }
[12] }
[13]
[14] void incrTwice() {
[15]  try {
[16]   lock.lock();
[17]   incr();
[18]   incr();
[19]  } finally {
[20]   lock.unlock();
[21]  }
[22] }
[23]}
```

Listing 2: An example of a ReentrantLock

In Java, the synchronized construct is also reentrant. This means that after entering a synchronized block, one can enter another synchronized block protecting the same resource without ending up in a deadlock. An example of this is given in Listing 3.

```
void method1() {
 // ...
 synchronized(this) {
  // ...
  method2();
  // ...
 }
 // ...
}
void method2() {
 // ...
 synchronized(this) {
  // ...
 }
}
```

Listing 3: Reentrant use of the synchronized construct

If the synchronized construct was non-reentrant, calling method1() would result in a deadlock, as upon entering the body of method2(), the thread would wait until the synchronized block of method1() would be left. To solve this, method2() would have to be inlined into method1()'s definition, which would result in code duplication. Unfortunately, VerCors currently implements Java's locks as non-reentrant locks. This means that the synchronized keyword is also implicitly non-reentrant in VerCors. Therefore, such a reentrant synchronization construct is to be avoided.

Using the synchronized construct is not always an option. While it's a simple way to provide access control, and forgetting to unlock a lock is impossible, some applications require a more fine-grained locking mechanism. For instance, consider the example in Listing 4. When the lock cannot be immediately obtained, the thread is suspended for a period of time before attempting to lock again. A normal synchronized block does not have capabilities to support this, but a custom lock implementation could.

```
public void lock() {
// ...
 while (true) {
  if (tryLock()) return;
  // ...
  Thread.sleep(1000);
  // ...
 }
}
```

Listing 4: An example of a custom lock() implementation

Furthermore, the synchronized keyword only supports a write lock, meaning only one thread is allowed to have the lock at a time. More fine-grained locking mechanisms such as read/write locks, which are described in the next section, are not supported.

2.4 Readers-Writers problem

Suppose there's a piece of shared memory protected by a lock. If there is a high number of accesses for users which need to read from the protected memory, with a comparatively low number of accesses for users who need to write to the protected memory, it may make sense to distinguish between read-only permissions and write permissions when obtaining the lock. Devising a thread-safe manner of doing this is called the readers-writers problem [8]. Several variants of this problem exist, in which the priority of readers and writers is specified. In the implementation discussed in this thesis, the priority used is a weak reader priority [16]. This means that whenever a read lock is acquired, other readers may also obtain the read lock, regardless of any waiting writers. When the last read lock is released, there is no specified priority between readers and writers, allowing either to obtain the lock. This means that writers may encounter some starvation if readers continue to arrive while a read lock is held.

As long as there are only read-only accesses, these accesses may happen concurrently without risk of data races or race conditions. In order to do this in a thread-safe manner, the read lock exists. This lock offers read-only permissions to the protected memory. Any process attempting to access the protected memory must first obtain the read lock.

When there are threads requiring write access, they cannot safely access the resource at the same time as the readers, since such accesses may lead to race conditions. Therefore, a separate type of lock is required, which provides write access to the protected memory. This lock is called a write lock.

Arbitrarily many readers may access the resource concurrently, given that they are holding the appropriate read lock, as long as no writers have the write lock. Alternatively, exactly one writer may hold the write lock, as long as no readers have the read lock.

A lock which supports both read locks and write locks is called a ReadWriteLock. It solves the readers-writer problem.

2.5 Permission-based Separation Logic

PBSL is a logic which allows reasoning about memory at any point in the program. This is done by applying a Hoare-like logic [11], which has the form of `{P}S{Q}`, in which `P` and `Q` are expressions in propositional logic which represent respectively the pre- and postcondition and `S` represents the program statement which is executed. This logic can be used to reason about the results of executing a set of program statements. Applying this technique to blocks of code, such as methods, allows for pre- and postconditions to be created and reasoned about. For instance, consider the following example in Listing 5:

```
int a = 0;
a = a + 3;
assert b > a;
```

Listing 5: A sample set of program statements

Annotations can be added, indicating for each statement which pre- and post-conditions exist. This yields Hoare triples for each program statement. The annotated code is given in Listing 6. Note that the postcondition of each program statement is also the precondition for the next program statement.

```
{b > 3}
int a = 0;
{b > a + 3}
a = a + 3;
{b > a}
assert b > a;
{b > a}
```

Listing 6: A sample set of statements annotated with Hoare-like pre- and postconditions

These Hoare triples represent for each point in the program which pre- and post-conditions must hold for the program to finish execution without errors, while respecting the pre- and post-conditions for each individual statement. This allows us to conclude that given the precondition that `b > 3`, we can conclude that `b > a` after execution of this block of code. Actually executing the code is not necessary, as we inferred this through deductive verification, a technique where the syntax of the program is analysed to infer properties related to its behaviour, without actually running any code.

Note that in the example given in Listing 6, the pre- and postcondition were chosen such that the final assertion could be proven in an optimal manner. Here, optimal means that these pre- and postconditions are chosen such that there are no larger set of initial states which still satisfy each of the pre- and postconditions specified.

The pre- and postconditions could be strengthened (made more restrictive), for instance by requiring `b > 10` instead of `b > 3`. This has the effect of shrinking the set of initial states which satisfy the boolean assertions given in each of the Hoare triples.

This would then yield the most restrictive postcondition of `b > a + 7`. This postcondition can also be weakened without an effect on the correctness of the specification. For instance, consider Listing

7. Here, the precondition has been strengthened from `b > 3` to `b > 10`, and the postcondition has been weakened from `b > a + 7` to `true`. Here, the postcondition has been weakened, 'losing' information about the values of `a` and `b`.

```
{b > 10}
int a = 0;
{b > a + 10}
a = a + 3;
{b > a + 7}
assert b > a;
{true}
```

Listing 7: An example of a strengthened precondition and a weakened postcondition

In PBSL, variables stored on the heap cannot be accessed without the proper permissions. Permissions are represented by fractions between 0 and 1 (inclusive), usually being of the form 1/(2n), with n = 1, 2, ….

In VerCors, this is written Perm(var, frac). These fractions represent the level of access the program has at the current location. A permission of 1 indicates write permission. A permission of 0 indicates no permissions, and any other fraction indicates a read-only permission. Permissions can be split and combined. For instance, a write permission on the variable count can be split up and combined as such:

```
Perm(count, 1) = Perm(count, 1\2) ** Perm(count, 1\2)
```

Here, the $**$ operator is used to combine permissions. This operator is explained in more detail later in this section.

At any point in the code, the permissions that are currently owned are known. VerCors checks that no unauthorized access or update operations are performed through deductive verification of the code.

Through this explicit handling of (fractional) ownership of variables, data race freedom and memory safety are guaranteed, as a consequence of the soundness of the underlying logic [6].

Since many program statements also affect the heap, PBSL must also allow reasoning about (parts of) the heap. Due to the presence of aliasing, this is not entirely trivial. When two variables point to the same heap location, this is called aliasing. This causes logic, which otherwise seems correct, to potentially be problematic. Consider the following example: $(x \mapsto 3) \land (y \mapsto 4)$. At first sight, there does not seem to be any problem with this expression. However, if x and y both map to the same heap location, then this expression can no longer hold, as the heap location which x refers to would have to point to the values 3 and 4 simultaneously.

Classic Separation Logic handles this through the points-to predicate and the $*$-conjunction operator. In VerCors, the notation for this operator is $**$, to prevent possible conflicts with the multiplication operator $*$.

The points-to predicate $e \mapsto v$ describes that the heap contains a location addressed by e, and the value stored at that location is v. This predicate allows reasoning about variables stored on the heap. In VerCors, additionally, permission to access e is specified. This is written PointsTo(e, frac, v), where frac refers to the permission value. This is shorthand notation for Perm(e, frac) $**$ e == v. Recall that a permission value of 1 indicates write permission, where any other 0 < frac < 1 indicates read permission.

In VerCors, a permission value of 1 can equivalently be replaced with the keyword `write`. The keyword `read` can be used to represent an unspecified infinitesimally small fraction greater than 0, which can be split and combined endlessly without ever combining into a full write permission. This is useful for the case where it is not necessary to know exactly what fraction of permission is available, as long as there is enough to be granted read permission.

The $*$-conjunction operator describes that, given an expression $\phi * \theta$, the heap can be split into disjoint parts of the heap satisfying ϕ and θ respectively. $\phi * \theta$ can be read "ϕ holds, and separately θ holds". This allows us to reason about aliasing. For instance, a predicate $a \mapsto _ ** b \mapsto _$ allows us to conclude that a and b are disjoint locations of the heap, therefore a and b are not aliases of each other.

Furthermore, this logic employs local reasoning – In a predicate ϕ, only the parts of the heap directly affecting ϕ are reasoned about. The rest of the heap is implicitly unaffected. This allows us to define an additional rule, the Frame rule [14].

$\frac{\{P\}S\{Q\}}{\{P * R\}S\{Q * R\}}$, where R does not contain any variable that is modified by S.

This rule states that any heap location which is not modified by S is implicitly unaffected. This rule allows us to forego reasoning about the entirety of the heap, but instead allows us to focus only on the parts which are directly affected by the program, also known as the footprint of the program

2.6 Pre- and Postconditions

This Permission-based reasoning is applied through the use of pre- and postconditions, as well as invariants on object state. Pre- and postconditions are discussed in this section. Lock invariants, a special class of invariants, are discussed in Section 2.7.

In the precondition of a method, the permissions which are required to call the method may be specified. For instance, for a method incr() which increments a variable, write permission to that variable is required in order to run the method. In the postcondition, the permissions which are released again can be specified. In the example of incr(), it may make sense to specify write permission in the postcondition of the method. This has the effect of returning the write permission to the caller. If this is not done, the caller no longer has access to the variable, and the permission is lost.

One way this can be useful is in setting a flag indicating whether an object has been initialised, as shown below. After running the init() method, it's no longer possible to write to the initialised variable, but it can still be read from, as read permission is returned by the method. This can be used in other methods to change behaviour based on whether the class was properly instantiated. An example of this is given in Listing 8. Note that the specification is written in comment blocks which start with `/*@` or `//@`. These are treated as comments by the JVM and therefore do not affect the behaviour of the program.

```
/*@
requires Perm(initialised, 1);
ensures Perm(initialised, read);
@*/
void init() {
 //...
 initialised = true;
}
/*@
requires Perm(initialised, read);
@*/
boolean insert() {
 if (!initialised) return false;
 // ...
}
```
Listing 8: Calling init() makes the initialised field read-only

Aside from specifying permission access, pre- and postconditions can also be used to restrict the allowed inputs and specify functional properties about the method. For instance, for a method div(int a, int b), which computes a / b and returns the result, it makes sense to include b != 0 as a precondition to avoid division by 0. Furthermore, in the postcondition, it makes sense to specify \result == a / b.

```
/*@
requires b != 0;
ensures \result == a / b;
@*/
int div(int a, int b) {
 return a / b;
}
```
Listing 9: An implementation of div() with annotations

2.7 Lock Invariants

Java classes may have lock invariants associated with them. In a lock invariant, permissions and boolean assertions regarding the fields of the class can be described. Since a lock invariant relates to permissions on fields of a class, as well as boolean assertions regarding these fields, lock invariants always relate to one instance of a Java class.

The lock invariants must be established at the end of the constructor of the class. After establishing the lock invariant, its contents are no longer available to any caller. Instead, the knowledge contained in the lock invariant (both the permissions and the boolean assertions) are obtained through synchronization.

In Java, this is done through the synchronized keyword. Prior to entering a synchronized block, a lock on the corresponding object is obtained. Upon exiting the synchronized block, this lock is released again. This ensures that at most one thread may be inside of a synchronized block at any time.

Methods may also be synchronized. The behaviour is very similar. Upon encountering a method call on a synchronized method, the lock on the corresponding object is obtained. Only when this lock is obtained, may the method be executed. After execution of the method, the lock is released again.

Inside of synchronized blocks which synchronize on a Java instance, the permissions and boolean assertions inside the corresponding lock invariant are available. Outside of these synchronized blocks, the permissions and assertions within a lock invariant are not available. Since only one thread may be inside a synchronized block on a class instance at any one time, the lock invariant is only ever given out at most once.

An example of the usage of a lock invariant through the synchronized keyword is given in Listing 10 below.

```
//@ lock_invariant Perm(f, write) ** 0 <= f ** f < 10;
class C {
 int f;

 //@ ensures committed(this);
 C() {
  f = 4;
  //@ commit this;
 }

 //@ requires committed(this);
 synchronized void m() {
  //@ assert Perm(f, write) ** 0 <= f ** f < 10;
  f = 7;
 }

 //@ requires committed(this);
 void p() {
  m();
 }
}
```

Listing 10: An example of a lock invariant and its usage

Note that some new syntax is introduced here. Inside of the constructor, the 'commit this' ghost statement is encountered. This statement signals the prover that the lock invariant may be removed from state at this point. The lock invariant is established there. The committed(this) statement indicates that the object is properly instantiated, that is, the lock invariant has previously been committed through the commit this statement.

In order to enter a synchronized block, the object must first be committed. Inside of the synchronized block, such as in method m, the permissions and assertions given in the lock invariant are available. Method p is not inside of a synchronized block, but it contains a synchronized method call. Therefore, it also has the requirement of committed(this). Only inside of the method call for m are the permissions and assertions in the lock invariant available.

2.8 Predicates

Predicates in VerCors are a syntactical construct which allows the user to bundle permissions and properties related to these permissions. Among other things, this is useful for defining properties of recursive datastructures, such as a LinkedList. Consider the following example in Listing 11. Here, a predicate is used to specify that the list is strictly increasing, with a set starting value.

```
public class ListNode {
 int value;
 ListNode next;

 /*@
 resource increasing(int i) = Perm(next, write) **
  Perm(value, write) ** value >= i **
  (next != null ==> next.increasing(value + 1));
 */

 // ...
}
```

Listing 11: A predicate defined for a LinkedList

Here, the predicate `increasing(int i)` contains permissions regarding the next and value fields, and furthermore indicates that the value field must be at least `i`. Furthermore, the value of all nodes after this node must be higher than the value of this node.

Predicates can also be used to bundle commonly used permissions and properties into a short hand form. For instance, if the expression

```
Perm(variable, 1\2) ** lower <= variable ** variable < upper
```

is used in multiple places in the program, with different values for lower and upper, it makes sense to define a predicate to abbreviate them, as such:

```
inline resource checkBounds(int low, int high) =
Perm(variable, 1\2) ** low <= variable ** variable < high;
```

Then, at any place where this expression is needed, it can be replaced with checkBounds(lower, upper), with the appropriate values for lower and upper substituted in. This improves readability and maintainability of the specification.

Note that the predicate checkBounds had the keywords 'inline resource' before it. The resource keyword indicates that what follows is a predicate definition. The inline keyword indicates that wherever this predicate is encountered, it can be automatically inlined and replaced with the contents of the predicate.

Not all predicates are necessarily inline predicates. When the 'inline' keyword is missing, the predicate is opaque. An opaque predicate can be 'folded' to hide part of the state in exchange for an instance of the predicate. The same predicate can then be 'unfolded' to obtain the information contained within.

Opaque predicates can be likened to a voucher for ice cream. As long as you have the voucher, you do not have access to ice cream, but you know that it would be possible to obtain ice cream. The voucher can be traded for ice cream, resulting in you having ice cream, but no voucher. This is similar to the process of unfolding an opaque predicate. An instance of the predicate (the voucher) is traded for the permissions and boolean assertions contained in the predicate (the ice cream).

Another analogy is a labelled container, in which information is stored. When the container is closed, the information inside is hidden, but the container has a label on the lid. When looking for a specific piece of information, the container is closed, so this information cannot be found.

However, if we open the container, then the information inside is visible, but the container's label is no longer visible. In this case, when looking for a piece of information, the inside of this container is also inspected to see if the information is there. This is similar to unfolding an opaque predicate.

The reverse is also possible in this analogy. The information can be put back inside of the container, and the container can be closed. Now, the information is hidden again, but the label on the container's lid is once again visible. This is similar to folding an opaque predicate.

In this thesis, opaque predicates are used to limit the state space which is searched when proving or disproving certain properties of the program. Opaque predicates are used in this research to improve performance of the solver.

3 Related Work

3.1 Alternative locking mechanisms

Winblad, Sagonas and Jonsson [19] developed a search tree which helps minimize lock contention by altering the structure of the tree at run-time, depending on heuristics such as lock contention and performance metrics for range operations. This data structure helps adjust to differing contention scenarios. In contrast, this thesis applies a different optimization to minimize the time spent on lock contention, namely the failFastLock optimization, described in Section 4.8. While this does not directly prevent lock contention, it does lower the impact of failing lock operations.

3.2 Verification of other lock implementations

Bultan, Yu and Can [7] presented a method to verify synchronization behaviour in programs which employ reentrant locks. They replace the lock interface in question with a finite state machine which represents the interface's behaviour and subsequently use this finite state machine to draw conclusions on the behaviour of the thread which accesses the lock. Our approach instead specifies the behaviour directly in the source code of the Lock interface, more clearly showing the link between code and specified behaviour.

Shirako, Vrvilo, Mercer, Sarkar [15] presented an alternative read/write lock which promises improved performance as compared to Java's ReentrantReadWriteLock implementation from java.util.concurrent. They used Java Pathfinder (JPF) model checker [18] to verify the correctness of their implementation. This thesis also verifies a ReentrantReadWriteLock implementation, though it is a similar one to the implementation from java.util.concurrent. VerCors is used to verify the correctness of the implementation.

Van Gastel [17] presented a verification of a reentrant read/write lock, based on Nokia's QT framework. In their verification effort, they found a deadlock and a possible starvation. An alternative version was developed which solves these problems. This improved version was then verified using the SPIN model checker [5]. This thesis presents the verification of a distributed reentrant read/write lock, where different lock instances may be aliases of each other. This implementation is verified using the VerCors toolset.

3.3 Alternate logics for reasoning about lock operations

Gotsman, Berdine, Cook, Rinetzky and Sagiv [9] presented a logic which allows reasoning about an unbounded number of locks and threads, through support of the fork/join parallelism. Locks can be created and deleted at runtime arbitrarily, which helps build the link to real-life scenarios, where locks are not all initialised at the start of runtime, but instead are initialised as needed. However, they also use lock initialisation, locking and unlocking as its base building blocks. In contrast to that, this thesis shows that a specific lock implementation is valid, on top of the built-in synchronized keyword, which is assumed to work.

Naik and Aiken [12] presented a method to correlate lock operations with the guarded memory locations. They work under the assumption that two separate lock instances must also guard different memory locations. They provide formal semantics for reasoning about heap locations and how to prevent aliasing of such locations. VerCors handles this through PBSL, discussed in Section 2.5. Furthermore, in the implementation discussed in this thesis, multiple lock objects may actually alias. If a thread holds a lock on one of the objects, the other object may be used by the same thread to unlock the same lock.

4 Implementation

This thesis discusses a case study regarding the verification of an implementation of a named, distributed, reentrant, read/write lock provided by BetterBe.

This chapter introduces the concepts above and explains how they were originally implemented. Furthermore, the structure of the project is given. This implementation forms the basis for the code which is studied in this thesis. The structure presented here was slightly altered for compatibility reasons. The specific transformations that were applied and the methodology in performing this research further are discussed in Chapter 6.

4.1 Named locks

Multiple locks with different names may exist independently of each other. These locks protect mutually exclusive parts of shared memory, without overlap. Therefore, the operations to obtain and release a lock are independent between locks with different names. Each lock instance has a field lockName which stores an identifier to distinguish between locks with different names.

Multiple lock instances may exist with the same lock name. These lock instances are considered to be aliases of each other and protect the same memory locations.

Each lock instance furthermore has a unique identifier (UUID) which is used to uniquely identify a lock instance. This UUID is used for identifying the client that owns a given lock. This is further discussed in the Section 4.2.

4.2 Distributed access

The system has a central database with a lock table. This lock table contains information on all currently held locks. In order to obtain a lock, a lock row is inserted into this table. This is a database row containing information pertaining to the lock, such as lock name, lock type and the lock instance’s UUID. Each lock row corresponds to one held lock. When a lock row is deleted, the corresponding lock is considered to be released.

The lock implementation is distributed. This means that clients from different threads, machines or even physical locations may obtain and release the same set of locks. The synchronization of these locks occurs through the lock database.

4.3 Reentrancy

The lock implementation is reentrant (see Section 2.3), meaning that a client holding a lock may re-obtain the lock. This results in a lock that is held multiple times. This can be useful for situations where keeping track of whether or not a lock is held is difficult or expensive. An example of such a scenario is given in Section 2.3.

Access to the protected resource is only given the first time the lock is obtained, and this access is only revoked when the lock is released exactly as many times as it was obtained. Therefore, multiple unlock() operations may be necessary to fully unlock a lock.

4.4 Read/Write Lock

A distinction is made between read locks and write locks. When a write lock is obtained, exclusive access is given to the protected resource. No other clients may obtain the read or write lock while the write lock is held. When a read lock is obtained, shared (read-only) access is given to the protected resource. Other threads may concurrently obtain the read lock and obtain shared access, however, the write lock cannot be obtained until all read locks are released. A more detailed explanation of this can be found in Section 2.4.

This mechanism is implemented in the database. Prior to an insert operation, it is checked whether the insertion is legal. That is, when a write lock row is to be inserted, it is checked that no other lock rows with this lock name exist. If this condition is satisfied, the row is inserted. Otherwise, no row is inserted. Prior to the insertion of a read lock row, it is checked that no write lock row exists with the same lock name. Again, if this condition is satisfied, the row is inserted. Otherwise, the insertion is cancelled. This has the effect of preventing read and write locks to occur simultaneously. In this thesis, these conditions are referred to as read/write exclusivity.

In this implementation, there is no explicit fairness policy implemented. When a lock is released and threads are waiting to obtain the lock, it is not explicitly specified which waiting thread gets precedence. Therefore, the next thread to obtain the lock is considered to be randomly selected.

However, if a read lock is randomly obtained, all other threads which are waiting to obtain a read lock may do so while the read lock is held. If there are many waiting readers at any one time, this may result in starvation for the writers. Therefore, this implementation has a weak precedence for readers[16].

In the usage scenario for the given implementation, contention is unlikely, since read access mostly occurs during office hours, whereas write operations mostly occur during the night.

4.5 Obtaining a lock

A lock is considered to be obtained when a new row is successfully added to the locking database. Such a row contains the lock name and type of lock (read or write), as well as the lock instance's UUID to be able to later delete the correct database row upon releasing a lock. Prior to insertion, the locking database then checks whether this lock would violate the read/write exclusivity. If so, the insertion is rejected and the lock is not obtained. If read/write exclusivity is respected if this lock is given out, the row is inserted and the lock is obtained successfully. This operation is atomic on the database level.

There are two methods which can be used to obtain a lock on an existing lock instance. The tryLock() method attempts to obtain the lock once, and returns whether the lock was successfully obtained or not. The lock() method repeatedly calls the tryLock() method, until it succeeds in obtaining the lock.

4.6 Releasing a lock

In order to release a lock, it suffices to delete the database row corresponding to the lock. This row is identified through its UUID. If such a lock row exists, it is removed. Upon the final release, the permissions to access the protected resource are considered to be revoked. As the correct usage of the lock is not checked by the implementation, this is assumed to be the case. Later in this thesis, this access is actually revoked through the use of the specification.

4.7 Code structure

The code is structured into several classes. A class diagram of the system is given in Figure 1. Each of these classes will be briefly touched upon in this section.

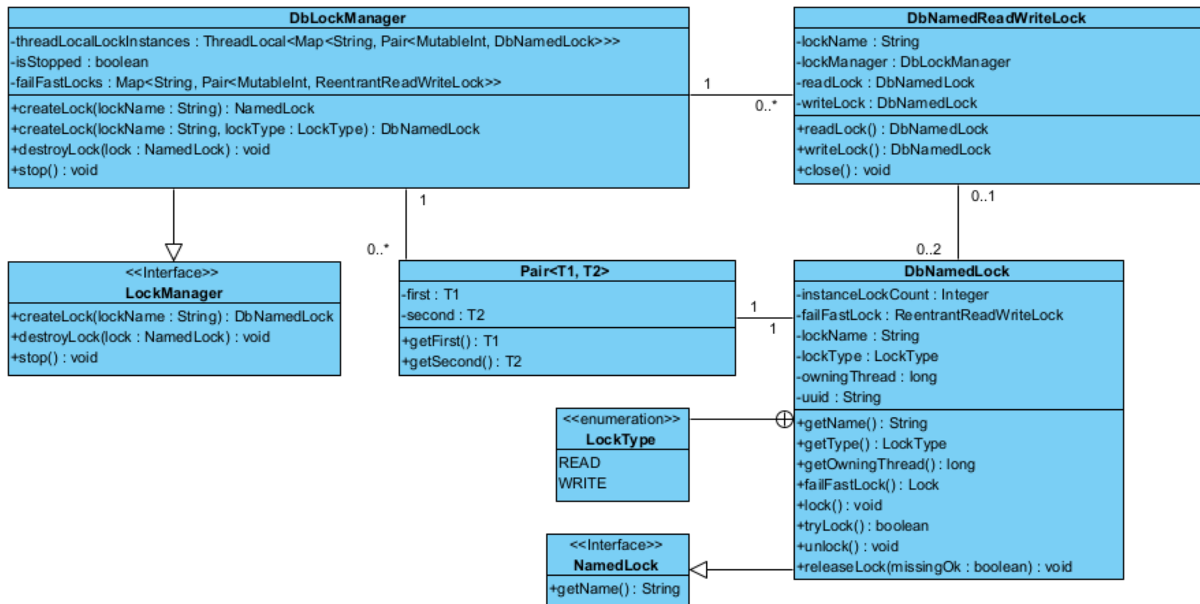

Figure 1: Class Diagram of the implementation

The DbNamedLock class represents a thread-local instance of a lock with a given lock name and lock type. It has methods to obtain and release the lock and interfaces with the database to perform these operations, as described earlier in this chapter. It implements the NamedLock interface. Many such DbNamedLock instances may exist, though within a thread, only one DbNamedLock instance with a given name and lock type exists. This is handled by the DbLockManager class.

The DbNamedReadWriteLock class represents a ReadWriteLock. It contains one readLock and one writeLock, which are both DbNamedLocks. It can be used as if it were a ReentrantReadWriteLock, but it is implemented through the use of the DbNamedLock class for its read- and write locks.

The DbLockManager class is responsible for creating and deleting DbNamedLock instances. It keeps track of the different instances that are in existence within the current Java Virtual Machine (JVM). This creates an overview of which DbNamedLock instances are in existence, and allows for some optimizations which are described in the section below. Furthermore, as these instances are created as necessary, this saves memory, as not all possible names must have DbNamedLock instances associated with them at all times.

When a client wishes to obtain a DbNamedLock instance, the createLock() method is called. This method returns a reference to a DbNamedLock instance with the given name. Furthermore, a count is stored of how many times this specific lock instance was given out. Subsequent calls to createLock() return the same lock instance and increment this count. When the instance is destroyed through destroyLock(), this count is decremented. When the count reaches 0, this means that the lock instance is no longer in use. The reference that is stored in the DbLockManager instance is deleted, allowing the garbage collector to destroy the associated DbNamedLock object. This does add the expectation that any thread calling createLock() will eventually also call destroyLock() equally many times. This allows lock instances to be destroyed when no more references to the instance exists and prevents memory leakage.

The DbNamedLock instances are thread-local. This means that different threads calling createLock will receive different lock instances. This removes the requirement for DbNamedLock to be thread-safe, since only one thread will ever call its methods.

These lock instances are stored in a ThreadLocal<Map<String, Pair<MutableInt, DbNamedLock>>>. The Pair class, as its name implies, simply stores a pair of its first argument type and its second argument type. It has methods to obtain the first and second arguments. In this case, the first type is a MutableInt, which stores the number of existing references to the lock instance. The second type is DbNamedLock. This contains the reference to the lock instance.

These Pairs are stored in a Map<String, Pair<…>>. The key of this map is a String, which is constructed as a combination of lock name and lock type. This means that there's a unique entry for each combination of lock name and lock type within this Map.

Finally, these Maps are stored in a ThreadLocal<Map<…>>. This ensures that each thread obtains a different Map instance. Therefore, DbNamedLock instances are not shared between threads.

4.8 FailFastLock optimization

In Figure 1, in the DbLockManager class, there's one more field of note, namely the failFastLocks field. It is a Map<String, Pair<MutableInt, ReentrantReadWriteLock>>. This field is necessary for an optimization called the failFastLock optimization. The failFastLocks field is named such, because when an attempt to lock it fails, it fails quickly, when compared to a database request. Below, the reasoning and behaviour of this optimization is described.

Consider the following scenario. A DbLockManager currently has two active DbNamedLock instances which share a lock name. One lock is a write lock, the other a read lock. Suppose one of these two locks is held. Then, the other DbNamedLock instance will never perform a successful tryLock() call, as read locks and write locks cannot be held at the same time.

As the tryLock() operation is certain to fail, it is not necessary to poll the database to obtain this result. Since database operations are considered to be expensive in terms of performance, it is preferable that such operations, which are certain to fail, are avoided.

In order to detect such a scenario, the failFastLock was introduced. There is a single ReentrantReadWriteLock for each lock name. This ReentrantReadWriteLock instance is shared between DbNamedLock instances within the DbLockManager. Within a DbLockManager, one failFastLock exists per lock name, and two DbNamedLock instances exist per thread (one read and one write lock).

As with the DbNamedLock instances, the failFastLock instance is stored in a Map<String, Pair<MutableInt, ReentrantReadWriteLock>>. The Pair holds the lock instance and its associated count (keeping track of how many references to this instance exist), and these Pairs are stored in a Map, with the lock's name as its key, and the Pair as its value.

When constructing a DbNamedLock instance, an instance of this failFastLock is created if necessary and passed along to the DbNamedLock. This way, each DbNamedLock instance will have an associated failFastLock instance.

Inside the DbNamedLock class, there's a method failFastLock(), which returns the failFastLock's read resp. write lock, depending on the lock type of the DbNamedLock instance.

The failFastLock must be locked, before a DbNamedLock.tryLock() call may contact the database. If this failFastLock.tryLock() call fails, another thread within this DbLockManager has already obtained an incompatible lock (a read or write lock when this DbNamedLock instance is a write lock, or a write lock if this DbNamedLock instance is a read lock). In this case, the database is certain to also have a row representing this lock, and the DbNamedLock.tryLock() call may exit with a negative result.

If the failFastLock.tryLock() call succeeds, then within this DbLockManager, no other incompatible lock is currently held. An attempt is made to insert the new lock row into the database. This may still fail if an incompatible lock is held, but the incompatible lock will be held by a DbNamedLock instance which belongs to a different DbLockManager instance.

4.9 Further implementation details

For brevity, some parts of the implementation are left out here. These parts are abstracted away or simplified for the purpose of verification. These parts are the database and the watchdog functionality.

The database is replaced by a model of the database's behaviour. A further description of the exact functionality of this database model is discussed in Section 6.1.8.

In the given implementation, a timeout system was in place. This timeout system was designed to prevent deadlocks from occurring. When a lock is obtained, the lock will 'time out' after a specific period of inactivity. The watchdog functionality is responsible for refreshing this inactivity period, ensuring that an active lock will not be released under normal circumstances.

The reasoning behind the made abstractions and a more detailed description of these parts can be found in Section 6.1 and Appendix A.

5 Assumptions

A number of assumptions were made over the course of this thesis, which were necessary to conduct this research. These assumptions are listed below.

5.1 No deadlocks

All locks will eventually be released. This assumption, while simple, has far-reaching effects. Since all locks are eventually released, events that cause abnormal termination of a thread, such as power outages or crashes, can be assumed not to happen before the held lock is released. Therefore, there is no longer a need for a timeout system that is supposed to prevent deadlocks upon such abnormal circumstances. This timeout system releases inactive locks, that is, locks that haven't been accessed in a specific period of time.

Recall that in Section 4.9, the watchdog functionality was briefly introduced. This functionality was responsible for refreshing the lock, such that it does not time out. Because of the assumption made here, this functionality can be left out of the system to be verified.

This assumption simplifies verification effort greatly, as it removes all need to explicitly handle timing-related issues. Furthermore, this assumption removes the complications that arise when large write operations are halted mid-way through.

In the usage scenario of this system, these complications sometimes cause the protected resource to end up in an invalid state, where data was partially, but not completely, overwritten. This was deemed a known and acceptable limitation to the system. Therefore, it could be left out of the verification effort. If this assumption was not made, the watchdog functionality had to be proven to be correct. However, since there is a known scenario where the behaviour of the system is incorrect (mixed read/write access is possible in rare situations), this would not be possible. This situation is detailed in Appendix A.

5.2 Database reachability

The database is always reachable. If the database were to be unreachable, no locks can be locked or unlocked by definition, as locks never time out (Section 5.1), and an unreachable database cannot handle insertion or deletion requests.

Therefore, this assumption can be compared to the situation where all distributed clients are not performing database requests for the duration of any database outage. Held locks are still considered to be held.

This behaviour is simpler to model with a database which is always reachable, while not impacting the behaviour because of the justification given above.

5.3 Limited number of threads per DbLockManager

No more than T threads operate on a single lock manager at the same time. This is a minor assumption, since there can be arbitrarily many lock managers in existence at any one time. Furthermore, BetterBe does not have a use case where the number of threads active under any one lock manager keeps growing without limit, so this assumption has no far-reaching effect for the validity of the results.

5.4 Unique UUIDs

The UUIDs generated for the lock instances are unique. These UUIDs were first mentioned in Section 4.2. While there is admittedly a probability for them to overlap, this probability is so astronomically small that it can be ignored for the purpose of verification.

6 Methodology

6.1 Code transformations

A number of steps had to be taken to prepare the program for verification, since some of the syntactical constructs used in the given program were not supported by VerCors, and some constructs were no longer necessary due to the assumptions made in Chapter 5. Therefore, the code had to be transformed. The transformations described in this chapter were applied manually, and are separate from the transformations applied by the VerCors toolset during verification.

Below, the removed constructs are enumerated, and it is described how these constructs were transformed to work with VerCors.

6.1.1 Watchdog functionality

The watchdog functionality, which was briefly mentioned in Section 4.9 and explained in detail in Appendix A, was removed. This could be done because of the assumption made in Section 5.1, that locks will always eventually be released.

6.1.2 Enums

In the code, one Enum was specified for the lock's type. This type was either READ or WRITE. Enums were not supported in VerCors, so this construct had to be refactored. Since there were two options, they could be encoded into a boolean field named lockType, in which false represents a read lock and true represents a write lock.

6.1.3 MutableInt

This project used the MutableInt class to keep track of how many references to a given lock instance were given out. This was described in Section 4.7. Furthermore, the lock instances used the Integer class to keep track of the reentrancy level of the lock.

These usages of MutableInt and Integer were refactored into usages of the AtomicInteger class. The reason this is needed is that the AtomicInteger class is thread-safe. This was an important requirement for intermediate parts of the research, which are described in Section 6.2. In particular, since lock instances are shared between threads in the earlier intermediates, these lock instances required a thread-safe implementation of the Integer class.

Since the MutableInt class shows identical behaviour to the AtomicInteger class with regards to single-threaded scenarios, this was a reasonable replacement to make, even in the parts of the research that did not explicitly require a thread-safe implementation. This simplified verification, as there was already an existing specification for the AtomicInteger class, and no new specification had to be written for the MutableInt class.

The specification and full description of the AtomicInteger class is given in Section 7.1.1.

6.1.4 Generics

In Java, a class can be made generic by giving it a generic type signature. This allows the class to be used with different types as its parameters, without any additional programming effort. In the code, this was used in the Pair<T1, T2> class, which represents a tuple (T1, T2) with types T1 and T2 respectively. The original code of the Pair class can be found in Listing 12.

```java
public class Pair<T1, T2> {
 private final T1 first;
 private final T2 second;

 public Pair(T1 first, T2 second) {
  this.first = first;
  this.second = second;
 }

 public T1 getFirst() {
  return first;
 }

 public T2 getSecond() {
  return second;
 }
}
```

Listing 12: The Pair class using generics

Generics are not supported in VerCors, so a workaround had to be found for this. The original Pair class had two usages, namely Pair<MutableInt, ReentrantReadWriteLock> and Pair<MutableInt, DbNamedLock>. The Pair class received one field for each of these types. This eliminated the generics from the class.

```java
public class Pair {
 private final AtomicInteger first;
 private final DbNamedLock dbNamedLock;
 private final MyReentrantReadWriteLock failFastLock;

 public Pair(AtomicInteger first, DbNamedLock dbNamedLock) {
  this.first = first;
  this.dbNamedLock = dbNamedLock;
  this.failFastLock = null;
 }

 public Pair(AtomicInteger first, MyReentrantReadWriteLock failFastLock) {
  this.first = first;
  this.dbNamedLock = null;
  this.failFastLock = failFastLock;
 }

 public AtomicInteger getFirst() {
  return first;
 }

 public DbNamedLock getDbNamedLock() {
  return dbNamedLock;
 }

 public MyReentrantReadWriteLock getFailFastLock() {
  return failFastLock;
 }
}
```

Listing 13: The Pair class after transformation, without generics

It was implemented such that there was one constructor for the case of Pair<AtomicInteger, ReentrantReadWriteLock> and one constructor for Pair<AtomicInteger, DbNamedLock>. The other field was set to contain a null value. The code for the Pair class is shown in Listing 13. Note that the MutableInt class was also replaced with the AtomicInteger class, as mentioned in Section 6.1.3.

6.1.5 Bounded unique locks

In the given implementation, no bound was given for the number of unique lock names. Without loss of generality, an upper bound of maxLockName was added. Since this number can be arbitrarily large (up to 2^{30}), this does not provide an actual limitation, as a user of this system will be unlikely to have quite that many unique locks in use in their systems anyway. Adding this bound allows for the following two abstractions: The removal of Strings and the removal of the Map construct.

6.1.6 Strings

Strings were not supported in VerCors at the time of writing this thesis. Since Strings were used extensively in the given implementation for error logging, lock names, UUIDs and more, a major refactoring of the code was needed.

First of all, all the strings in the error logging were removed. This did not change the functional behaviour of the program, since all field access inside the error logging was read-only. Therefore, removing the statements using error logging could be done safely.

Lock names were refactored into being integers. This is semantically equivalent, as some of the most important characteristics are maintained, namely that unique inputs are considered to not be equal, identical inputs are considered to be equal, and there are many possible unique inputs. One possible encoding to achieve this conversion is to call .hashCode() on each String object. In practice, in the test suite provided alongside the implementation, only 7 unique inputs were given, so each unique String input was encoded as a unique integer lock name.

Without loss of generality, an upper bound for the amount of unique lock names used in the system is given (see Section 6.1.5). This upper bound is stored in a variable maxLockName in the DbLockManager class. The associated locks do not need to be instantiated upon startup, but are instead created and destroyed as needed through the createLock() and destroyLock() methods of the DbLockManager class, which were described in Section 4.7.

The UUIDs, which were mentioned in Section 4.2, were stored as Strings in the given implementation. The purpose of these UUIDs is to provide a randomly generated universally unique identifier, which is used to identify the client to the database. In the transformed version of the code, an integer was used to store the UUID. For this purpose, an integer is equally good, under the assumption that such a unique integer can be generated, and no more than roughly 4 million lock instances exist at any one time, which is a reasonable assumption to make under normal circumstances. This assumption is also discussed in Section 5.4.

Finally, the other usages of the String class were all in the context of the database functionality, which is also not supported in VerCors. Database support is discussed in Section 6.1.8.

6.1.7 ThreadLocal and Map

In the code, lock instances were stored in a ThreadLocal<Map<String, Pair<MutableInt, DbNamedLock>>>. An example of how this construct was stored and how it was used is shown in Listing 14.

```
private ThreadLocal<Map<String, Pair<MutableInt, DbNamedLock>>>
threadLocalLockInstances;

...

String lockKey = lockName + "-" + lockType;
Map<String, Pair<MutableInt, DbNamedLock>> lockInstances =
threadLocalLockInstances.get();
Pair<MutableInt, DbNamedLock> lockPair = lockInstances.get(lockKey);
```

Listing 14: ThreadLocal construct

Since both ThreadLocal<T> and Map<U, V> were not supported, these constructs had to be replaced. They were both replaced with fixed-size arrays. This was possible since an upper limit for both the number of threads per DbLockManager (see Section 5.3) and the number of unique named locks (see Section 6.1.5) was assumed. The array representing the Map contained the Pair objects, and was therefore named pairs. As mentioned in Section 6.1.6, the lockName field was made an integer. As there are two entries for each lock name (one for the write lock and another for the read lock), the pairs array has size 2 * maxLockName. This allows us to use a combination of the lockName and the lockType to index into the pairs array. The first half of the pairs array is reserved for write locks and the second half of the pairs array is reserved for read locks. This allows us to index into this array using the expression `lockName + (lockType ? 0 : maxLockName)`.

The array representing the ThreadLocal construct contained these pairs arrays. Since multi-dimensional arrays were not supported in VerCors, this was done through an inner class LockInstanceArray, which stored the pairs array. The final code that replaced the ThreadLocal construct is given in Listing 15.

```
private final LockInstanceArray[] lockInstanceArray = new
 LockInstanceArray[num_threads];

...

Pair lockPair = lockInstanceArray[current_thread].pairs[lockName +
 (lockType ? 0 : maxLockName)];

...

class LockInstanceArray {

 final Pair[] pairs;

 LockInstanceArray(int maxLockName) {
  this.pairs = new Pair[2 * maxLockName];
 }
}
```

Listing 15: Construct after performing transformation

6.1.8 Database support

In VerCors, no support existed for external database connections. A (Java) database model was created which emulated some of the functionality of a database, such as inserting and deleting rows. The relation between the actual database behaviour and the behaviour of this model are explained at the end of this section. A class diagram of the database model is given in Figure 2.

A database row contains three fields: The lock's name, the lock's type and the caller's UUID. In the given implementation, there were more fields stored in each database row, but as they had no

further semantic meaning in the target program, aside from the watchdog functionality, they were removed for simplicity. The watchdog functionality was removed, as described in Section 4.9.

Such a database row is inserted into a list of database rows. ListDbRow represents an ArrayList<DbRow>. Since generics are not supported (see Section 6.1.4), its type parameter is hardcoded into the class. Its implementation contains methods to insert, delete and retrieve elements. An helper method arraycopy() was added, with behaviour analogous to that of System.arraycopy().

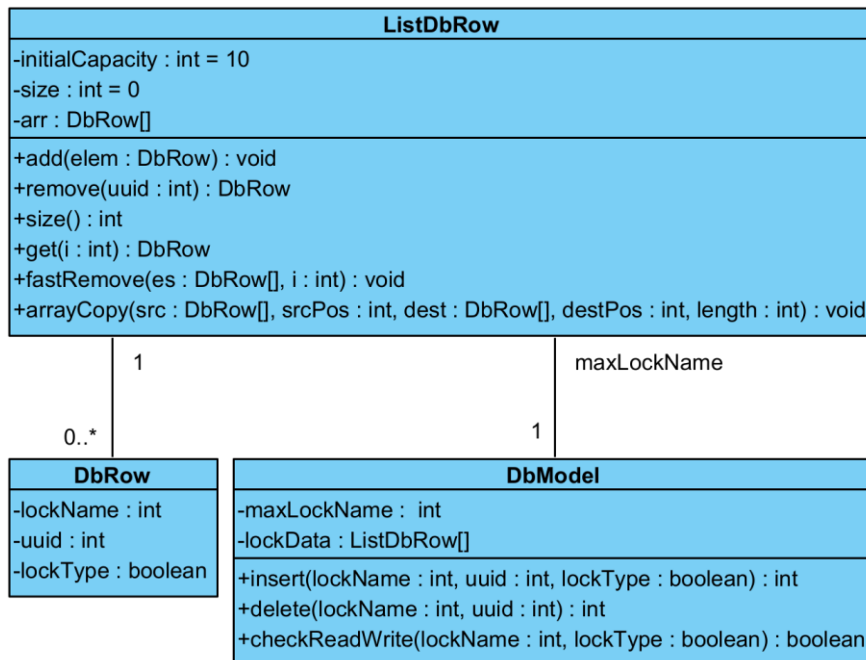

Figure 2: A class diagram of the database model

The database model has the expected insert and delete methods, to insert new lock rows or delete them. Additionally, it has a method checkReadWrite() to check whether, for a new lock row, the read/write exclusivity rules are respected. When a write lock is requested, it is granted only if there are no other read or write lock rows for the given lock name. When a read lock is requested, it is granted only if there are no write lock rows for the given lock name.

The database model is structured such that exactly one ListDbRow object exists for each possible lock name. This makes it easier to check whether the read/write exclusivity rules are respected by a row to insert and simplifies the specification, because ghost variables can be created which keep track, for each lock name, whether it's held and whether it's held in read mode or write mode.

6.1.8.1 Justification for choice of database model

The database model is a sound representation of the functionality of the real database used in the given implementation. This claim is backed by a number of observations on database behaviour and the database model's representation of this behaviour.

The database is reachable from all lock instances, across all lock managers. This behaviour is mirrored by having one single DbModel instance which is shared by all DbLockManager instances (and their associated DbNamedLock instances).

The database is assumed to always be online. This behaviour is mirrored by the database model, which is always reachable.

Only the queries and behaviour relevant to this implementation need to match real database behaviour. These queries are listed below, with a description of their corresponding database model equivalents.

- DELETE FROM lockTable WHERE name = ? AND last_modified < ?
  This query deletes all locks which were outdated (and therefore lost). Under the assumption made in Section 5.1 that all locks will be eventually unlocked, there will never be rows which are matched by this DELETE statement, and therefore, it can safely be abstracted away. This is done in the database model.
- SELECT 1 FROM lockTable WHERE name = ? FOR UPDATE
  In the original implementation, this statement occurred within a transaction together with the INSERT statement below. Its purpose is to lock all other rows with this lock name, such that threads may not simultaneously insert a new lock row, as insertion of a lock row is dependent on the state of other lock rows with the same lock name.
  This is built into the database model by making all database-related methods synchronized, ensuring that only one thread at a time may call any database method. This is not a one-to-one translation, but the database model has a stricter access policy: In the initial implementation, it was possible for two insertions for different lock names to happen simultaneously, whereas in the solution presented in the database model, these insertions also happen sequentially. While this is less performant, this does not introduce any data races or other unwanted behaviour. This is because locks with separate names are considered to have completely separate state. Therefore, this is sufficient to represent the functionality of the real database.
- INSERT INTO lockTable (name, uuid, type, last_modified, created, cluster_host, thread_id, thread_name) VALUES (?, ?, ?, ?, ?, ?, ?, ?)
  In the database model, most of the columns which aren't directly used in the implementation are left out for brevity. This leaves the name, uuid and type columns. This does not change the behaviour. The assumption made in Section 5.1 removes the need for the created and the last_modified columns, and the other columns are also not needed, since the cluster_host, thread_id and thread_name columns are mostly used for debugging situations that led to errors.

  The database furthermore contains a trigger expression upon insertion of a new row. This trigger expression ensures that mixed read and write access is not possible and the rules presented in Section 2.4 are respected.
  This trigger expression is modelled into the database model through a helper method checkReadWrite, which checks that the new row to insert will not violate the read/write exclusivity rules. When calling the insert method, first, this checkReadWrite method is called to check this insertion is legal. Only if this is the case may the insertion continue. This is consistent with the behaviour of the trigger expression, since the checkReadWrite method is called from a synchronized method, and therefore can be considered to part of the atomic operation of insertion.
  Upon completion of the insertion operation, an integer is returned. This integer represents the number of rows inserted. In the database model, the insert method returns 1 on a successful insertion and 0 otherwise. This is consistent with the behaviour of the actual database under the above query.
- DELETE FROM lockTable WHERE uuid = ?

Upon completion of this operation, an integer representing the number of rows deleted is returned. Considering that the UUID is unique across all lock names, this can be represented by returning 1 upon success and 0 upon failure.
In the database model, the delete method not only requires a UUID parameter, but also the lock's name. As this query is only run upon destroying a lock instance, the lock's name is also known to the caller, so this is no big issue.
The lock name parameter is added, because this allows us to more easily specify the results of calling the delete method. Furthermore, this allows us to organise the data contained within the database model in a convenient manner, such that each lock name has a separate ListDbRow associated with it, as mentioned earlier in this section.

All database methods are made to be synchronized, such that only one thread may access the database at a time. This mimics the database property that all queries are run sequentially, and ensures that data races may not occur.

All the above observations lead to the conclusion that the database model is a sufficient model for real database operation for this particular implementation and usage thereof.

6.1.9 Miscellaneous (minor syntactical changes left out)

Other minor syntactical changes are mostly left out for brevity. This included, but was not limited to:

- Removing @Override keywords, since they interfered with the specification keyword "@".
- Transforming static method calls, such that they are explicitly imported:
  ```
  Thread.sleep(ms);
  ```
  becomes
  ```
  import static Thread.sleep();
  ...
  sleep(ms);
  ```

6.2 Preparing intermediate steps

As the implementation which was to be studied and verified was large and complex, the decision was made to split the implementation into four intermediate steps. These steps were constructed by removing parts of the implementation which add complexity, such that a smaller and less complex program could be verified.

Doing this allowed for the specification to initially be made as simple as possible. Verification times are therefore also shorter for the earlier intermediate steps, and a simple verification outline could be created and incrementally improved upon as the later intermediate steps are reached.

These intermediates were constructed in order of decreasing complexity. At each step, compatibility with VerCors was checked by running the tool through the Parsing and Name Resolution phases of the tool. The further verification was done in order of increasing complexity.

For each of these intermediates, a git branch was created. First, the implementation was transformed, as described in the previous section. This code was stored in the branch intermediate-4. Then, this code was cloned to the intermediate-3 branch, and the failFastLock functionality was taken out. For the intermediate-2 branch, the database model was taken out, and lock instances were now shared between threads, to allow locking to be dependent only on the lock instance, rather than the state of the lock manager. Finally, for the intermediate-1 branch, the read/write functionality was taken out, leaving us with a reentrant lock.

Below, the features and challenges of each intermediate are described. Chapter 7 describes the specification for each of the intermediates. A full overview of all specification and implementation changes made until intermediate 3 can be found in Appendix B.

6.2.1 Intermediate 1

Intermediate 1 is a reentrant lock, using a similar code structure to BetterBe's implementation. Since a reentrant lock specification was already made as part of a prior student project by the author, the main challenge of this intermediate was to adapt that specification to match the new code structure.

An overview of the structure of this intermediate is shown in Figure 3.

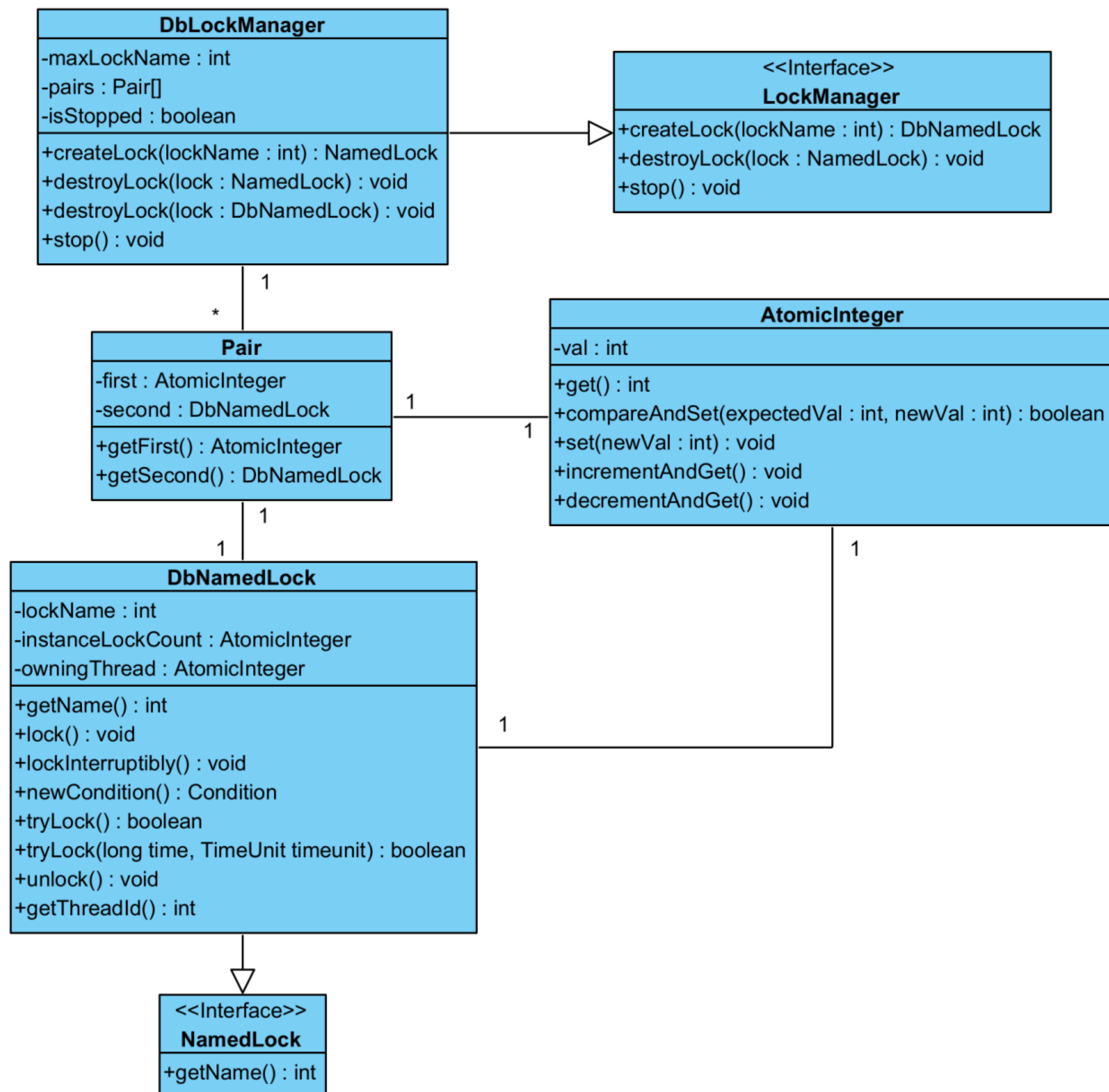

Figure 3: Class diagram of intermediate 1

The DbLockManager is responsible for keeping track of which lock instances are actively being used. For each lock instance in existence, a reference count to the instance is tracked. This is done through the Pair class, which holds a reference to a lock instance and an AtomicInteger, which keeps track of the number references given out to the lock instance.

In the existing specification of the reentrant lock, all locks and all threads had to be instantiated prior to the first locking/unlocking operation. In intermediate 1, a DbLockManager exists, which handles lock creation and deletion as requested. Therefore, the restriction on lock creation before system use, which was present in the student project, was lifted.

Additionally, the concept of named locks was introduced here. Different lock instances with the same lock name are considered to be the same lock in this system. In intermediate 1, this is only possible in some limited manner, namely through a shared reference to the same object, but this is expanded upon in intermediate 3, where the database model is introduced.

When calling createLock(), a reference to the existing lock instance is given, and its corresponding count is incremented. If no lock instance existed, it is created.

When calling destroyLock(), the count corresponding to the lock instance is decremented. When this count reaches 0, the Pair holding the lock instance and its corresponding count is set to null, so it can be subsequently cleaned up by the garbage collector.

In the implementation given by BetterBe, obtaining and releasing locks is handled by the database, as described in Sections 4.5 and 4.6. However, since intermediate 1 does not contain any database functionality, the lock implementation of intermediate 1 is shared between threads, and different lock instances do not interface with each other.

These changes required some alterations to the specification created for the student project to allow for this. The specification of this intermediate is described in Section 7.1.

6.2.2 Intermediate 2

Intermediate 2 adds back the read/write functionality, which was first described in Section 2.4. Each lock is held in either read mode or write mode. The main challenge here is that the DbNamedReadWriteLock class is added, which holds two DbNamedLock objects: a read lock and a write lock. These locks must synchronize their lock state with each other, as when the read lock is held, the write lock cannot be locked at the same time and vice versa. However, when the read lock is held, other threads may also obtain the same read lock, as read access is shared. This means that the tryLock() and unlock() operations of the DbNamedLock class must be altered significantly, as their behaviour differ based on the presence or absence of a linked DbNamedReadWriteLock object. The specification of this intermediate is described in Section 7.2.

6.2.3 Intermediate 3

Intermediate 3 adds the database model. This is the most complex step to add back in, as the other lock classes (DbNamedLock and DbNamedReadWriteLock) must also be altered significantly. These classes no longer synchronize their lock operations on fields of a lock instance, but instead synchronize on the existence or absence of database rows. This means that the structure of tryLock() and unlock() operations must be significantly altered. The newly added database model must have a complete specification. Upon attempting to insert a row, it should be possible to reason about the result of this insertion based on the current state of the database model. Furthermore, specification was needed for a List class, which stores the database rows. The specification of this intermediate is described in Section 7.3.

6.2.4 Intermediate 4

Intermediate 4 adds the failFastLock optimization, which was discussed in Section 4.8. This step adds an additional step to the locking process, which is designed to alleviate the number of redundant database requests. Prior to attempting to insert a row into the database, an attempt is made to lock

a local lock, called the failFastLock. This lock keeps track, for a given lock name, whether this lock is held in read or write mode by one of the lock instances connected to a lock manager. Upon releasing the database lock, the failFastLock is also released.

Suppose that a database lock is held in read mode. Then, the failFastLock is also read locked. When another thread, which has a reference to the same DbLockManager, requests a write lock, it must first call tryLock() on the failFastLock. This will fail, as read locks and write locks cannot be held at the same time, and another thread within the DbLockManager already holds an incompatible lock. It is therefore unnecessary to contact the database to request this database lock. Since database requests are expensive in terms of performance, this optimization saves resources.

This failFastLock is local to the DbLockManager. This means that if there are two DbLockManager objects, the corresponding failFastLock objects don't share any state between them. When a lock instance of the first DbLockManager is holding a read lock and a lock instance of the second DbLockManager attempts to obtain a write lock, the failFastLock of the second DbLockManager instance will not return false (as it would for a similar call from the first DbLockManager).

The main challenge in this intermediate step is to integrate the failFastLock optimization with the rest of the system. This introduces additional complexity, as there may exist up to two lock instances (a read lock and a write lock) which have an effect on the failFastLock's state. The specification of this intermediate is described in Section 7.4.

6.3 Verifying the intermediate steps

The first three intermediates were successfully verified. The most important specification and implementation details of each intermediate step are described in Chapter 7. The full list of changes made from each intermediate to the next (excluding intermediate 4) can be found in Appendix B.

Intermediate 4 was not successfully verified, though significant progress towards a successful verification were made. The specification of this intermediate is described in Section 7.4, and the progress made and steps left to take are described in Section 8.3.

7 Verification

7.1 Intermediate 1

As mentioned in Section 6.2.1, intermediate 1 consists of a reentrant lock. Figure 3 on page 39 shows a class diagram containing the structure of this intermediate. As this is the first intermediate, an overview of the different classes and their specification is given.

7.1.1 AtomicInteger

AtomicInteger is a class with an internal integer value, which can be retrieved and updated through its methods:

- get() – to obtain the current value
- set(int newVal) – To set a new value
- compareAndSet(int expected, int newVal) – To set a new value, if the current value is equal to the expected value
- incrementAndGet() – To increment the current value and get the new value
- decrementAndGet() – To decrement the current value and get the new value.

For brevity, only two methods and their specification are shown here. The full specification details can be found in Appendix B.

```
/*@
 requires committed(this);
 context Perm(val, write);
 ensures val == newVal;
@*/
public synchronized void set(int newVal) {
 this.val = newVal;
}
```

Listing 16: The AtomicInteger's set() method

Listing 16 shows the implementation and specification for the set method. Note that all specification constructs are contained within a special comment block, indicated by the /*@ ... @*/ notation. This tells VerCors that the contents of this block are to be treated as specification constructs.

The requires clause indicates that the object must be committed. This indicates that the AtomicInteger instance is fully instantiated.

The context clause, which is short for ‘requires and ensures’, holds the permissions that are required for this method to operate, as well as the permissions that are given out after this method call terminates. In this case, write permission for the val field is required, as this field is written to in the method body.

The ensures clause indicates the postcondition for the method. After terminating, the new value of the val field will equal the newVal parameter which was passed along.

Listing 17 shows the implementation and specification of the compareAndSet method. Again, as before, the requires and context clauses are identical to those of the set method. The same permissions are needed.

The ensures clauses here indicate two branches. In the first branch, expectedVal is equal to the value of val, prior to this method call. In this case, true is returned and the value is set to be equal to newVal. In the second branch, this condition is not met. Then, false is returned, and the value is untouched.

```
/*@
 requires committed(this);
 context Perm(val, write);
 ensures expectedVal == \old(val) ==>  \result && val == newVal;
 ensures expectedVal != \old(val) ==> !\result && val == \old(val);
@*/
public synchronized boolean compareAndSet(int expectedVal, int newVal) {
 if (val == expectedVal) {
  val = newVal;
  return true;
 }
 return false;
}
```

Listing 17: The AtomicInteger's compareAndSet method

Similarly to this, the other methods were specified. For brevity, their specification is only found in Appendix B.

7.1.2 Subject

The Subject class represents the resource which is protected by the lock. It has a predicate inv(), which represents the permissions given out, when the owner of the lock has access to the protected resource. The inv() predicate is unspecified in VerCors. This allows us to use it within the pre- and postconditions of other methods, without explicitly mentioning what the permissions regarding the protected resource are exactly. The full specification and implementation of the Subject class is found in Listing 18.

```
public class Subject {
 //@ resource inv();

 //@ ensures inv();
 public Subject() {
  //@ assume false;
 }
}
```

Listing 18: The Subject class

Note that the postcondition of the constructor indicates that inv() is returned. Establishing these (unspecified) permissions is done through the assume false statement inside the constructor's body. This statement effectively means that anything can be proven from this point onwards. Since the constructor is verified in isolation, this means that the postcondition can be assumed to be established from that point onward.

7.1.3 DbNamedLock

The DbNamedLock class represents a lock object. In intermediate 1, these lock objects are shared between threads, as mentioned in Section 6.2.1. This means that the implementation should be thread-safe, unlike the implementation given by BetterBe, which does not need to be thread-safe, due to the thread-local nature of the Lock objects (see Section 4.7). Because of this, the implementation of the methods in this class is different from that of the final result.

7.1.3.1 Fields

The DbNamedLock class contains the following fields:

- final AtomicInteger instanceLockCount – An AtomicInteger which keeps track of the reentrancy level of this lock
- final int lockName – An integer representing the lock's name

- final AtomicInteger owningThread – An AtomicInteger which keeps track of the thread which is the current owner of the lock
- ghost int T – A ghost variable representing the maximum thread ID. It is assumed that all thread identifiers are between 0 and this number (exclusive), see also Section 5.3.
- ghost int holder – A ghost variable which keeps track of the current owner of the thread, analogous to owningThread.
- ghost int heldArray[] – A ghost variable which keeps track of the reentrancy level of this lock, for each thread identifier. If a thread with identifier x is not holding the lock, heldArray[x] will be equal to 0. If it is, heldArray[x] will be greater than 0.
- ghost Subject subject – A ghost variable representing the resource protected by the lock. Its inv() predicate is given out when the lock is first obtained, and reclaimed when the lock is fully released.

One might think that the holder and heldArray fields are unnecessary, considering the existence of the owningThread and instanceLockCount fields. However, this is not the case. The reasoning behind it will be explained, as we go through the predicates of the DbNamedLock class.

7.1.3.2 Predicates

```
[1]inline resource lockset_part(int current_thread) =
[2] 0 <= current_thread ** current_thread < T **
[3] Perm({:heldArray[current_thread]:}, 1\2) **
[4] ({:heldArray[current_thread]:} > 0 ==>
[5]  PointsTo({:instanceLockCount.val:}, 1\2, heldArray[current_thread]) **
[6]  PointsTo({:owningThread.val:}, 1\2, current_thread));
```

Listing 19: The lockset_part predicate

In Listing 19, the predicate lockset_part is shown. A half permission to access heldArray[current_thread] is specified in line 3. Note that this permission can be obtained regardless of whether the lock is held, because other threads don't need access to this array element, and the other half is held by the lock invariant. When heldArray[current_thread] > 0, the lock is held by this thread. This is reflected by the PointsTo expressions in lines 4-6, which indicate permission to access instanceLockCount.val and owningThread.val, and specify their exact values. Note that when the lock is not held by this thread, this knowledge is not available, and the permissions to access these fields is not given.

```
[ 1] lock_invariant
[ 2]   Value(T) ** T > 0 **
[ 3]   Value(heldArray) ** heldArray != null ** heldArray.length == T **
[ 4]   Value(subject) **
[ 5]   Perm(instanceLockCount.val, 1\2) **
[ 6]   Perm(owningThread.val, 1\2) **
[ 7]   (instanceLockCount.val == 0 ==> subject.inv() **
[ 8]     Perm(instanceLockCount.val, 1\2) **
[ 9]     Perm(owningThread.val, 1\2)) **
[10]   Perm(holder, 1) ** -1 <= holder ** holder < T **
[11]   (holder == -1) == (instanceLockCount.val == 0) **
[12]   (\forall* int i = 0 .. T;
[13]     Perm({:heldArray[i]:}, 1\2) **
[14]     (i != holder ==> {:heldArray[i]:} == 0) **
[15]     {:heldArray[i]:} >= 0 **
[16]     ({:heldArray[i]:} == 0 ==> owningThread.val != i)
[17]   );
```

Listing 20: The lock invariant of the DbNamedLock class

The lock invariant is shown in Listing 20.

In lines 5 and 6, 1\2 permissions are specified for instanceLockCount.val and owningThread.val. Recall that a full 1 permission grants the holder of these permissions write access to the heap location the permission is held over, and any permission value below that (but above 0) grants the holder of these permissions read access.

In lines 7-9, the other half of these permissions are held, if the lock is not held. Recall that when the lock is held, the other half of these permissions is specified in the lockset_part predicate. This ensures that when the lock is not held, it can be obtained by any thread which entered a synchronized block and therefore has access to the contents of the lock invariant. When the lock is held, just entering a synchronized block is not enough to access these permissions. Instead, the lockset_part predicate must be held, and heldArray[current_thread] must be larger than 0. This happens if the lock is held.

A full write permission for the holder field is specified in line 10. This ensures that the holder field can only be altered when holding the lock invariant. Line 11 indicates that when the lock is not held, the value of holder must be -1. Line 14 specifies that at most one element of heldArray[i] can be larger than 0. Finally, line 16 indicates that any thread for which heldArray[i] is 0 cannot be the lock holder.

The lock invariant and the lockset_part predicate combined ensure that only the holder of the lock has write access to the instanceLockCount.val and owningThread.val fields. Furthermore, read permission for these fields are only available to other threads when inside a synchronized block (and therefore holding the lock invariant).

Finally, there are also two other predicates: common() and initialised(int N). These predicates specify permissions to access the different fields, and specify that they are properly initialised.

7.1.3.3 The tryLock method

Figure 3 on page 39 showed the methods that are in the DbNamedLock class. For brevity, only the tryLock() and unlock() methods are described here.

A flowchart of the tryLock() method is given in Figure 4. Note that all of the operations and the check for whether the lock is held are performed inside synchronized blocks. This ensures that the knowledge inside the lock invariant is available.

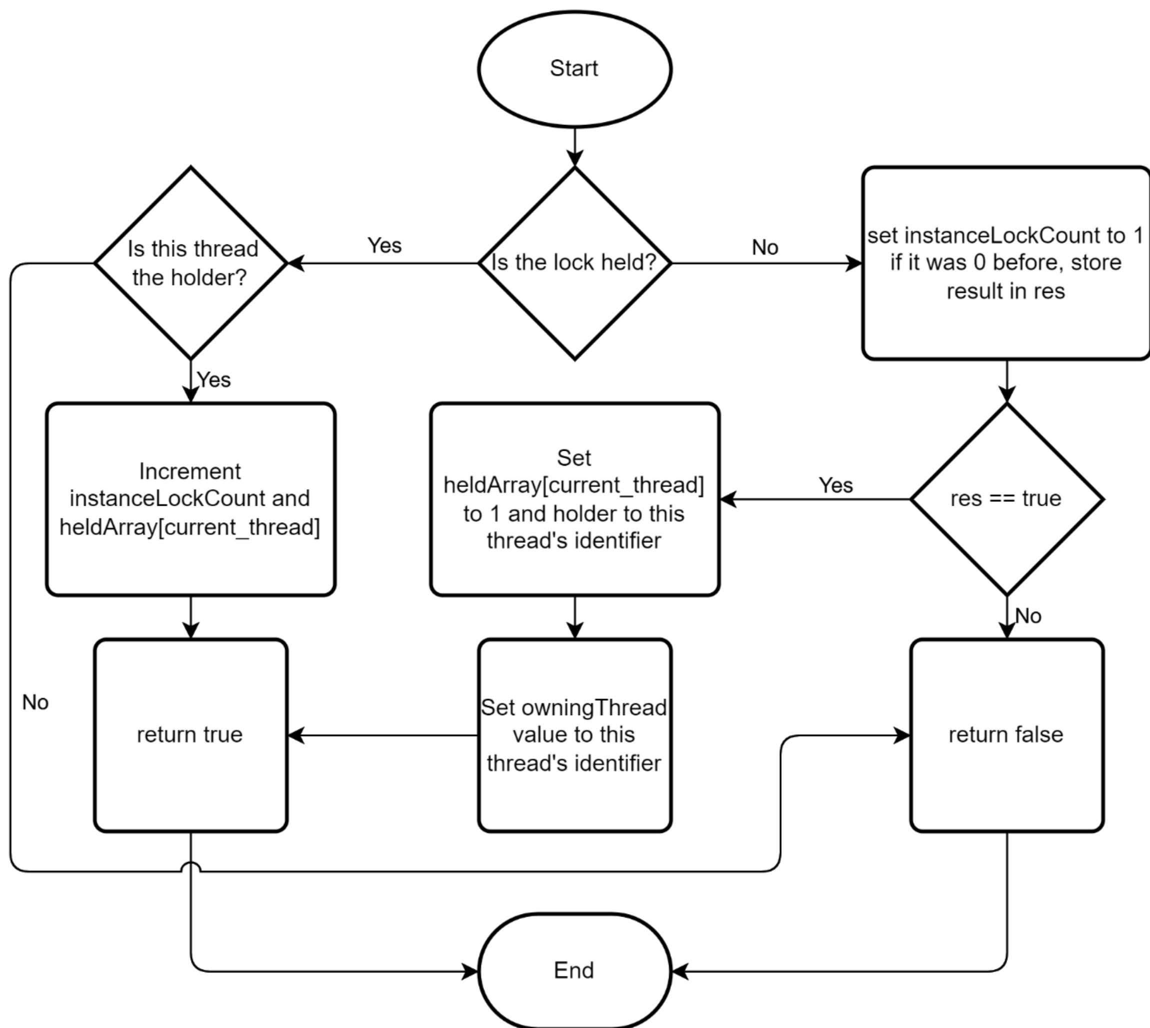

Figure 4: A flowchart model of the tryLock method

The specification for the tryLock method is given in Listing 21. Here, the identifier of the calling thread is passed along as an additional ghost parameter. The ensures clauses here indicate what happens when a positive or negative result is encountered, as well as what state ensures a positive result. Furthermore, the inv() predicate of the Subject class is given out if this lock is locked for the first time.

```
/*@
   given int current_thread;
   context common() ** lockset_part(current_thread);
   context initialised(T);
   ensures (!\result ==> heldArray[current_thread] ==
\old(heldArray[current_thread]));
   ensures ( \result ==> heldArray[current_thread] ==
\old(heldArray[current_thread]) + 1);
   ensures (\old(heldArray[current_thread]) >  0             ==> \result);
   ensures (\old(heldArray[current_thread]) == 0 && \result ==>
subject.inv());
 @*/
public boolean tryLock() {
 // ...
}
```

Listing 21: The specification for the tryLock method

Note that no specification is given which indicates the result of a tryLock() call when the lock is not held by this thread. This is because specifying such a result requires permissions to access fields which are not available at the point where the specification is checked. In particular, any one of these fields must be accessed to be able to specify whether a positive or negative result is reached:

- holder – This allows us to check whether another thread is holding the lock. If so, false is returned.
- All elements of heldArray – This allows us to check whether any other threads are holding the lock. If so, false is returned.
- owningThread – This allows us to check whether another thread is holding the lock. If so, false is returned.
- lockInstanceCount – This allows us to check whether another thread is holding the lock. If so, false is returned.

The permission to access all of these fields are contained inside the lock invariant. At the point of checking the pre- and postcondition for these methods, the lock invariant is not held, so these permissions are not available to the prover.

Even if these permissions were to be available outside of the lock invariant, the lock invariant still needs to be held to ensure that no other thread could obtain the lock in-between checking the pre- and postconditions and them being established. This is not possible in the prover, and therefore, precisely specifying the result of this operation is impossible to achieve with the current setup of permissions and not done here.

7.1.3.4 The unlock method

A flowchart model of the unlock method is given in Figure 5. As with the tryLock method, each of the operations is performed within synchronized blocks, except for the Exception throwing code.

The specification for the unlock method is given in Listing 22. The first three clauses are identical to the clauses in the specification for the tryLock method. The precondition specifies on line 4 that the lock must be held in order to call unlock. The clause on line 5 indicates that when the lock is to be released for the final time, the permissions to access the protected resource must also be given out again. Finally, the postcondition on line 6 indicates that the lock's reentrancy level will be reduced by one after returning from this method.

The signals clause indicates that an IllegalStateException could be thrown, and that nothing is proven about the state if this happens. In the implementation, this Exception is thrown when the owningThread.get() != current_thread, or when instanceLockCount.get() == 0. However, the precondition in line 4 excludes both of these options, because lockset_part() is held (see Section 7.1.3.2). Since heldArray[current_thread] > 0, we get that instanceLockCount.get() == heldArray[current_thread], and owningThread.get() == current_thread.

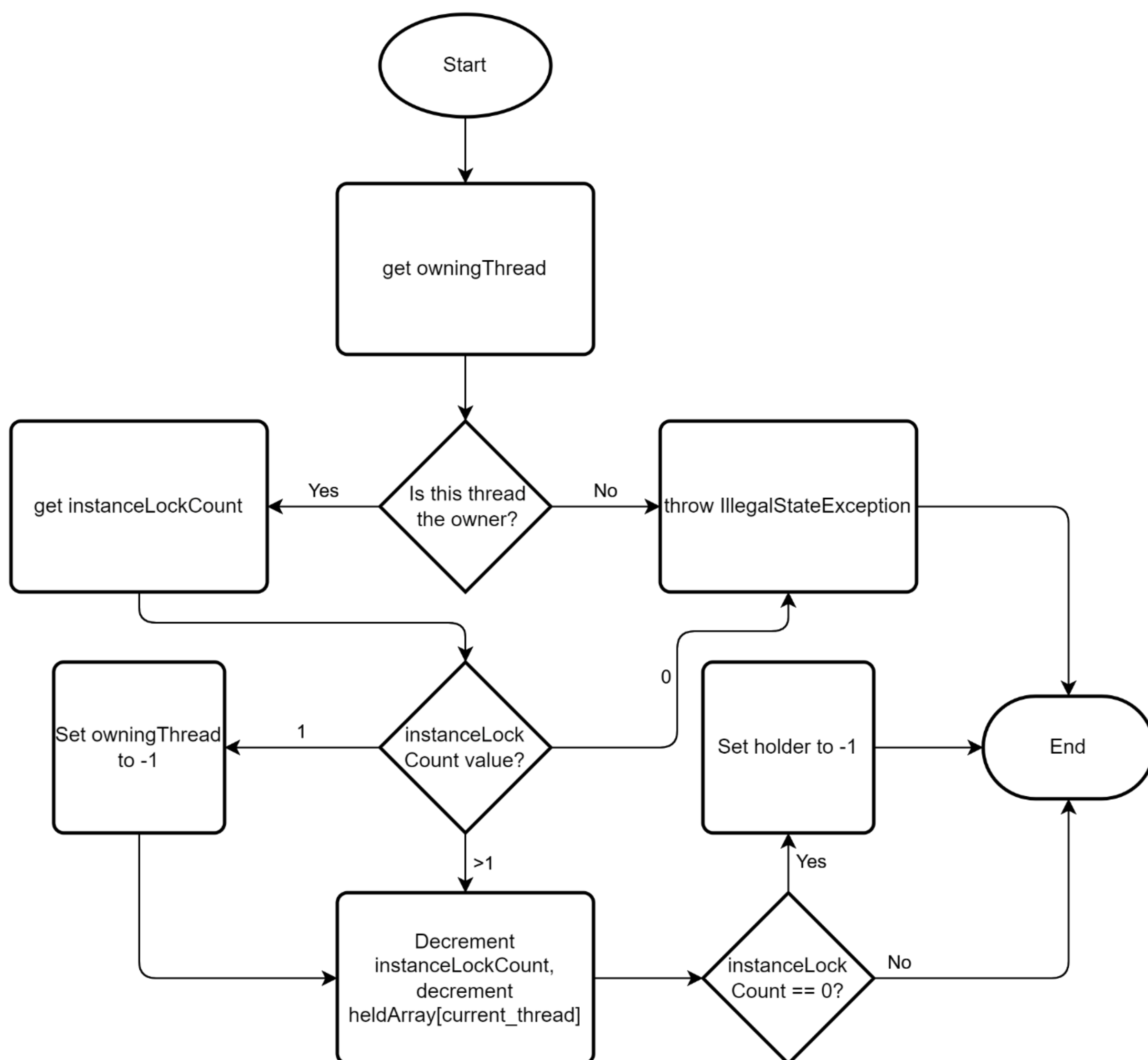

Figure 5: A flowchart model of the unlock method

```
/*@
[1] given int current_thread;
[2] context  common() ** lockset_part(current_thread);
[3] context  initialised(T);
[4] requires heldArray[current_thread] > 0;
[5] requires heldArray[current_thread] == 1 ==> subject.inv();
[6] ensures heldArray[current_thread] == \old(heldArray[current_thread]) -
1;
[7] signals  (IllegalStateException e) true;
 @*/
```
Listing 22: The specification for the unlock method

7.1.4 Pair

The Pair class is an immutable construct which contains two final fields:

- AtomicInteger first
- DbNamedLock second

Methods exist to retrieve each of these fields, and a constructor exists to initialise these fields with given parameters. Listing 23 shows the specification and implementation of this constructor.

Note that the requires clauses refer to the given parameters, whereas the ensures clauses refer to the state of the fields. Each of the permissions and predicates mentioned in the precondition are also present in the postcondition, though quantified over the fields, rather than the parameters.

Note that the common() and initialised() predicates on the DbNamedLock indicate valid state, as well as required permissions, as mentioned in Section 7.1.3.2.

```
/*@
 requires Perm(first.val, write) ** second.common() **
second.initialised(second.T);
 requires committed(first) ** committed(second);
 ensures  Perm(this.first.val, write) ** this.second.common() **
this.second.initialised(this.second.T);
 ensures  this.getFirst() == first ** this.getSecond() == second;
 ensures  this.getFirst().val == \old(first.val);
 ensures  committed(this.first) ** committed(this.second);
 @*/
public Pair(AtomicInteger first, DbNamedLock second) {
 this.first = first;
 this.second = second;
}
```

Listing 23: The constructor of the Pair class

7.1.5 DbLockManager

The DbLockManager class (introduced in Section 4.7 on page 25) is responsible for keeping track of the DbNamedLocks in existence. It does this by holding references to all existing DbNamedLock references and keeping track of how many times they are given out.

When a client wishes to acquire a lock, the createLock() method is called, which creates a DbNamedLock instance if it did not already exist, and returns a reference to the instance. The associated reference count is incremented.

When a client no longer needs access to a lock instance, destroyLock() is called. This signals the DbLockManager that one less client holds a reference to the lock instance, so the associated reference count is decremented. When this count reaches 0, the reference held by the DbLockManager is released, allowing the garbage collector to destroy the DbNamedLock object.

A DbLockManager instance can be stopped by calling the stop() method. This disallows further calls to createLock(), but still allows clients to call destroyLock(). This is done through a boolean flag. This method is fairly trivial in terms of specification and implementation, so the specification thereof is left out for brevity.

Below, a description of the fields, predicates and createLock and destroyLock methods of the DbLockManager class is given, as well as the specification for these methods.

7.1.5.1 Fields

- ghost int T – The maximum number of Threads (and maximum thread id) which interface with this DbLockManager
- final int maxLockName – The upper bound for the allowed lock names that are allowed. The allowed lock names are 0 to maxLockName (exclusive).

- final Pair[] pairs – This array stores the references to the lock instances, using the Pair class described in Section 7.1.4. Recall that the Pair class contains both a reference to an AtomicInteger and a reference to a DbNamedLock instance.
- boolean isStopped – This boolean keeps track of whether or not this DbLockManager is in its stopped state. When a DbLockManager is in its stopped state, locks can no longer be created, but they can still be destroyed.

7.1.5.2 Predicates

The following (inline) predicates are in DbLockManager:

- common() – This predicate indicates valid state for this DbLockManager and holds 1\2 permission for each element of the pairs array, as well as write permission over the isStopped field. In particular, it indicates that pairs is non-null and has length maxLockName, T is a valid number and each non-null element of pairs is correctly initialised (has committed first and second, which are non-null and unique with regards to other elements of the pairs array).
- pureSupport(int N) – This predicate indicates injectivity over the fields of all non-null elements of the pairs array (first and second). Furthermore, it indicates the common() predicate for each non-null pair, which holds write permission for pair[i].first.val.

7.1.5.3 Methods

As mentioned before, the methods createLock() and destroyLock() handle creation and destruction of lock instances. The stop() method sets the isStopped field to true, disallowing creation of new locks.

As mentioned in Section 6.2.1, lock instances have to be shared across threads to work properly in intermediate 1, as there is no synchronizing method between lock instances with identical names.

Figure 6 shows a flowchart model of the createLock method. If the isStopped flag is set to true, creating references to lock instances is no longer allowed, and an IllegalStateException is thrown. If no lock instance exists yet for the requested lockName, a new lock instance is created. The pair (whether it existed prior to the method call or not) has its associated count incremented, and a reference to the lock instance is returned.

Listing 24 shows the specification for the createLock method. The context clauses indicate valid state. The ensures clauses indicate that after calling this method, a lock instance will exist with the name lockName. If this lock instance already existed, it is unaltered, and its associated count will be incremented, or set to 1 if it did not yet exist. The lock instance is returned.

Finally, the signals clause indicates that prior to calling this method, isStopped was set to true, and therefore, creating a lock was not allowed from that state.

```
  /*@
  context common();
  context pureSupport(this.T);
  context 0 <= lockName && lockName < maxLockName;
  ensures pairs[lockName] != null;
  ensures \old(pairs[lockName]) != null ==>
   pairs[lockName] == \old(pairs[lockName]);
  ensures \old(pairs[lockName]) == null ==>
    pairs[lockName].getFirst().val == 1;
  ensures \old(pairs[lockName]) != null ==>
   pairs[lockName].getFirst().val ==
   \old(pairs[lockName].getFirst().val) + 1;
  ensures \result == pairs[lockName].second;
  signals (IllegalStateException e) common() ** \old(isStopped);
 @*/
public synchronized DbNamedLock createLock(int lockName) {
```
Listing 24: The specification for the createLock method

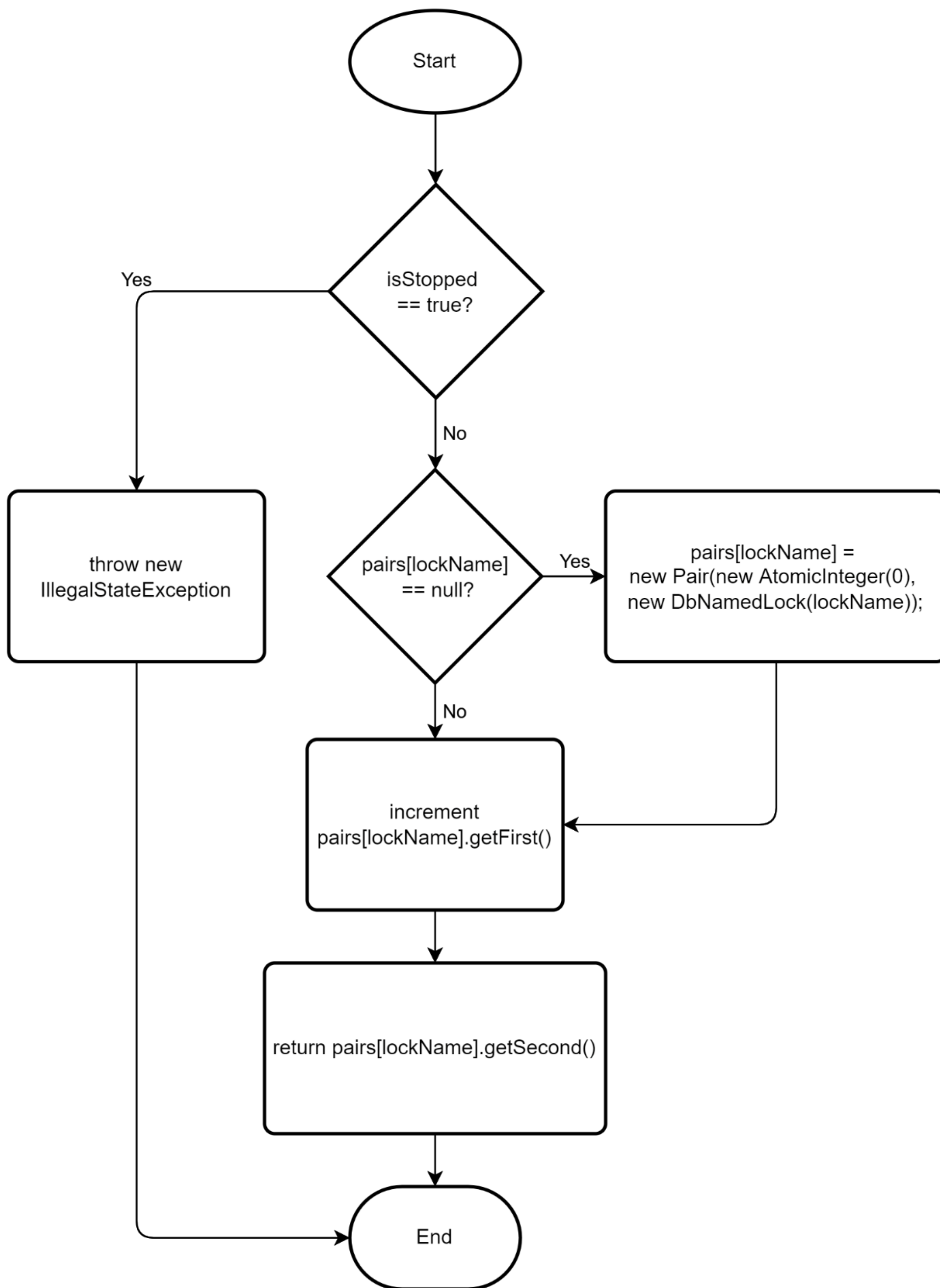

Figure 6: A flowchart model of the createLock method

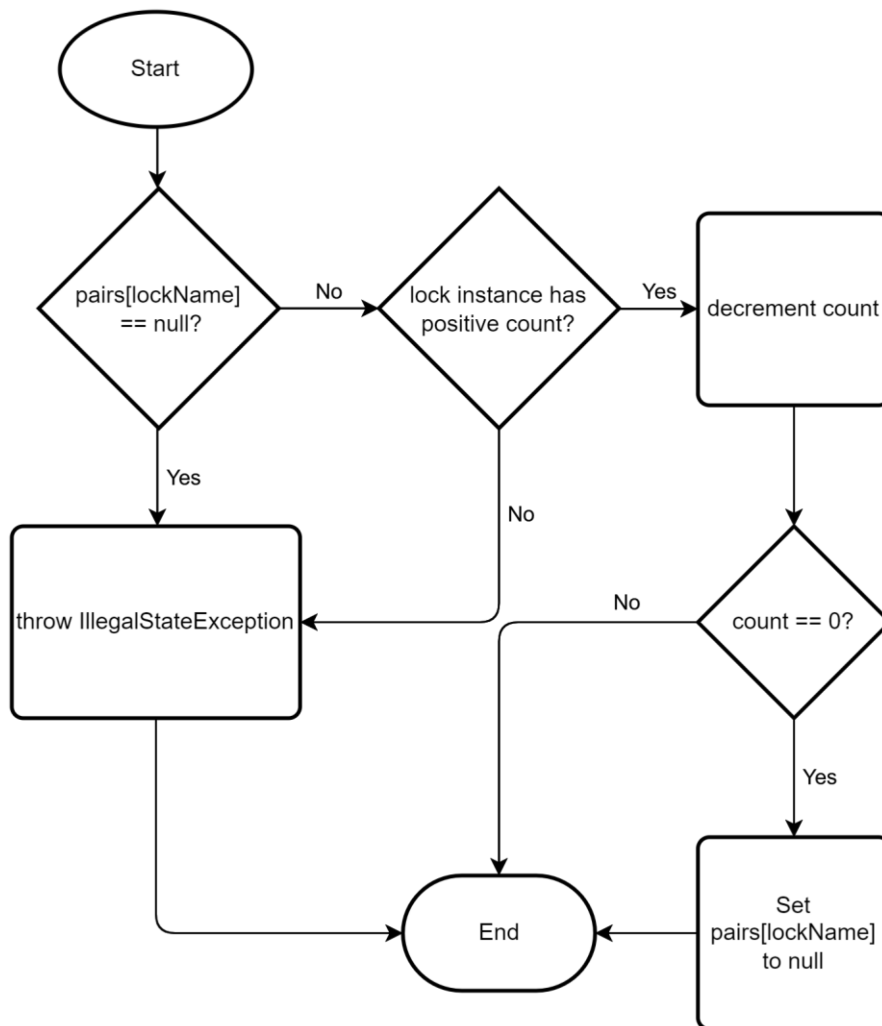

Figure 7: A flowchart model of the destroyLock method

Figure 7 shows a flowchart model of the destroyLock method. If the lock instance to be deleted does not exist, or there are no active references to the lock instance, an IllegalStateException is thrown. Otherwise, its corresponding count is decremented. If this count reaches 0, the pair is deleted (by removing the reference to the Pair). This effectively means that unused lock instances get cleaned up by the garbage collector.

```
/*@
  context common();
  context pureSupport(this.T);
  context lock != null;
  context 0 <= lock.getName() && lock.getName() < maxLockName;
  requires pairs[lock.getName()] != null;
  ensures \old(pairs[lock.getName()].getFirst().val) == 1 ==>
    pairs[lock.getName()] == null;
  ensures \old(pairs[lock.getName()].getFirst().val)  > 1 ==>
    pairs[lock.getName()] == \old(pairs[lock.getName()]) **
    pairs[lock.getName()].getFirst().val ==
    \old(pairs[lock.getName()].getFirst().val) - 1;
  signals (IllegalStateException e) common() ** pureSupport(this.T) **
(\let Pair p = pairs[lock.getName()]; p == null || p.getFirst().val == 0);
@*/
public synchronized void destroyLock(DbNamedLock lock) {
```

Listing 25: The specification for the destroyLock method

Listing 25 shows the specification for the destroyLock method. The context clauses indicate permissions and valid state for the lock manager and its instances. The requires clause indicates that the lock must exist in order to be destroyed. The ensures clauses indicate that when the final reference to the pair containing the lock instance is returned, the pair is set to null. In all other scenarios, the pair is the same object as before, but its reference count will be decremented. Finally, the signals clause indicates that when an IllegalStateException is thrown, the lock manager is still in a valid state, and either the pair was null (excluded by the precondition) or its first value equals 0.

7.2 Intermediate 2

As mentioned in Section 6.2.2 on page 40, intermediate 2 adds in read/write functionality. In intermediate 1, every lock granted exclusive access to the protected resource. However, in intermediate 2, a lock is either a read lock or a write lock. A write lock grants exclusive access, as before, but a read lock grants shared read access to the protected resource.

This allows for a more fine-grained locking policy, where many readers can concurrently access a protected resource, or a single writer can have exclusive access to the same resource.

The specification of intermediate 2 is heavily based on that of intermediate 1. Because of this, only some of the most important changes will be highlighted here. A complete overview of the changes made can be found in Appendix B.

7.2.1 DbNamedReadWriteLock

The DbNamedReadWriteLock is a class used to store a reentrant read/write lock. It has methods to create and retrieve its read lock and write lock, and has a method to destroy them both. The read and write locks are DbNamedLock objects. The read and write locks are synchronized through a field in the DbNamedReadWriteLock class called readWriteFlag. This is a flag which indicates whether the lock is unlocked (-1), locked in read mode (0) or locked in write mode (1). The read lock and write lock access this flag to determine whether a lock operation is legal.

7.2.1.1 Fields

The DbNamedReadWriteLock class has a number of fields:

- int lockName, for storing the lock's name
- DbLockManager lockManager, for creating and destroying lock instances
- AtomicInteger readWriteFlag, for indicating the current status of the lock and synchronizing the read and write locks
- DbNamedLock readLock, the read lock
- DbNamedLock writeLock, the write lock

7.2.1.2 Predicates

The DbNamedReadWriteLock class contains two predicates: the lock invariant and the common() predicate, which indicates valid state for the object and is responsible for synchronizing the DbLockManager and DbNamedReadWriteLock object.

- lock_invariant, this stores 1\2 permission for readWriteFlag.val. This ensures that write permission is only held when exclusive access to the class is present.
- common(), this stores the other half of the permissions for readWriteFlag.val, as well as write permissions for the readLock and writeLock objects. Aside from permissions, this predicate is also responsible for synchronization of the DbLockManager and DbNamedReadWriteLock instances. This is done by indicating that whenever readLock resp. writeLock is non-null, this lock instance must also be known by the lockManager.

7.2.1.3 Methods

As mentioned before, methods to retrieve and instantiate the readLock and writeLock fields are present. These methods lazily instantiate the readLock and writeLock fields, by first checking whether the field is null. If so, lockManager.createLock() is called, and the lock is instantiated. Then, the created (or stored) lock object is returned.

There's also a method to destroy both the readLock and writeLock objects. This close() method calls lockManager.destroyLock() with readLock/writeLock as parameters, if the locks were not null.

There are also two methods related to the readWriteFlag: getReadWriteFlag, to retrieve its current value, and casReadWriteFlag, to do a compare-and-set operation on the readWriteFlag object. These methods call the similarly named methods of the AtomicInteger class. See Section 7.1.1 for further details on these methods.

7.2.2 DbNamedLock

As mentioned in Section 7.2.1, the lock objects now access the DbNamedReadWriteLock object to synchronize their lock operations. However, not every lock is necessarily instantiated through a DbNamedReadWriteLock object. This required some changes in the way the locking mechanism worked, which are described below. First, the changes in the fields are described, then the changes made in the predicates, and finally, the changes in the implementation and specification of the locking methods are addressed.

7.2.2.1 Fields

There were some changes needed in the fields, in order to support the different access modes.

In addition to the fields described in 7.1.3.1, two additional fields were added:

- final boolean lockType – This field contains the lock instance's type. True indicates a write lock, false indicates a read lock.
- ghost bag<int> holders – This field contains the thread identifiers of all holders of the lock. A bag<int> is an unordered collection of elements (in this case integers), in which elements may occur multiple times. This field allows us to keep track of all concurrent holders of the lock, as well as the reentrancy level of the lock for each thread. If a given thread holds the lock reentrantly, its thread identifier occurs multiple times in the bag<int>. This replaces the usage of the holder ghost field, which is now only used for write locks.
- DbNamedReadWriteLock readWriteLock – This field holds the reference to the DbNamedReadWriteLock, if the lock was created through that class. If not, this field is null.

7.2.2.2 Predicates

Recall that in Section 7.1.3.2, the predicates of the DbNamedLock class were introduced.

In Listing 19, the lockset_part predicate was introduced. This predicate was slightly altered, in that the last implication now only holds for write locks. The corresponding permissions and values of the read locks were moved into the lock invariant, which is described below. The reason this was needed, is that many threads may share a read lock, but only one thread may hold the lockset_part predicate. Therefore, if the lockInstanceCount.val permission was stored in the lockset_part predicate, the lock can still only be held by a single thread.

In Listing 20, the lock invariant for the DbLockManager class was described. In this intermediate, this is expanded upon and altered significantly.

Below boolean assertions are now only used for the write locks, as the holder and owningThread fields are only used in write locks. Because of this, they're wrapped with (lockType ==> …)

- (holder == -1) == (lockInstanceCount.val == 0)
- (i != holder) ==> (heldArray[i] == 0)
- (heldArray[i] > 0) ==> (owningThread.val == i)

Below boolean assertions are only true for read locks, as it uses the bag<int> holders field and handles permissions slightly differently:

- (!lockType && instanceLockCount > 0) ==> Perm(instanceLockCount.val, 1\2)
- instanceLockCount.val == |holders|
- (\forall int i = 0 .. T; {:heldArray[i]:} == (i \in holders))

The first point here indicates that the full write permission for instanceLockCount.val is stored in the lock invariant, as the other half was already stored here.

The second point indicates that the reentrancy level of the lock is exactly equal to the cardinality of the holders field. This means that if the lock is held reentrantly 10 times, instanceLockCount.val is 10, and the amount of elements in holders is also 10. This can be a single reader, who holds the lock reentrantly 10 times, or multiple readers holding the lock (possibly reentrantly) for a total of 10 times.

The last point indicates that for any thread holding the lock, the number stored in heldArray corresponds to the number of times this thread's identifier occurs in the holders bag.

Furthermore, as some, but not all, locks were created using the DbNamedReadWriteLock, there were some assertions added to manage permissions for the case where it was used:

- readWriteLock != null ==> Perm(readWriteLock.readWriteFlag.val, 1\2)
- (readWriteLock != null && !lockType && {:heldArray[i]:} > 0) ==> readWriteLock.readWriteFlag.val == 0
- (readWriteLock != null && lockType && {:heldArray[i]:} > 0) ==> readWriteLock.readWriteFlag.val == 1
- (readWriteLock != null && readWriteLock.readWriteFlag.val != 1) ==> holder == -1

Here, the first point indicates that the lock invariant stores some of the permissions to access the readWriteFlag. The other half is required upon calling the tryLock or unlock.

The second point indicates that when the lock is held in read mode, the readWriteFlag's value reflects this.

The third point indicates that when the lock is held in write mode, the readWriteFlag's value reflects this.

Finally, the fourth point indicates that when the lock is not held in write mode, the value of the holder field is -1. This was added to ensure that the value of the holder field stays properly specified, even if the read lock had been held previously.

7.2.2.3 Methods

The specification and implementation for the methods were also altered to accommodate the changes described above.

The unlock method received an additional check at the end, to reset the readWriteFlag's value to unlocked upon the last unlock operation. Since this is a fairly minor change (one added if statement and catching off some illegal states), this is left out for brevity.

The tryLock method, however, did get significantly changed. The implementation had to be adjusted to allow for three different scenarios:

1. The lock was not instantiated through a DbNamedReadWriteLock object

2. The lock is a write lock and was instantiated through a DbNamedReadWriteLock object
3. The lock is a read lock and was instantiated through a DbNamedReadWriteLock object

These three scenarios require different behaviour, as the DbNamedReadWriteLock is used to synchronize the read- or write lock with its counterpart. A flowchart model of the implementation of the tryLock method in intermediate 2 is given in Figure 8.

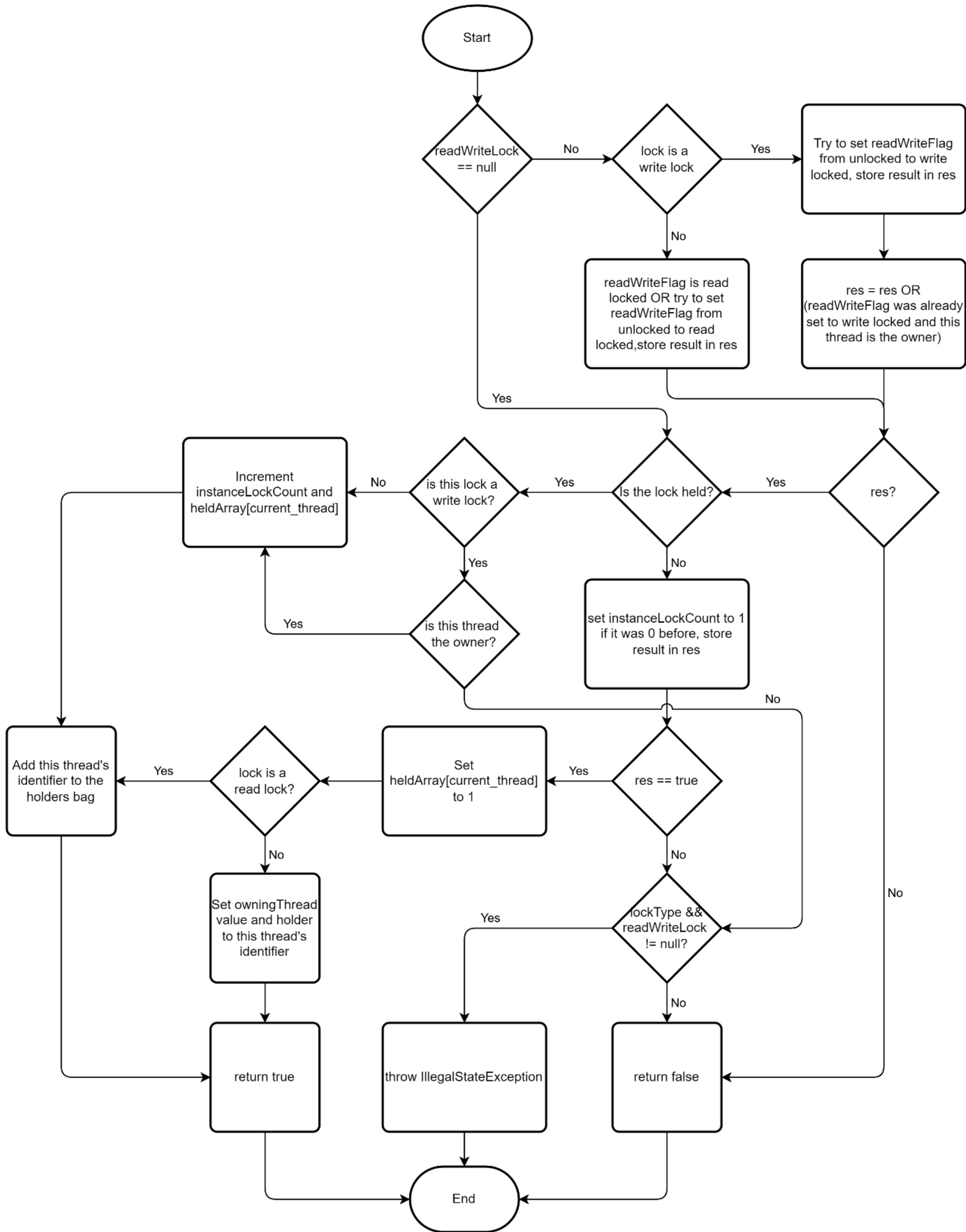

Figure 8: A flowchart model of the tryLock method

In the first scenario, there is no way to synchronize with another lock instance, so if a read lock and write lock are both created through the DbLockManager class, rather than through the DbNamedReadWriteLock class, they cannot possibly synchronize their locking behaviour, so mixed read and write access is possible in this case. This is a limitation of intermediate 2, which is solved in intermediate 3, when all locking operations occur through the database model.

The first scenario is the most similar to the implementation described in Section 7.1. If the lock in the first scenario was a write lock, then the behaviour is identical to that of the tryLock implementation described in Section 7.1.3.3. In particular, a flowchart model for the implementation of the tryLock method in intermediate 1 was given in Figure 4.

If the write lock was instantiated through a DbNamedReadWriteLock object, an additional step is taken before attempting to alter the state of the lock. This is the situation of the second scenario described above. In order to obtain the write lock, the readWriteFlag field in the DbNamedReadWriteFlag has to be set to 1 (write mode). If the field was previously set to -1 (unlocked), this succeeds and the lock operation may continue. Otherwise, it is checked that the readWriteFlag was already set to 1 and the owning thread of this lock is set to this thread's identifier. If so, the lock operation may continue as described in the previous paragraph. In any other case, the lock operation fails and false is returned.

In the first scenario, if the lock in question was a read lock but was not instantiated through a DbNamedReadWriteLock object, the behaviour is slightly different. If the lock was previously not held and is successfully obtained by this thread, rather than setting the holder and owningThread fields, the holders bag is updated, by adding the current thread's identifier to it. However, if the lock was previously held, then the instanceLockCount and heldArray[current_thread] are both incremented. Finally, true is returned.

In the third and final scenario, the read lock was instantiated through the DbNamedReadWriteLock class. If the DbNamedReadWriteLock's readWriteFlag is already set to 0 (read mode) or a compare-and-set operation successfully sets it from -1 (unlocked) to 0 (read locked), the lock operation may continue. Otherwise, false is returned and the lock attempt is aborted. After performing this check, the behaviour is identical to the read lock described in the previous paragraph.

Finally, the specification of the tryLock method, given in Listing 26, also got changed significantly, as compared to the specification of the tryLock method in intermediate 1, given in Listing 21.

In particular, there was an added context clause for the case where the lock was created through a DbNamedLock instance, in which permissions for readWriteFlag.val were given out. Recall that the other half is stored in the lock_invariant (see also Section 7.2.2.2). This ensures that a full write permission is available inside the method body.

Upon first obtaining the lock, access to the protected resource has to be given out. This is represented by subject.inv(). In intermediate 2, the postcondition which ensures this was split into two parts: one for a first write access, and another for a first read access.

```
/*@
  given int current_thread;
  context common() ** lockset_part(current_thread);
  context initialised(T);
  context readWriteLock != null ==>
    Perm(readWriteLock.readWriteFlag.val, 1\2);
  ensures (!\result ==>
    heldArray[current_thread] == \old(heldArray[current_thread]));
  ensures ( \result ==>
    heldArray[current_thread] == \old(heldArray[current_thread]) + 1);
  ensures (\old(heldArray[current_thread]) > 0 ==> \result);
  ensures ( lockType && \result && \old(heldArray[current_thread]) == 0 ==>
    subject.inv();
  ensures (!lockType && \result && \old(|holders|) == 0) ==> subject.inv();
  signals (IllegalStateException e) false; // should never occur
@*/
public boolean tryLock() {
   ...
}
```

Listing 26: The specification for the tryLock method in intermediate 2

7.2.3 DbLockManager

In intermediate 2, the pairs array is expanded to twice the size. The first half of the array is reserved for write pairs, and the second half is reserved for read pairs. This allows for easy indexing of the array, based on the lock name and lock type.

All of the knowledge and permissions contained within the common() and pureSupport() predicates in intermediate 1 were moved into the lock_invariant. This ensures that injectivity is present within synchronized methods, where write access is needed, but is hidden outside of them. When only read access to a single element is necessary, it is not needed to know the state of the entire array.

Instead, a new predicate common_part() is present, which contains all the information and permissions relating to a specific lock name and lock type combination. This includes permissions to access the relevant pair, as well as permissions over its fields (pair.first.val and pair.second.common()).

At the methods, the context clauses which previously indicated common() and pureSupport() are replaced with common_part(). Other than this, the specification is not significantly altered.

A ghost parameter was added to the destroyLock method. This ghost parameter holds the index for the lock to be deleted. The exact value of this parameter is defined in a requires clause:

```
requires i == lock.getName() + (lock.getType() ? 0 : maxLockName);
```

This allows i to be used inside the rest of the specification, rather than the expression which defines its value. This improves readability, and allows the solver to deduce that these identical expressions all have the same value. No comprehensive testing was done, though this is suspected to improve verification performance.

7.3 Intermediate 3

As mentioned in Section 6.2.3, intermediate 3 adds the database model. This model was described in detail in Section 6.1.8.

7.3.1 DbRow

The DbRow class represents a single database row. It contains three final fields: int lockName, int uuid and boolean lockType. These fields are final and have a constructor to initialise their values.

7.3.2 ListDbRow

The ListDbRow class represents a list of database rows. As support for generics was lacking (see Section 6.1.4), an implementation of a List class was made, which contains methods to add, remove and retrieve elements, as well as a few helper methods to aid in the implementation.

The list implementation has an underlying array, which stores the data. Upon calling add(), the array is expanded as needed and an element is added to it.

7.3.2.1 Fields

The class contains a number of fields:

- final int initialCapacity – This field determines the initial size of the underlying array.
- int size – This field keeps track of how many elements are currently in the data structure. This field determines at which index elements are added, and helps determine when the underlying array needs to be expanded.
- DbRow[] arr – This is the underlying array, which stores the references to the database rows. It initially has length initialCapacity, but is expanded to double the size whenever necessary.
- ghost seq<int> uuidSeq – This is a sequence of integers. A sequence is an immutable ordered list, which can be indexed into. This sequence is used to store the uuid's of the stored DbRow objects. This sequence is used to simplify specification of the remove method.

7.3.2.2 Predicates

The class also contains some predicates, which indicate valid state and are used to manage the permissions. There are three predicates in this class.

The common() predicate indicates valid state for the fields, except for the contents of the underlying array. It stores write permission to the non-final fields. Furthermore, it establishes that the underlying array must be initialised and have a positive length, the size field must be at least 0 and at most arr.length, and the amount of elements in uuidSeq must be equal to size.

The common_arr() predicate indicates valid state for the contents of the underlying array. It stores write permissions for each array element. For each element of arr which has an index strictly smaller than size, it establishes that the element is non-null, unique (no two indices refer to the same object) and has a unique uuid. Furthermore, a one-to-one mapping between uuidSeq and arr are established by specifying that for all indices i from 0 to size-1, uuidSeq[i] == arr[i].uuid. Finally, all elements of arr which have an index of size or greater are specified to be null.

Finally, the total() predicate simply combines the common() and common_arr() predicates.

7.3.2.3 Methods

Two helper methods exist for this class. The fastRemove method removes one element from an array from a specified index. All non-null elements that had a greater index than the removed element are shifted one position to the left. This method ensures that the only elements that are non-null are the left-most elements with index smaller than size, whereas all array elements with a

larger index remain null. This shifting is done through a call to the arrayCopy method, which is described below.

The arrayCopy method copies (part of) a source array into (part of) a destination array. Its behaviour is analogous to System.arrayCopy(). The implementation and specification of the arrayCopy method is given in Listing 27.

The specification establishes a few things:

- The source (src) and destination (dest) arrays are non-null and the length parameter is non-negative (line 2)
- The srcPos and destPos parameters are given such that the sub-arrays to copy are within the array bounds (lines 3-4)
- The result of the method is that dest[destPos..destPos+length] = \old(src[srcPos..srcPos+length]). The \old is needed in case the source array and destination arrays are the same array (lines 32-33).
- The right amount of permissions are held for each array element (read for src[srcPos .. srcPos+length], write for dest[destPos .. destPos+length]). This is described by lines 7-31.

The last point requires some additional explanation. A full 24 lines of specification is needed to specify the correct amount of permissions over the arrays. This is needed, because in total, only a full write (1) permission can be held over any array element.

If we were to note 1\2 permissions for src[srcPos .. srcPos+length] and 1 permission for dest[destPos .. destPos + length], then 1 + 1\2 permissions may be held over some of the array elements. This happens in the case that the two arrays are the same, and there's some overlap in the ranges of [srcPos .. srcPos + length) and [destPos .. destPos + length). To avoid this, the specification specifies each of the ways in which overlap may or may not be present.

The different ways in which overlap may be present in this situation are illustrated in Table 2. Here, the vertically shaded cells represent the range of src[srcPos .. srcPos + length], and the horizontally shaded cells represent the range of dest[destPos .. destPos + length]. The cells shaded in a grid pattern represent the overlap between these two ranges. In each of the rows, the length parameter is set to 5.

First, 1\2 permission is specified over the array elements of the src array (lines 5-6).

If the source and destination arrays are not the same array, there is no problem, and a full write permission can be specified over the array elements of dest (lines 29-31). This situation is not illustrated in the table.

There are two situations in which the arrays are equal, but there is no overlap. These are illustrated by the row 1 and 4 of Table 2. These situations are described by respectively lines 13-17 and lines 24-28 of Listing 27. In this case, a full write permission can be specified over the elements in the range [destPos .. destPos+length), as there is no overlap with the elements in the range [srcPos .. srcPos+length).

There are also two situations in which the arrays partially overlap. These situations are illustrated by rows 2 and 3 of Table 2. The situation in row 2 is covered by lines 7-12, and the situation in row 3 is covered by lines 18-23. Note that the situation where there is a 100% overlap between the two ranges is also covered by lines 7-12. In these situations, a full write permission is specified over the parts of the [destPos .. destPos+length) range which don't overlap with [srcPos .. srcPos+length),

whereas 1\2 permission is specified over the part of the ranges which do overlap. This, in combination with the 1\2 permission specified over the source range, ensures that write (1) permission is available over the overlapping parts.

1																								
2																								
3																								
4																								

Table 2: Overlap modes of the arrayCopy method

Something interesting also happens inside the method body (lines 37-59). If the source range is to the right of the destination range, the elements to copy are copied left-to-right. This situation is illustrated by rows 2 and 3 of Table 2.

However, if the same left-to-right copying strategy is applied to the situation illustrated by row 2 of Table 2, something may go wrong. Suppose that the source range contains the elements [1, 2, 3, 4, 5]. Then using a left-to-right copying strategy, by the time the third element is reached, the source range now contains the elements [1, 2, 1, 2, 5]. The end result would not be that the original elements of the source array are copied into the destination array. In order to remedy this, in the case where the source array starts to the right of the destination, the copying is one in a right-to-left manner. This ensures that the elements are copied correctly. Returning to the situation of row 2, the end result would now be [1, 2, 1, 2, 3, 4, 5] for the shaded areas of row 2. Generalising this, the end result is that dest[destPos .. destPos + length] == \old(src[srcPos .. srcPos + length]), which is exactly what is specified in the postcondition of the method described in lines 32-33 of Listing 27.

```
[ 1]/*@
[ 2] context_everywhere src != null ** dest != null ** length >= 0;
[ 3] context_everywhere 0 <= srcPos && srcPos + length <= src.length;
[ 4] context_everywhere 0 <= destPos && destPos + length <= dest.length;
[ 5] context_everywhere (\forall* int i = srcPos .. srcPos + length;
[ 6]  Perm({:src[i]:}, 1\2));
[ 7] context_everywhere src == dest && srcPos <= destPos &&
[ 8]  srcPos + length > destPos ==>
[ 9]   // srcPos is below destPos, but there is some overlap
[10]   (\forall* int i = destPos .. srcPos + length; ({:dest[i]:}, 1\2)) **
[11]   (\forall* int i = srcPos + length .. destPos + length;
[12]    Perm({:dest[i]:}, write));
[13] context_everywhere src == dest && srcPos <= destPos &&
[14]  srcPos + length <= destPos ==>
[15]   // srcPos is below destPos, no overlap
[16]   (\forall* int i = destPos .. destPos + length;
[17]    Perm({:dest[i]:}, write));
[18] context_everywhere src == dest && srcPos > destPos &&
[19]  destPos + length > srcPos ==>
[20]   // srcPos is above destPos, overlap
[21]   (\forall* int i = destPos .. srcPos; Perm({:dest[i]:}, write)) **
[22]   (\forall* int i = srcPos .. destPos + length;
[23]    Perm({:dest[i]:}, 1\2));
[24] context_everywhere src == dest && srcPos > destPos &&
[25]  destPos + length <= srcPos ==>
[26]   // srcPos is above destPos, no overlap
[27]   (\forall* int i = destPos .. destPos + length;
[28]    Perm({:dest[i]:}, write));
[29] context_everywhere src != dest ==>
[30]  (\forall* int i = destPos .. destPos + length;
[31]   Perm({:dest[i]:}, write));
[32] ensures (\forall int i = destPos .. destPos + length;
[33]  {:dest[i]:} == \old(src[srcPos + (i - destPos)]));
[34]@*/
[35]private void arraycopy(DbRow[] src, int srcPos, DbRow[] dest,
[36]int destPos, int length) {
[37] if (srcPos >= destPos) {
[38]  /*@
[39]  loop_invariant 0 <= i && i <= length;
[40]  loop_invariant (\forall int j = srcPos + i .. srcPos + length;
[41]   {:src[j]:} == \old(src[j]));
[42]  loop_invariant (\forall int j = destPos .. destPos + i;
[43]   {:dest[j]:} == \old(src[srcPos + (j - destPos)]));
[44]  @*/
[45]  for (int i = 0; i < length; i++) {
[46]   dest[destPos + i] = src[srcPos + i];
[47]  }
[48] } else {
[49]  /*@
[50]  loop_invariant -1 <= i && i < length;
[51]  loop_invariant (\forall int j = srcPos .. srcPos + i + 1;
[52]   {:src[j]:} == \old(src[j]));
[53]  loop_invariant (\forall int j = destPos + i + 1 .. destPos + length;
[54]   {:dest[j]:} == \old(src[srcPos + (j - destPos)]));
[55]  @*/
[56]  for (int i = length - 1; i >= 0; i--) {
[57]   dest[destPos + i] = src[srcPos + i];
[58]  }
[59] }
[60]}
```

Listing 27: Implementation and specification of the arrayCopy method

Aside from the helper methods, there are also a number of methods to add, remove or retrieve elements from the list.

The constructor establishes an empty list, with an initial size of 0 and an initial capacity of 10.

The add method takes in an element to add, and checks that enough space is left in the list. If not, the list is expanded to twice the original size. Then, the element to add is appended to the end of the list and the uuidSeq is updated with the uuid of the new element. The element to add must not already be in the list, and must have a unique uuid compared to the other elements in the list. The specification of the add method is given in Listing 28.

Note that if the array had reached its maximum capacity prior to this call to add, the array's length is doubled, and all elements are copied over.

```
/*@
 context total();
 requires elem != null ** !(elem.uuid \in uuidSeq);
 requires (\forall int i = 0 .. size; {:arr[i]:} != elem);
 ensures \old(size) >= \old(arr.length) ==> (
  arr.length == \old(arr.length * 2) &&
  (\forall int i = 0 .. \old(arr.length); {:arr[i]:} == \old(arr[i])) &&
  (\forall int i = size .. arr.length; arr[i] == null)
 );
 ensures size == \old(size) + 1;
 ensures arr[size - 1] == elem;
 ensures uuidSeq == \old(uuidSeq) + seq<int>{elem.uuid};
@*/
public void add(DbRow elem) {
```

Listing 28: The specification of the add method

The remove method takes in a UUID and, if present in the list, removes the corresponding element. All elements to the right of the removed element are shifted one position to the left, and the removed element is returned. If no element with the given UUID was present in the list, then null is returned instead. The specification for the remove method is given in Listing 29.

Rather than specifying the presence of the UUID in the actual array, the presence of the UUID in the uuidSeq is specified. This is made possible through the total() predicate described in Section 7.3.2.2. Specifically, in common_arr(), which is part of total(), a one-to-one mapping is specified between the elements of uuidSeq and the elements of arr. Because of this, the presence of the UUID in uuidSeq is enough to specify whether or not the UUID is in the array.

Note that an additional ghost return value is present, namely x. This indicates the position of the element in the array. It is only used to specify the exact result of the method call, and is not stored at any call sites, as this information is not useful to the caller. Because of this, it is possible to specify which elements are shifted to the left, which element is returned and how the uuidSeq is updated.

```
/*@
 yields int x;
 context total();
 ensures arr.length == \old(arr.length);
 ensures !(uuid \in \old(uuidSeq)) ==>
  \result == null &&
  uuidSeq == \old(uuidSeq) &&
  x == -1 &&
  size == \old(size) &&
  (\forall int j = 0 .. size; {:arr[j]:} == \old(arr[j]));
 ensures uuid \in \old(uuidSeq) ==>
  0 <= x && x < \old(size) **
  \result == \old(arr[x]) **
  size == \old(size) - 1 **
  (\forall int j = 0 .. x; {:arr[j]:} == \old(arr[j])) **
  (\forall int j = x .. size; {:arr[j]:} == \old(arr[j + 1])) **
  uuidSeq == \old(uuidSeq).removeAt(x);
@*/
public DbRow remove(int uuid) {
```
Listing 29: The specification of the remove method

7.3.3 DbModel

The DbModel class is a model of real database behaviour. A brief introduction to the class, its purpose and its structure was given in Section 6.1.8.

7.3.3.1 Fields

The DbModel class contains a number of fields and ghost fields.

- final int maxLockName – This field stores how many unique locks may exist at any one time.
- final ListDbRow[] lockData – This field stores a list of database rows for each lock. This way, if a lock is unlocked, lockData[i] == 0, and if a lock is locked, lockData[i] > 0.
- ghost final int[] heldArray – This array keeps track of how many times each lock is held.
- ghost final int[] readWrite – This array keeps track of whether each lock is unlocked (-1), read locked (0) or write locked (1).

7.3.3.2 Predicates

The DbModel class contains a number of predicates. These predicates indicate valid state for the database model to be in, and store the permissons needed to use the different methods in the DbModel class. The predicates are shown in Listing 30, and the lock invariant is shown in Listing 31.

```
[ 1]inline resource common() = \array(heldArray, maxLockName) **
[ 2] \array(readWrite, maxLockName) ** \array(lockData, maxLockName) **
[ 3] this != null ** committed(this);
[ 4]
[ 5]inline resource perm_all(int lockId) =
[ 6] perm_part(lockId) **
[ 7] [1\2]lockData[lockId].common() **
[ 8] [1\2]lockData[lockId].common_arr();
[ 9]
[10]inline resource perm_part(int lockId) =
[11] 0 <= lockId && lockId < maxLockName **
[12] Perm(heldArray[lockId], 1\2) ** Perm(readWrite[lockId], 1\2) **
[13] Perm(lockData[lockId], 1\2) ** lockData[lockId] != null;
```
Listing 30: The predicates of the DbModel class

```
[ 1]public /*@ lock_invariant
[ 2] common() ** Perm(heldArray[*], 1\2) ** Perm(readWrite[*], 1\2) **
[ 3] Perm(lockData[*], 1\4) **
[ 4] (\forall int i = 0 .. maxLockName; lockData[i] != null) **
[ 5] (\forall int i = 0 .. maxLockName, int j = 0 .. maxLockName; i != j;
[ 6]  {:lockData[i]:} != {:lockData[j]:}) **
[ 7] (\forall* int i = 0 .. maxLockName; [1\2]lockData[i].common()) **
[ 8] (\forall int i = 0 .. maxLockName, int j = 0 .. maxLockName; i != j;
[ 9]  {:lockData[i].arr:} != {:lockData[j].arr:}) **
[10] (\forall* int i = 0 .. maxLockName; [1\2]lockData[i].common_arr()) **
[11] (\forall  int i = 0 .. maxLockName, int j = 0 .. maxLockName,
[12] int k = 0 .. lockData[i].size(), int l = 0 .. lockData[j].size();
[13] i != j;
[14]  {:lockData[i].arr[k]:} != {:lockData[j].arr[l]:} &&
[15]  lockData[i].arr[k].uuid != lockData[j].arr[l].uuid
[16] ) **
[17] (\forall  int i = 0 .. maxLockName; {:heldArray[i]:} >= 0) **
[18] (\forall  int i = 0 .. maxLockName;
[19]  -1 <= {:readWrite[i]:} && {:readWrite[i]:} <= 1) **
[20] (\forall  int i = 0 .. maxLockName;
[21]  ({:heldArray[i]:} == 0) == ({:readWrite[i]:} == -1)) **
[22] (\forall  int i = 0 .. maxLockName;
[23]  {:readWrite[i]:} == 1 ==> {:heldArray[i]:} == 1) **
[24] (\forall  int i = 0 .. maxLockName;
[25]  {:heldArray[i]:} == lockData[i].size()) **
[26] (\forall  int i = 0 .. maxLockName, int j = 0 .. lockData[i].size();
[27]  readWrite[i] == 0; !{:lockData[i].arr[j].lockType:}) **
[28] (\forall  int i = 0 .. maxLockName, int j = 0 .. lockData[i].size();
[29]  readWrite[i] == 1;  {:lockData[i].arr[j].lockType:})
[30] ;
[31]@*/class DbModel {
```

Listing 31: The lock invariant of the DbModel class

The common() predicate indicates that each of the fields (except for maxLockName) is properly initialised.

The perm_part(int lockId) predicate indicates, for a given lock identifier, 1\2 permissions for heldArray[lockId], readWrite[lockId] and lockData[lockId]. The other half of the permissions for heldArray and readWrite are stored in the lock invariant, which is shown in Listing 31 and described later in this section. For lockData, after initialisation, the array elements are read-only. This is because after initialisation, the lists of DbRows only need to be edited, not replaced entirely.

The perm_part() predicate also indicates that the underlying list storing the lock rows must be non-null. This predicate allows for these fields to be accessed inside the specification, and ensures that a full write permission to a specific array element can be held if this predicate is held along with the lock invariant.

The perm_all(int lockId) predicate indicates perm_part(lockId), as well as half of the common() and common_arr() predicates of lockData[lockId]. Again, the other half of these permissions are stored in the lock invariant, such that, inside methods that require write permissions, the full write permission is available.

The lock invariant contains permission to access all fields, and indicates valid state for the object. Note that only 1\4 permission for lockData[*] is specified. This is because after initialisation, the array elements (the ListDbRow objects) no longer need to be replaced, only altered.

Lines 4-16 of Listing 31 describe the proper initialisation of the lockData field and its elements. Each of the elements are non-null, injective (unique) with regards to other elements in the array, and are in a valid state, as specified through the common() and common_arr() predicates of the ListDbRow class. These predicates were described in Section 7.3.2.2. The underlying arrays are also injective (unique) with regards to other elements of the lockData array. Furthermore, since each DbRow has a unique UUID, these UUIDs are specified to be unique with regards to all other DbRow objects contained within the lockData array.

Lines 17-19 describe valid bounds for the fields. The heldArray field contains non-negative integers, one for each lock name. The readWrite field contains integers with the values -1 (unlocked), 0 (read locked) or 1 (write locked), with one entry per lock name.

Lines 20-23 describe how the heldArray and readWrite fields relate to one another. When an element heldArray[i] has the value 0 (unlocked), then readWrite[i] must have the value -1 (unlocked). When an element readWrite[i] has the value 1 (write locked), then heldArray[i] must have the value 1.

Note that the value never gets higher than 1 in this case, since DbNamedLock.tryLock() only contacts the database upon attempting to obtain the lock initially. All other reentrant tryLock calls are handled locally. This is further described in Section 7.3.4.

Finally, lines 24-29 describe how the heldArray and readWrite fields relate to the lockData field. Lines 24-25 state that the value of heldArray[i] is always equal to lockData[i].size(). This means that for each unique thread holding the lock, one database row must exist in the lockData[i] list. Lines 26-29 indicate that whenever readWrite[i] is 0 (read locked), the lockType of all elements of the corresponding list lockData[i] must be false (read), and whenever readWrite[i] is 1 (write locked), the lockType of all elements of the corresponding list lockData[i] must be true (write).

This all combines into a state, where the heldArray, readWrite and lockData fields are linked sufficiently to ensure the database is always in a valid state. Furthermore, when holding the lock invariant, a full write permission is available for most fields (but not lockData[*]), and elements can be added or removed from the underlying lists.

7.3.3.3 Methods

The constructor initialises the database model in an empty state. This means that all locks are considered unlocked.

The checkReadWrite() method checks whether, given the current state of the system, the lock with the name lockName can be locked. This is the case when a read lock is requested and the lock is held in read mode, or when a read or write lock is requested when the lock is unlocked.

The insert() method takes in three parameters: a lockName, a UUID and a lockType. It requires that the UUID is not yet in the list corresponding to the lockName. If the lock can be locked in the current state of the system (as reported by checkReadWrite), 1 is returned and a new DbRow is inserted into the correct ListDbRow. Otherwise, 0 is returned and the state is unchanged. This is captured in the specification given in Listing 32.

The requirements in lines 2 and 5 indicate that the lockName parameter must be within valid bounds, and the UUID that is to be inserted is unique with regards to the other UUIDs already in the list.

The context clauses in lines 3 and 4 indicate that all fields are properly initialised and that write permissions are available to all fields (except for lockData[lockName]), in combination with the lock invariant.

The ensures clause in lines 6-12 captures the case where the lock is already locked in an incompatible manner. If the lock was already write locked, or read locked, but a write lock is requested, then 0 is returned and the state remains unchanged. The values of heldArray, readWrite and lockData remains the same. For the other fields, only read permission is available, and thus, they cannot possibly have changed values.

The ensures clauses in lines 13-26 captures the case where the lock can be locked by the caller. In this case, 1 is returned and a new DbRow is inserted with the given lockName, UUID and lockType fields is inserted. This occurs when the lock was unlocked, or read locked and a read lock is requested. The readWrite[lockName] element is changed to the correct read or write type, heldArray[lockName] is incremented, as is the size of the corresponding list. Finally, the inserted row has the correct values for its fields, and its UUID is added to the uuidSeq field.

```
[ 1]/*@
[ 2]requires 0 <= lockName && lockName < maxLockName;
[ 3]context common();
[ 4]context perm_all(lockName);
[ 5]requires !(uuid \in lockData[lockName].uuidSeq);
[ 6]ensures \old(readWrite[lockName]) == 1 ||
[ 7] \old(readWrite[lockName]) == 0 && lockType ==>
[ 8]  \result == 0 &&
[ 9]  heldArray[lockName]         == \old(heldArray[lockName]) &&
[10]  lockData[lockName].size     == \old( lockData[lockName]).size &&
[11]  lockData[lockName].uuidSeq == \old( lockData[lockName].uuidSeq) &&
[12]  readWrite[lockName]         == \old(readWrite[lockName]);
[13]ensures (\old(readWrite[lockName]) == -1 ||
[14] (\old(readWrite[lockName]) == 0 && !lockType)) ==>
[15]  \result == 1 **
[16]  readWrite[lockName] == (lockType ? 1 : 0) **
[17]  heldArray[lockName]        == \old(heldArray[lockName]) + 1 **
[18]  lockData[lockName].size() == \old( lockData[lockName].size) + 1 **
[19]  (\let int s = lockData[lockName].size() - 1;
[20]   0 <= s ** s < lockData[lockName].size() **
[21]   lockData[lockName].get(s).lockName == lockName **
[22]   lockData[lockName].get(s).uuid     == uuid **
[23]   lockData[lockName].get(s).lockType == lockType **
[24]   lockData[lockName].uuidSeq         ==
[25]    \old(lockData[lockName].uuidSeq) + seq<int>{uuid}
[26]  );
[27]@*/
[28]public synchronized int insert(int lockName, int uuid,
[29] boolean lockType) {
```
Listing 32: The specification of the insert method

The delete() method takes in a lockName and a UUID, and deletes the corresponding DbRow object if it exists. The specification for the delete() method is given in Listing 33.

The precondition in line 3 indicates that the lockName parameter must be within valid bounds. The context clauses indicate that all fields are initialised, and we have full write permission over most fields (except the lockData[lockName] field, as before).

The ensures clause in line 6-7 indicates that the length of the underlying array remains unchanged.

The ensures clause in lines 8-15 indicates what happens when no DbRow with the given lockName and UUID exists in the list. In this case, 0 is returned and the state is unchanged.

The ensures clause in line 16-41 indicates what happens when a DbRow with the given lockName and UUID does exist in the list. In this case, 1 is returned. Note that in line 2, an extra ghost parameter x is specified. This is an additional return value, which is used here in the specification to specify exactly which DbRow is deleted. Line 18 indicates the bounds for this value. Line 20 ensures that the DbRow which was at index x indeed has the correct UUID, and lines 21-22 ensures that this UUID is removed from the uuidSeq, as is also specified in line 25.

Lines 19 and 23-24 indicate that exactly one element was deleted, and lines 26-30 indicate exactly how the uuidSeq was altered.

```
[ 1]/*@
[ 2]yields int x;
[ 3]requires 0 <= lockName && lockName < maxLockName;
[ 4]context common();
[ 5]context perm_all(lockName);
[ 6]ensures lockData[lockName].arr.length ==
[ 7] \old(lockData[lockName].arr.length);
[ 8]ensures !(uuid \in \old(lockData[lockName].uuidSeq)) ==>
[ 9] \result == 0 &&
[10] lockData[lockName].size() == \old(lockData[lockName].size()) &&
[11] heldArray[lockName] == \old(heldArray[lockName]) &&
[12] readWrite[lockName] == \old(readWrite[lockName]) &&
[13] lockData[lockName].uuidSeq == \old(lockData[lockName].uuidSeq) &&
[14] (\forall int j = 0 .. lockData[lockName].size();
[15]  {:lockData[lockName].get(j):} == \old(lockData[lockName].get(j)));
[16]ensures uuid \in \old(lockData[lockName].uuidSeq) ==>
[17] \result == 1 &&
[18] 0 <= x && x < \old(lockData[lockName].size) &&
[19] heldArray[lockName] == \old(heldArray[lockName]) - 1 &&
[20] \old(lockData[lockName].arr[x].uuid) == uuid &&
[21] lockData[lockName].uuidSeq ==
[22]  \old(lockData[lockName].uuidSeq).removeAt(x) &&
[23] |lockData[lockName].uuidSeq| ==
[24]  \old(|lockData[lockName].uuidSeq|) - 1 &&
[25] !(uuid \in lockData[lockName].uuidSeq) &&
[26] (\forall int i = 0 .. x; lockData[lockName].uuidSeq[i] ==
[27]  \old(lockData[lockName].uuidSeq[i])) &&
[28] (\forall int i = x .. lockData[lockName].size;
[29]  lockData[lockName].uuidSeq[i] ==
[30]   \old(lockData[lockName].uuidSeq[i + 1])) &&
[31] (\forall int i = lockData[lockName].size ..
[32]  lockData[lockName].arr.length;
[33]   lockData[lockName].arr[i] == null) &&
[34] (\old(heldArray[lockName]) == 1 ==> readWrite[lockName] == -1) &&
[35] (\old(heldArray[lockName]) >  1 ==>
[36]  readWrite[lockName] == \old(readWrite[lockName])) &&
[37] lockData[lockName].size == \old(lockData[lockName].size) - 1 &&
[38] (\forall int j = x .. lockData[lockName].size;
[39]  {:lockData[lockName].arr[j]:} ==
[40]   \old(lockData[lockName].arr[j + 1]));
[41]@*/
[42]public synchronized int delete(int lockName, int uuid) {
```

Listing 33: The specification of the delete method

7.3.4 DbNamedLock

Intermediate 3 introduces the database model. The DbNamedLock class now interfaces with the database model to obtain locks, rather than only locking on local lock instances. Because of this, a few things are changed:

- Lock instances are now thread-local, meaning each lock instance can only be accessed by a single thread, its 'owning thread'. Each lock instance has their own UUID, which is used to identify the database rows belonging to this lock instance (and by extension, each calling thread).
- The lock, tryLock and unlock operations now query the database to determine whether a lock can be obtained, rather than deciding this based on local lock state of this instance or the DbNamedReadWriteLock. Because of this, the implementation for each of them is significantly altered.
- A new method releaseLock() was added, which signals the database that the lock row must be deleted
- Because the lock operates on a single thread, and the locking logic has been moved to the database, the lock instance no longer holds any information over the lock state of other threads. Instead, only the local locking state is required. This simplifies the fields and predicates.

Since the changes made were significant, the sections below describe the implementation and specification of intermediate 3 without referring back to the specification and implementation of intermediate 2. The full list of changes is found in Appendix B.

7.3.4.1 Fields

The fields of the DbNamedLock class were updated. This is because locks now only hold lock state over a single thread. The following fields now exist:

- ghost final int T – This field indicates the maximum number of threads in existence. Only used to specify that the current thread's identifier is legal (0 <= thread_id < T).
- ghost int heldCount – This field keeps track of how many times this lock is held reentrantly by this thread. This ghost field can be accessed from within the specification, also if the lock is unlocked.
- ghost final Subject subject – This field represents the protected resource. Its inv() predicate represents access to the protected resource, which is granted upon first locking the lock.
- int lockInstanceCount – This field keeps track of the reentrancy level of the lock. Unlike the heldCount ghost field, when the lock is unlocked, the full write permission is held by the lock invariant, so this field cannot be referred to in the specification. The value of this field is always equal to heldCount.
- final DbModel dbModel – The reference to the database model.
- final boolean lockType – The lock type of this lock instance. A value of false represents a read lock, and a value of true represents a write lock.
- final int owningThread – The thread identifier of this lock's owner. Only the lock's owner is permitted to call methods of this lock instance.
- final int uuid – The UUID of this lock instance. Used to identify database rows corresponding to this lock instance.

7.3.4.2 Predicates

- initialised(int N) – this predicate indicates that T == N.

- common() – This predicate indicates that all fields are initialised and the lockName is within valid bounds (0 <= lockName < dbModel.maxLockName).
- lockset_part(int thread_id) – This predicate holds 1\2 permissions over heldCount. When the lock is locked, it also holds 1\2 permission over lockInstanceCount.
- lock_invariant – Since all lock state regarding other threads was moved out of DbNamedLock, this predicate is simplified greatly. The lock_invariant is given in Listing 34.

```
[ 1]public /*@ lock_invariant
[ 2] T > 0 ** Perm(heldCount, 1\2) ** heldCount >= 0 **
[ 3] dbModel != null **
[ 4] 0 <= lockName ** lockName < dbModel.maxLockName **
[ 5] Perm(instanceLockCount, 1\2) **
[ 6] (instanceLockCount == 0 ==>
[ 7]  subject.inv() ** Perm(instanceLockCount, 1\2)) **
[ 8] (!lockType && instanceLockCount > 0 ==>
[ 9]  Perm(instanceLockCount, 1\2)) **
[10] heldCount == instanceLockCount;
[11]@*/ class DbNamedLock implements NamedLock {
```

Listing 34: The specification of the lock_invariant of DbNamedLock

Lines 2-5 hold 1\2 permission for heldCount and instanceLockCount and indicate proper initialisation of the T, heldCount, dbModel and lockName fields.

Lines 6-7 indicate that when the lock is not held, the permissions needed to access the protected resource are stored in the lock_invariant. Furthermore, the other half of the permissions to access instanceLockCount are also stored here. This ensures that when the lock_invariant is held while the lock is unlocked, the full write permission is available.

Lines 8-9 indicate that when the lock is read locked, the other half of permissions needed to access lockInstanceCount are also stored in the lock invariant. This is a leftover from intermediate 2, when the permission needed to be managed in this way. Since the value of instanceLockCount is always equal to heldCount, that field can be used in the specification instead, and this is no problem (see paragraph below).

Finally, line 10 indicates that the value of heldCount is always equal to that of instanceLockCount. This allows us to use heldCount in the specification to indicate what happens to the state of DbNamedLock after locking and unlocking the lock.

7.3.4.3 Methods

For all methods, if any other thread than the owning thread call any of its methods, an IllegalStateException is thrown.

The tryLock() method attempts to acquire the lock. A flowchart model of its implementation is given in Figure 9. If the lock is already held, the local reentrancy level is updated by incrementing the lockInstanceCount and heldCount fields, and then true is returned. Otherwise, an attempt is made to insert a new lock row into the database. If this succeeds, the lock is considered to be obtained and the lockInstanceCount and heldCount fields are incremented and true is returned. Otherwise, the tryLock call fails and false is returned.

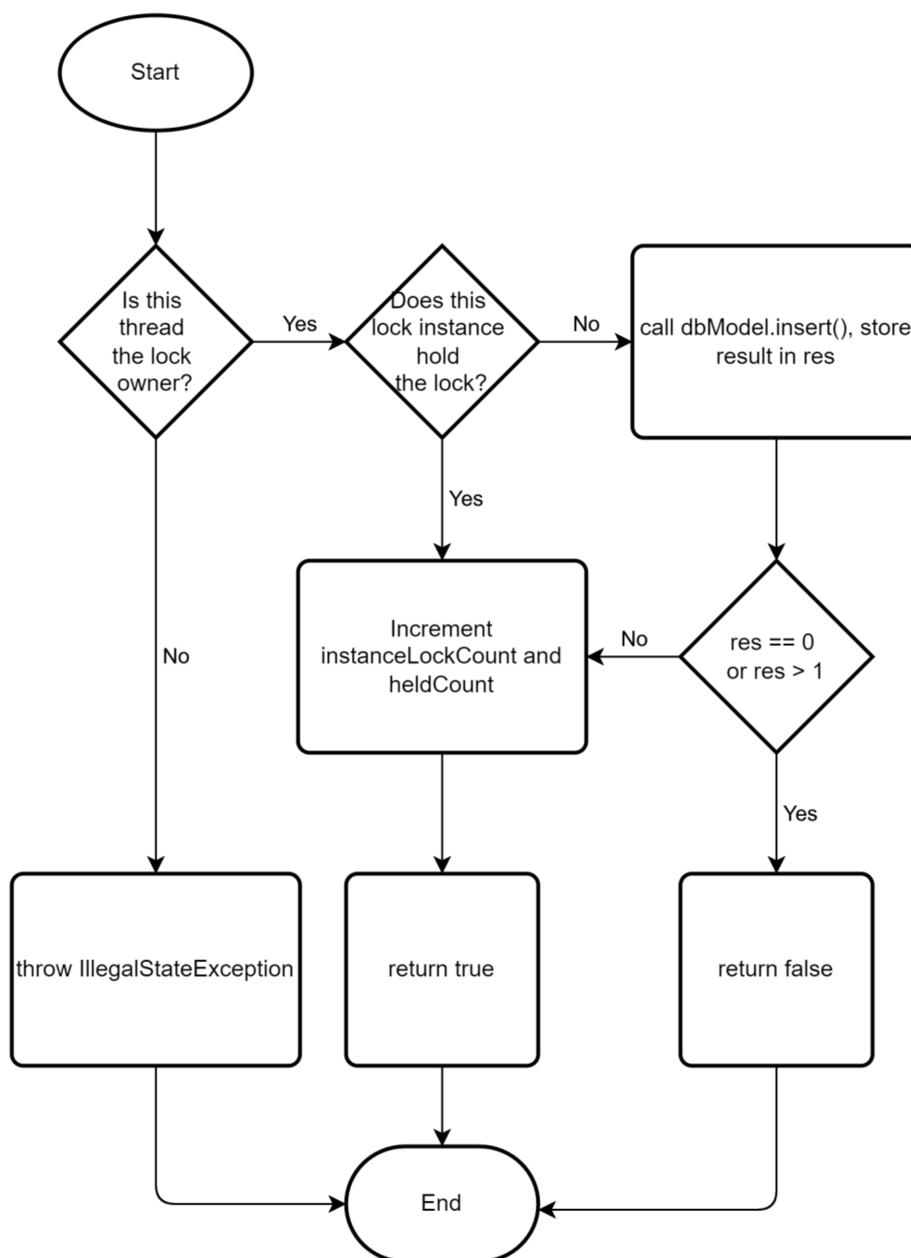

Figure 9: A flowchart model of the tryLock method

```
[ 1]/*@
[ 2] given int current_thread;
[ 3] context common() ** lockset_part(current_thread);
[ 4] context dbModel.common() ** dbModel.perm_all(lockName);
[ 5] context initialised(T);
[ 6] ensures (!\result ==> heldCount == \old(heldCount));
[ 7] ensures ( \result ==> heldCount == \old(heldCount) + 1);
[ 8] ensures \old(heldCount > 0 ==> \result);
[ 9] ensures \old(heldCount == 0 && \result) ==> subject.inv();
[10] signals (IllegalStateException e) owningThread != current_thread;
[11] // should never occur
[12]@*/
[13]public boolean tryLock() {
```

Listing 35: The specification of the tryLock method

The specification for the tryLock() method is given in Listing 35. The context clauses in lines 3-5 indicate valid state, and the ensures clauses in lines 6-7 indicate how the state is transformed if a positive or negative result is encountered. Line 8 indicates that when the heldCount was already held, the tryLock call will always succeed. Note that no guarantees are given for the initial call, as this is dependent on state outside of the scope of this specification, namely the state of the database model. Line 9 indicates that upon a first successful tryLock() operation, permissions to access the protected resource is given. Finally, line 10 indicates that when an IllegalStateException occurs, then necessarily owningThread != current_thread. This happens if a thread that is not the owner of the lock calls this method.

The releaseLock() method simply calls dbModel.delete(), which was described in Section 7.3.3.3. If no rows are deleted, a LockLostException is thrown.

The unlock() method releases a held lock. The specification, given in Listing 36, indicates that the lock must be held (line 6), has its reentrancy level decreased by one upon each call (line 8) and upon the final release, the permissions to access the shared resource are rescinded (line 7). The implementation decrements the instanceLockCount and heldCount fields, and if the lock is released for the final time (such that instanceLockCount == 0 after the decrement operation), the releaseLock() method is called to delete the corresponding database row.

```
[ 1]/*@
[ 2] given int current_thread;
[ 3] context  common() ** lockset_part(current_thread);
[ 4] context  dbModel.common() ** dbModel.perm_all(lockName);
[ 5] context  initialised(T);
[ 6] requires heldCount > 0;
[ 7] requires heldCount == 1 ==> subject.inv();
[ 8] ensures  heldCount == \old(heldCount) - 1;
[ 9] signals  (IllegalStateException e) owningThread != current_thread;
[10] // should not occur if precondition is satisfied
[11] signals  (LockLostException e) false;
[12] // Can't occur, since missingOk parameter is true
[13]@*/
[14]public void unlock() {
```

Listing 36: The specification of the unlock method

7.3.5 DbNamedReadWriteLock

The DbNamedReadWriteLock class, first introduced in intermediate 2, was changed. The read and write lock objects contained within no longer synchronize using a field of the DbNamedReadWriteLock class, but instead rely on the database to perform the synchronization. Because of this, the field was removed and all related permissions were also removed. Other than this, the description given in Section 7.2.1 is still accurate.

7.3.6 DbLockManager

The DbLockManager class, earlier described in Sections 7.1.5 (intermediate 1) and Section 7.2.3 (intermediate 2), is significantly altered in intermediate 3. The main reason for this is that in intermediate 3, lock objects are no longer shared between threads. Instead, each thread has a reference to an array of pairs, similar to how in intermediate 1 and 2, one such array existed.

In this section, the specification, including its fields, predicates and methods, are described in full, as this gives a clear overview of the specification.

Before diving into the specification for the DbLockManager class, the inner LockInstanceArray class is described, which holds these pairs.

7.3.6.1 LockInstanceArray

As stated before, the LockInstanceArray class holds an array of pairs. These pairs may be null. The first half of the array is reserved for write locks, and the second half is reserved for read locks, such that pairs[lockName] is a write lock with name lockName, and pairs[lockName + maxLockName] is a read lock with name lockName. The implementation and specification for this class is given in Listing 37.

```
class LockInstanceArray {
 final Pair[] pairs;

 /*@
  ensures pairs != null ** pairs.length == maxLockName * 2;
  ensures Perm(pairs[*], 1\2);
  ensures (\forall int i = 0 .. pairs.length; {:pairs[i]:} == null);
 @*/
 LockInstanceArray(int maxLockName) {
  this.pairs = new Pair[maxLockName * 2];
 }
}
```

Listing 37: The specification and implementation of the LockInstanceArray class

7.3.6.2 Fields

- final DbModel dbModel – This is the reference to the database model.
- final int maxLockName – This is the upper bound for the lock name, such that 0 <= lockName < maxLockName.
- final int num_threads – This is the upper bound for the thread identifiers, such that 0 <= threadId < num_threads.
- final LockInstanceArray[] lockInstanceArray – This is an array which is indexed by thread identifier. It has the length num_threads. Each thread has its own LockInstanceArray instance.
- boolean isStopped – The flag which indicates whether this DbLockManager is active. A value of true indicates that no new locks can be created, but existing locks may continue operation until deleted through destroyLock().
- ghost final int T – A ghost field with the value num_threads. Exists to simplify specification.
- ghost final int maxLockIndex – A ghost field with the value 2 * maxLockName. This is simply a shorthand form of this expression, meant to aid VerCors in determining that this value never changes.
- ghost int next_uuid – In the database model, each lock instance is considered to have a unique UUID. This field ensures that all lock instances created by a DbLockManager have unique UUIDs. The uniqueness of UUIDs between different DbLockManager objects is assumed.

7.3.6.3 Predicates

The lock_invariant, which is given in Listing 39, describes valid state for the DbLockManager object.

Permissions to access next_uuid is given (line 2), and the initialisation of the final fields is specified (lines 2-5). There are a number of forall quantifiers, which operate on the lockInstanceArray field. These indicate the following properties:

- Permission to read each element is present (line 7).
- All elements are non-null (line 8).
- All elements are unique, as are their fields (lines 10-11).

- Each pairs array is properly initialised, and permissions to access their elements is available (lines 13, 15).
- All non-null pairs are unique (line 18).
- Each non-null pair is properly initialised (lines 22-26).
- Every pair has unique fields (lines 36-38).
- Permission to access the reentrancy level for each pair is available, has a valid value and the corresponding lock object is in a valid state (lines 46-49).

Together, these properties ensure that the DbLockManager is in a valid state.

The general() predicate contains the same information as lines 2-5 from the lock invariant, with the addition that the DbLockManager has committed its lock invariant.

The common() predicate contains general(), and the information from lines 6-18 of the lock invariant. Furthermore, write permission over the isStopped field is specified. This predicate is used as the postcondition for the constructors of the DbLockManager class. Other than there, the common_part predicate is used in createLock and destroyLock calls.

Finally, the common_part(int current_thread, int i) predicate, given in Listing 38, contains part of the information in the lock_invariant, namely the permissions over the specific array element that belongs to the given thread identifier and pair index. Line 2 indicates that the given thread identifier must be valid. Lines 3-4 indicate that permissions are available to read out the element, and that the element is non-null.

Line 6 indicates that the pairs array is properly initialised, and permission to read out the correct element is available. If this element is not null, the element has a properly initialised reentrancy level (greater than 0, otherwise the pair would be null), as indicated in lines 10-11. Lines 14-18 indicate that the corresponding lock object must also be initialised, and the pair's index must correspond to the lock's name and type. Line 17 indicates that the lock type corresponds to the pair's index. Finally, line 22 indicates read permission for the isStopped field.

```
[ 1]inline resource common_part(int current_thread, int i) =
[ 2] 0 <= current_thread ** current_thread < T **
[ 3] Perm(lockInstanceArray[current_thread], 1\4) **
[ 4] lockInstanceArray[current_thread] != null **
[ 5] (\let Pair[] pairs = lockInstanceArray[current_thread].pairs;
[ 6]  \array(pairs, maxLockIndex) ** Perm(pairs[i], 1\2) **
[ 7]  (\let Pair pair = pairs[i];
[ 8]   (pair != null ==>
[ 9]    (\let AtomicInteger first = pair.getFirst();
[10]     first != null ** committed(first) **
[11]     Perm(first.val, 1\2) ** first.val > 0
[12]    ) **
[13]    (\let DbNamedLock second = pair.getSecond();
[14]     second != null ** committed(second) **
[15]     (\let boolean secondType = second.getType();
[16]      i == second.getName() + (secondType ? 0 : maxLockName) **
[17]      (secondType == (i < maxLockName))
[18]     )
[19]    )
[20]   )
[21]  )
[22] ) ** Value(isStopped);
```

Listing 38: The specification of the common_part predicate

```
[ 1]public /*@ lock_invariant
[ 2] Perm(next_uuid, 1\2) ** T == num_threads ** num_threads == 5 **
[ 3] maxLockIndex == 2 * maxLockName ** maxLockName == 10 **
[ 4] dbModel != null ** dbModel.maxLockName == maxLockName **
[ 5] \array(lockInstanceArray, T) **
[ 6] (\forall* int i = 0 .. T;
[ 7]  Perm(lockInstanceArray[i], 1\2) **
[ 8]  {:lockInstanceArray[i]:} != null) **
[ 9] (\forall int i = 0 .. T, int j = 0 .. i;
[10]  {:lockInstanceArray[i]:} != {:lockInstanceArray[j]:} &&
[11]  lockInstanceArray[i].pairs != lockInstanceArray[j].pairs) **
[12] (\forall int i = 0 .. T;
[13]  \array({:lockInstanceArray[i].pairs:}, maxLockIndex)) **
[14] (\forall* int i = 0 .. T, int j = 0 .. maxLockIndex;
[15]  Perm({:lockInstanceArray[i].pairs[j]:}, 1\2)) **
[16] (\forall* int i = 0 .. T, int j = 0 .. maxLockIndex, int k = 0 .. j;
[17]  lockInstanceArray[i].pairs[j] != null;
[18]  lockInstanceArray[i].pairs[j] != lockInstanceArray[i].pairs[k]) **
[19] (\forall* int i = 0 .. T, int j = 0 .. maxLockIndex;
[20]  lockInstanceArray[i].pairs[j] != null;
[21]  (\let Pair pi = {:lockInstanceArray[i].pairs[j]:};
[22]   pi.getFirst() != null ** pi.getSecond() != null **
[23]   pi.getSecond().uuid < next_uuid **
[24]   j == pi.getSecond().getName() +
[25]     (pi.getSecond().getType() ? 0 : maxLockName) **
[26]   pi.getSecond().getType() == (j < maxLockName)
[27]  )
[28] ) **
[29] (\forall* int i = 0 .. T, int j = 0 .. maxLockIndex, int k = 0 .. T,
[30]  int l = 0 .. maxLockIndex; lockInstanceArray[i].pairs[j] != null;
[31]  (\let Pair pi = {:lockInstanceArray[i].pairs[j]:};
[32]   (\let Pair pk = {:lockInstanceArray[k].pairs[l]:};
[33]    ((i != k || j != l) ==> (
[34]     pi != pk **
[35]     (pk != null ==> (
[36]      pi.getFirst() != pk.getFirst() **
[37]      pi.getSecond() != pk.getSecond() **
[38]      pi.getSecond().uuid != pk.getSecond().uuid
[39]     ))
[40]    ))
[41]   )
[42]  )
[43] ) **
[44] (\forall* int i = 0 .. T, int j = 0 .. maxLockIndex;
[45]  lockInstanceArray[i].pairs[j] != null;
[46]  Perm(lockInstanceArray[i].pairs[j].getFirst().val, 1\2) **
[47]  lockInstanceArray[i].pairs[j].getFirst().val > 0 **
[48]  (\let DbNamedLock second = lockInstanceArray[i].pairs[j].getSecond();
[49]   second.common() ** second.initialised(T)
[50]  )
[51] );
[52]@*/ class DbLockManager implements LockManager {
```
Listing 39: The lock invariant of the DbLockManager class

7.3.6.4 Methods

The specification for the methods of the DbLockManager class has changed relatively little. The flowchart for the createLock() method in intermediate 1, given in Figure 6, is still accurate, with a

few changes: rather than indexing into pairs[lockName], the indexing happens on lockInstanceArray[current_thread].pairs[lockKey], where lockKey is the corresponding index of the read or write lock (lockName for write locks, lockName + maxLockName).

The specification is conceptually very similar, but syntactically looks a lot less clean. This is because read and write locks are both in the same array, and the field names are quite long. An expression like

```
ensures pairs[lockName] != null;
```

is replaced with

```
ensures lockInstanceArray[current_thread].pairs[lockKey(lockName,
lockType, maxLockName)] != null;
```

The lockKey(lockName, lockType, maxLockName) function call here is a pure ghost method, which takes in these three parameters and returns the appropriate index. This function aids VerCors in determining that the value of this lockKey is the same across all pre- and postconditions.

This mixing of read and write locks is addressed in intermediate 4, by splitting the read- and write locks into two separate arrays: lockInstanceArray[current_thread].readPairs[lockName] and lockInstanceArray[current_thread].writePairs[lockName]. This is still quite a bit longer than pairs[lockName], but a lot simpler to read regardless.

The destroyLock() method had similar changes to the createLock() method described above, and is therefore left out for brevity. Figure 7 shows the flowchart for the implementation, and Listing 25 shows its specification.

7.4 Intermediate 4

Intermediate 4 introduced the failFastLock. Since this introduced quite a lot of complexity in terms of specification, this greatly affected verification performance. Because of this, some changes had to be made to the specification in order to ensure that verification could complete within a reasonable timespan.

The specification for this intermediate did not get fully finished, as refactoring inline predicates into opaque predicates turned out to be a lot more complex and time-consuming than expected.

In the end, intermediate 4 did not complete with a successful verification, though significant steps were made towards a working result. This section details the changes made. The limitations and challenges encountered while constructing this specification are glossed over here, but are detailed in Chapter 8.

7.4.1 ListDbRow

The common_arr predicate (see Section 7.3.2.2) contains a lot of state regarding elements of the underlying array. This state is encapsulated in forall quantifiers, which greatly affect prover performance. Because of this, the decision was made to refactor this predicate into an opaque predicate (see Section 2.8).

An opaque predicate requires that all fields accessed in the predicate have their access permissions also stored within the predicate. Since the permissions were specified in common(), the common() predicate had its permission value halved and was duplicated into the common_arr() predicate. This resulted in the specification given in Listing 40.

```
inline resource common() =
 Perm(size, 1\2) ** Perm(uuidSeq, 1\2) **
 Perm(arr, 1\2) ** arr != null ** arr.length > 0 **
 size == |uuidSeq| **
 0 <= size ** size <= arr.length
;

inline resource common_arr() =
 common() **
 Perm(arr[*], write) **
 (\forall int i = 0 .. size; {:arr[i]:} != null) **
 (\forall int i = 0 .. size, int j = 0 .. size; i != j;
  {:arr[i]:} != {:arr[j]:}) **
 (\forall int i = 0 .. size, int j = 0 .. size; i != j;
  {:arr[i].uuid:} != {:arr[j].uuid:}) **
 (\forall int i = 0 .. size; {:uuidSeq[i]:} == {:arr[i].uuid:}) **
 (\forall int i = size .. arr.length; {:arr[i]:} == null);

resource common_arr(frac x) = [x]common_arr();
```

Listing 40: The specification of the common and common_arr predicates

Note that the common_arr predicate is still an inline resource, but there is an opaque version: common_arr(frac x), which takes in a fraction, and opaquely stores the inline version of common_arr(). This was done due to a limitation in VerCors, where scaled opaque predicates could not be folded and unfolded. This was later fixed by the VerCors team, but the specification was not altered again to reflect this.

This did have some implications for the rest of the specification. Wherever an array element of arr was accessed in the specification, the opaque predicate had to be temporarily unfolded in order to show the prover that the permissions are in fact available.

This resulted in a refactoring of the following kind:

```
(\forall int i = size .. arr.length; arr[i] == null)

(\forall int i = size .. arr.length; \unfolding common_arr(1) \in arr[i] ==
null)
```

Where the line above previously sufficed with an inline version of common_arr(), now, the common_arr(1) predicate has to be temporarily unfolded in order to gain access to arr[i].

7.4.2 DbModel

The lock invariant of the DbModel class stores not only permission, but also information on the state. This information is stored within many forall quantifiers. In order to improve prover performance, many of these quantifiers were put inside an opaque predicate. This resulted in a lock invariant where the permissions are stored in an inline predicate (inv_perms) and the state-specific information is stored in an opaque predicate (inv_state). Together, these predicates store the same information as the original lock invariant, which was given in Listing 31. Here, line 1-10 is stored in inv_perms(), and lines 11-30 are stored in inv_state(). As with the refactoring applied in the ListDbRow class, inv_perms() had its permissions lowered by half, and was also included in the inv_state() predicate in order to ensure that permissions to access each field was available.

Since some of the specification of DbModel also uses the common_arr() predicate from ListDbRow, these references to common_arr were replaced with the opaque variant described in Section 7.4.1 and the predicate was unfolded as required.

7.4.3 MyReentrantReadWriteLock

Since intermediate 4 introduces the failFastLock, this lock had to be implemented or taken from somewhere. In the implementation provided by BetterBe, Java's ReentrantReadWriteLock was used for this. However, since no specification existed for this class, a custom implementation was written and specified for the purpose of this thesis.

The implementation of this lock is similar to the DbNamedReadWriteLock implementation used in Section 7.2.1, though there are a few differences.

7.4.3.1 Fields

- AtomicInteger readWriteFlag – This field indicates the current lock state. It can hold the values -1 (unlocked), 0 (read locked) or 1 (write locked).
- AtomicInteger owner – This field indicates the thread identifier of the owner of the lock. This can be -1 (unlocked or read locked) or >= 0 (write locked).
- AtomicInteger count – This field indicates the reentrancy level of the lock. This field has a non-negative value.
- ReadLock readLock – The internal read lock, which synchronizes on the MyReentrantReadWriteLock object
- WriteLock writeLock – The internal write lock, which synchronizes on the MyReentrantReadWriteLock object

7.4.3.2 Predicates

Two predicates are present. The general() predicate ensures that all fields are initialised and the readLock and writeLock synchronize using this object. The common() predicate contains 1\2 permissions to access the value of the readWriteFlag, count and owner fields. Furthermore, it establishes the bounds for the values of each of these fields and how they relate to each other.

The lock_invariant holds the other half of the permissions for the values of readWriteFlag, count and owner fields, and again establishes the connection between the values of these fields.

7.4.3.3 ReadLock

The local ReadLock class contains methods to obtain and release the lock in read mode.

The tryLock() method attempts to set the readWriteFlag's value from -1 (unlocked) to 0 (read locked). If this succeeds, or the value was already 0, then the count's value is incremented and true is returned. Otherwise, false is returned.

The unlock() method requires that the ReadWriteLock is held in read mode. It decrements count.val. If the count reaches 0, the readWriteFlag is set to -1 (unlocked). This method requires that the old value of readWriteFlag.val is 0 (read locked). Note that no attempt is made to check that this thread was actually holding this read lock. If the specification is respected, this is no problem.

However, if the MyReentrantReadWriteLock class is used as a standalone lock, and a thread not holding a read lock calls unlock(), incorrect behaviour will occur. This can be solved by applying a similar bag<int> holders as used in intermediate 2 in Section 7.2.2.1. Then, each holder is uniquely identified by its thread identifier. Due to time constraints, this was not applied, since the specification is correct when using the MyReentrantReadWriteLock in the context of the failFastLock.

Finally, the lock() method repeatedly calls tryLock() until it succeeds.

7.4.3.4 WriteLock

The local WriteLock class contains methods to obtain and release the lock in write mode.

The tryLock() method attempts to set the readWriteFlag's value from -1 (unlocked) to 1 (write locked). If this succeeds, or the flag was already set to 1 and this thread's identifier is stored in the owner.val field, the lock is (re-)obtained. The owner's value is set to this thread's identifier and the count is incremented. Otherwise, false is returned.

The unlock() method requires that the readWriteFlag's value is 1 and this thread is the owner. It decrements the count. If this value reaches 0, owner is set to -1 (unlocked or read locked) and readWriteFlag is set to -1 (unlocked).

Finally, the lock() method attempts to tryLock() until it succeeds.

7.4.4 Pair

The Pair class also received a few changes. Since intermediate 4 introduced a failFastLock, a place to store this failFastLock was required. The Pair class was used for this, as it was in the implementation provided by BetterBe.

This was done by adding a new field to the Pair class, named MyReentrantReadWriteLock failFastLock, and renaming the DbNamedLock second field to DbNamedLock dbNamedLock. At every place where pair.getSecond() was called, this was replaced with getDbNamedLock(). This ensures that no breaking changes were introduced.

In the original constructor, the dbNamedLock field was taken as a parameter and was set. The failFastLock field was set to null in this constructor.

A new constructor was added, which instead initialises the failFastLock field, and sets the dbNamedLock field to null. This results in a Pair class where the first (AtomicInteger) is always non-null, and either the dbNamedLock or the failFastLock field is null, but not both.

7.4.5 DbNamedLock

Section 7.3.4 shows the DbNamedLock class as it was implemented in intermediate 3.

In intermediate 4, some changes were made to introduce the usage of the failFastLock in lock operations.

A field was added to store a reference to the failFastLock object. This field is predictably called MyReentrantReadWriteLock failFastLock.

The lock_invariant was slightly changed as compared to the invariant given in Listing 34, namely lines 8-9 were removed. They were a redundant leftover from intermediate 2.

Some state regarding the failFastLock was added to the common() and lockset_part() predicates. The permission for heldCount which was previously stored in lockset_part() was moved to common(). Initialisation and valid state for the failFastLock were also stored in common().

The constructor received an additional parameter to initialise the failFastLock.

The tryLock() method is very similar to before. A flowchart of the version used in intermediate 3 is given in Figure 9. However, if the lock is not held, rather than immediately calling dbModel.insert(), an attempt is made to lock the failFastLock. If this fails, false is returned and no call to the database is made.

Furthermore, if the failFastLock.tryLock() call succeeds, but the dbModel.insert() call returns 0, the failFastLock is immediately unlocked.

In intermediate 3, the final action upon unlocking a lock for the last time is calling releaseLock(), as described in Section 7.3.4.3. This deletes the database row corresponding to this lock entry. In intermediate 4, after performing this operation, the failFastLock is unlocked. This ensures that even if another thread attempts to lock while the current lock is in the process of being unlocked, the failFastLock.tryLock() call will still return false, until the database row has been deleted.

7.4.6 DbLockManager

The failFastLock introduced in this intermediate is stored in the DbLockManager. Because of this, some changes had to be made to the fields, some new predicates had to be added and the specification of some of the methods had to be changed.

7.4.6.1 Changes in fields

- Pair[] failFastLocks – This field stores all the failFastLocks – one per lock name. This means that any one failFastLock is shared between up to two lock objects – a read lock and a write lock with the same lock name.
- LockInstanceArray[] lockInstanceArray – The pairs[] array stored within was split into two arrays: readPairs and writePairs. This simplifies the indexing of these arrays, resulting in performance improvements of the solver. Furthermore, it increases readability, as seeing that pairs[i + maxLockName] is a write lock is less intuitive than seeing that writePairs[i] is a write lock.
- The maxLockIndex field was removed, as it is no longer used.

7.4.6.2 Changes in methods

The administration done by the DbLockManager with regards to failFastLocks is very similar to that of the DbNamedLock objects. They are created and deleted as needed.

When createLock(lockName) is called, and no failFastLock exists yet, a failFastLock is created, stored in a pair and put in failFastLocks[lockName]. This failFastLock is then passed along to the constructor of DbNamedLock. If a failFastLock already existed, the count associated with the pair storing this failFastLock is incremented and the existing failFastLock is passed along to the constructor of DbNamedLock instead.

Similarly, when destroyLock() is called, the count corresponding to the failFastLock's pair is decremented. When this count reaches 0, both the read and write lock which held a reference to the failFastLock are deleted, and the failFastLock is also set to null.

7.4.6.3 Changes in predicates

There are quite a few large changes made to the predicates of the DbLockManager class. Most of the changes were done in order to refactor the inline predicates present into opaque counterparts.

Some new predicates and boolean functions were added to link the state of the failFastLocks field to that of the DbNamedLocks. Furthermore, some inline predicates were split and reorganized for readability and to make transformations into opaque predicates simpler.

Because of the many changes made, the predicates and boolean functions used are described here.

The generalPred predicate contains most of the information previously present in general(). The only change is that general() contained extra checks for this != null ** committed(this). Since generalPred() is later used in the lock invariant, this was removed.

```
inline resource generalPred() =
 Perm(next_uuid, 1\2) **
 T == num_threads ** num_threads == 5 **
 maxLockName == 10 **
 dbModel != null ** dbModel.maxLockName == maxLockName **
 \array(lockInstanceArray, T)
;
```

Listing 41: The generalPred predicate

The pairPerms() predicate contains the permissions required to access all elements of lockInstanceArray, initialisation over its readPairs and writePairs arrays and permissions to access each of the array elements of these two fields. The specification for this predicate is given in Listing 42. In intermediate 3, these permissions were present both in the lock invariant and in the common() predicate.

```
inline resource pairPerms() =
 (\forall* int i = 0 .. T; Value(lockInstanceArray[i])) **
 (\forall int i = 0 .. T; lockInstanceArray[i] != null) **
 // injectivity of lockInstanceArray and their pairs
 (\forall* int i = 0 .. T, int j = 0 .. i;
  lockInstanceArray[i] != lockInstanceArray[j] **
  lockInstanceArray[i].readPairs != lockInstanceArray[j].readPairs **
  lockInstanceArray[i].writePairs != lockInstanceArray[j].writePairs
 ) **
 // each pairs array is non-null and has length maxLockName
 (\forall int i = 0 .. T;
  \array({:lockInstanceArray[i].readPairs:}, maxLockName) &&
  \array({:lockInstanceArray[i].writePairs:}, maxLockName)
 ) **
 // We have read permissions to access each array element
 (\forall* int i = 0 .. T, int k = 0 .. maxLockName;
  Perm({:lockInstanceArray[i].readPairs[k]:}, 1\2) **
  Perm({:lockInstanceArray[i].writePairs[k]:}, 1\2)
 )
;
```

Listing 42: The pairPerms predicate

The pairPermsOpaque() predicate is given below. It is an opaque resource (the inline keyword is missing), meaning it has to be unfolded to access the information inside.

```
resource pairPermsOpaque() = generalPred() ** pairPerms();
```

The pairContents() boolean function contains information on the state of the pairs arrays. It requires pairPermsOpaque() to be held, and indicates which part of the state holds when the function returns true. The specification of this boolean function is given in Listing 43. In intermediate 3, the information stored in this function was stored inside of the lock invariant (see Listing 39, lines 19-43). Note that the forall quantifiers here are doubled, as separate quantifiers are needed over the readPairs and writePairs fields.

Note also that rather than directly accessing lockInstanceArray[i].readPairs[j], a boolean function readPair(i, j) is called. This is a boolean function which simply returns lockInstanceArray[i].readPairs[j] and locally unfolds the pairPermsOpaque() predicate to allow this to be done.

```
requires [1\1024]pairPermsOpaque();
pure boolean pairContents() =
 // Each non-null Pair has non-null first and lock instance
 // the lock's uuid is lower than next_uuid (that seems unnecessary)
 // The index corresponds to lockKey()
 // The type corresponds to the array it's in
 (\forall int i = 0 .. T, int j = 0 .. maxLockName; readPair(i, j) != null;
  (\let Pair pi = {:readPair(i, j):};
   pi.getFirst() != null && pi.getDbNamedLock() != null &&
   pi.getDbNamedLock().uuid < (\Unfolding [1\1024]pairPermsOpaque() \in
next_uuid) &&
   j == pi.getDbNamedLock().getName() &&
   !pi.getDbNamedLock().getType()
  )
 ) &&
 (\forall int i = 0 .. T, int j = 0 .. maxLockName; writePair(i, j) !=
null;
  (\let Pair pi = {:writePair(i, j):};
   pi.getFirst() != null && pi.getDbNamedLock() != null &&
   pi.getDbNamedLock().uuid < (\Unfolding [1\1024]pairPermsOpaque() \in
next_uuid) &&
   j == pi.getDbNamedLock().getName() &&
   pi.getDbNamedLock().getType()
  )
 ) &&
 // Each array element is unique, as are its fields.
 (\forall int i = 0 .. T, int j = 0 .. maxLockName, int k = 0 .. T, int l =
0 .. maxLockName; readPair(i, j) != null;
  (\let Pair pi = {:readPair(i, j):};
   (\let Pair pk = {:readPair(k, l):};
    ((i != k || j != l) ==> (
     pi != pk &&
     (pk != null ==> (
      pi.getFirst() != pk.getFirst() &&
      pi.getDbNamedLock() != pk.getDbNamedLock() &&
      pi.getDbNamedLock().uuid != pk.getDbNamedLock().uuid
     ))
    ))
   )
  )
 ) &&
 (\forall int i = 0 .. T, int j = 0 .. maxLockName, int k = 0 .. T, int l =
0 .. maxLockName; writePair(i, j) != null;
  (\let Pair pi = {:writePair(i, j):};
   (\let Pair pk = {:writePair(k, l):};
    ((i != k || j != l) ==> (
     pi != pk &&
     (pk != null ==> (
      pi.getFirst() != pk.getFirst() &&
      pi.getDbNamedLock() != pk.getDbNamedLock() &&
      pi.getDbNamedLock().uuid != pk.getDbNamedLock().uuid
     ))
    ))
   )
  )
 )
;
```
Listing 43: The specification of the pairContents() boolean function

The benefit of using a boolean functions over an opaque predicate is the following. When VerCors verifies a boolean function, only then will the contents of the function be analysed. The unfolding of

the pairPermsOpaque predicate is performed, and it is checked that this function is well-defined. When encountering the usage of the boolean function, the unfolding of the pairPermsOpaque predicate is no longer done, improving performance by a lot.

The limitation induced by this is that after unfolding the required opaque predicate, the boolean function can no longer be called. A workaround was developed for this.

First, half of the pairPermsOpaque() predicate was unfolded. This allows for asserting pairContents() as well as the contents thereof after replacing readPair(i,j) with lockInstanceArray[i].readPairs[j]. Then, the other half of the pairPermsOpaque() predicate was unfolded. This means that the contents of the boolean function are still available, in slightly altered format (namely without the readPair(i, j) and writePair(i, j) calls). However, re-establishing pairContents after folding pairPermsOpaque again failed. Due to time constraints, this remained unsolved.

The lockInstancePerms() predicate contains the final part of the lock_invariant. It is an inline resource. The only difference with the version presented in intermediate 3 (Listing 39 in lines 44-50) is that the forall quantifier is split into two: one for lockInstanceArray[i].readPairs, and one for lockInstanceArray[i].writePairs.

The pairPermsOpaque(), pairContents() and lockInstancePerms() predicates and function are combined into an opaque predicate lockInstancePermsOpaque().

```
resource lockInstancePermsOpaque() = pairPermsOpaque() ** pairContents() **
lockInstancePerms();
```

This opaque predicate stores the entirety of the lock invariant as it was in intermediate 3.

However, as stated at the start of this section, intermediate 4 introduces the failFastLock. The next predicates and functions describe valid state for this field and describes how its state relates to the other fields.

The failFastInjectivity() boolean function requires a properly initialised failFastLocks field, as well as read permissions over each of its elements. It establishes injectivity and proper initialisation of each of the non-null failFastLocks elements. That is, the following properties are described for each non-null Pair failFastLocks[i]:

- Its ‘first’ field is non-null
- Its failFastLock field is non-null and failFastLock.general() holds (see Section 7.4.3.2)
- The pair is unique with regards to the other pairs
- All of the fields of the failFastLock are unique with regards to other non-null pairs

The failFastPerms() inline predicate specifies 1\2 permission over failFastLocks[i].first.val (for each i). Furthermore, it establishes that half of the failFastLock’s common predicate is available.

The failFastInjectivity() and failFastPerms() function and predicate are combined into an opaque predicate:

```
resource failFastPermsOpaque() = this != null ** \array(failFastLocks,
maxLockName) ** Perm(failFastLocks[*], 1\4) ** failFastInjectivity() **
failFastPerms();
```

This predicate ensures proper initialisation of the failFastLocks. Still, no link is made between the failFastLocks field and the other fields. This is done by the next boolean function: failFastNull, which is given in Listing 44.

```
requires [1\1024]failFastPermsOpaque();
pure boolean failFastNull() =
  // A failFastLock exists iff either the readPair or writePair exists
  (\forall int i = 0 .. maxLockName, int j = 0 .. T;
   ({:readPairLi(j, i):} != null || {:writePairLi(j, i):} != null) ==
    {:failFast(i):} != null
  )
;
```

Listing 44: The specification of the failFastNull() boolean function

The failFastNull function establishes that whenever either the read- or write lock is initialised, so is the failFastLock. Also, whenever both read and write locks are null, so is the failFastLock.

Note that here, the readPairLi() function is used. This is a variant on the readPair() function, which unfolds lockInstancePermsOpaque() to obtain pairPermsOpaque() and calls readPair(). Furthermore, note that the failFast(i) function is used. This is a function which unfolds failFastPermsOpaque() and returns failFastLocks[i].

There is still no connection between the actual state of the non-null failFastLock and that of the readPairs' or writePairs' lock state. This is where the test() boolean function comes into play. This function links the state of the failFastLocks to that of the lock instances under this lock manager. The

The specification for the test() boolean function is given in Listing 46. It checks that for a specific lock name, the following properties hold:

- failFastNull() (line 5)
- When all read- or write locks are unlocked, so is the failFastLock (lines 6-28)
- When a lock is read locked, so is the failFastLock (lines 29-37)
- When a lock is write locked, so is the failFastLock (lines 38-48)

The test_all boolean function holds true when test(lockName) holds for each valid lock name. This ensures that, for each failFastLock, its state is consistent with that of the corresponding lock instances.

Finally, all these predicates and boolean functions are combined to describe valid state. This is done in the newCommon predicate, given in Listing 45.

```
resource newCommon() =
 lockInstancePermsOpaque() **
 \array(failFastLocks, maxLockName) **
 Perm(failFastLocks[*], 1\4) **
 failFastPermsOpaque() **
 failFastNull() **
 test_all()
;
```

Listing 45: The specification of the newCommon() predicate

The lock_invariant is replaced with the newCommon() predicate. This predicate holds exactly half permissions for each field of the DbLockManager class and its fields.

```
[ 1]requires 0 <= lockName && lockName < maxLockName;
[ 2]requires [1\1024]failFastPermsOpaque();
[ 3]requires [1\1024]lockInstancePermsOpaque();
[ 4]pure boolean test(int lockName) =
[ 5] failFastNull &&
[ 6] (
[ 7]  // All pairs are either null or contain an unlocked lock.
[ 8]  (
[ 9]   (\forall int t = 0 .. T;
[10]    (
[11]     // The read lock is unlocked or does not exist
[12]     ({:readPairLi(t, lockName):} == null) ||
[13]     {:readHeldCount(t, lockName):} == 0
[14]    ) && (
[15]     // The write lock is unlocked or does not exist
[16]     ({:writePairLi(t, lockName):} == null) ||
[17]     {:writeHeldCount(t, lockName):} == 0
[18]    )
[19]   ) &&
[20]   // At least one non-null pair exists, so failFast(lockName) != null
[21]   (\exists int t = 0 .. T; {:readPairLi(t, lockName):} != null ||
[22]    {:writePairLi(t, lockName):} != null
[23]   )
[24]  ) ==> failFast(lockName) != null &&
[25]   readWriteVal(lockName) == -1 &&
[26]   ownerVal(lockName) == -1 &&
[27]   countVal(lockName) == -1
[28] ) &&
[29] (
[30]  // At least one read lock exists which is locked.
[31]  (\exists int t = 0 .. T; readPairLi(t, lockName) != null;
[32]   {:readHeldCount(t, lockName):} > 0
[33]  ) ==> failFast(lockName) != null &&
[34]   readWriteVal(lockName) ==  0 &&
[35]   ownerVal(lockName) == -1 &&
[36]   countVal(lockName)  >  0
[37] ) &&
[38] (
[39]  // A write lock exists which is locked. Furthermore,
[40]  // failFastLocks[lockName] also has matching count and owner fields
[41]  failFast(lockName) != null &&
[42]  (\exists int t = 0 .. T; writePairLi(t, lockName) != null;
[43]   {:writeHeldCount(t, lockName):} > 0 &&
[44]   {:writePairLi(t, lockName).getDbNamedLock().owningThread:} ==
[45]    ownerVal(lockName) &&
[46]   {:writeHeldCount(t, lockName):} == countVal(lockName)
[47]  ) ==> readWriteVal(lockName) ==  1
[48] )
[49];
```
Listing 46: The specification of the test() boolean function

7.4.6.4 Further work

The structure of the fields, methods and predicates of the DbLockManager class has been discussed. However, no indication was made on how these predicates are established and used. Due to the unforeseen complexity of writing the specification for intermediate 3 and 4, there was not enough time left to reach a point where the verification of intermediate 4 terminated successfully.

The exact limitations of this intermediate are further discussed in Section 8.3.

8 Limitations

8.1 Intermediate 2

8.1.1 Usage of the DbNamedReadWriteLock

When a read lock is created through DbNamedReadWriteLock, the lock must be either null in DbLockManager, or already have a read lock created by another DbNamedReadWriteLock object. When then creating a write lock, the corresponding read lock must be created by the same DbNamedReadWriteLock object, otherwise the states of the read lock and write lock are unable to synchronize and mixed read/write access is possible.

8.1.2 Subject's invariant for read locks

When a read lock is granted for the first time, the subject's invariant is granted (see Listing 26). However, this invariant is supposed to represent write access to the protected resource. Therefore, granting the subject's invariant is not correctly specified.

A solution was theorized, though there are still some hurdles to be considered, before this can be fully implemented. This solution included splitting subject.inv() into two predicates, writeInv() and readInv(). Upon obtaining a lock in write mode for the first time, the full writeInv() is given out, identically to how inv() is currently given out. However, for read locks, this is a lot more complex.

In principle, a full readInv() is available. However, because this is a read-only predicate, it suffices to grant each reader only a fraction of this readInv(). This is the core idea behind this solution.

In order to facilitate granting a fractional part of the readInv(), we require some method of keeping track how much permission should be given out. A ghost field frac[] givenPermissions should keep track, for each thread, how large of a fraction of readInv() was given out to the thread. Upon encountering a thread which did not yet hold the read lock, a new fraction readInv() is given out. The n-th readInv() predicate given out will receive [$1\backslash 2^n$]readInv(), and this fraction is stored in the givenPermissions array. When this same thread releases the final readLock, this same fraction is returned.

This does require some (unsolved) bookkeeping regarding the total amount of readInv() given out. In particular, the total fraction of given out readInv() should never exceed 1, and the amount of readInv() 'remaining' in the subject should always equal 1 minus the sum of the fractions inside the givenPermissions array. These are not obstacles that cannot be overcome, however, due to time constraints, this was not solved, also not in later intermediates.

8.1.3 Shared permission for DbNamedReadWriteLock's lock type flag

The DbNamedReadWriteLock class has a flag which indicates whether the lock is unlocked, locked in read mode or locked in write mode, called readWriteFlag (see Section 7.2.1.1). Section 7.2.2.2 indicates that 1\2 permission to access readWriteFlag.val is stored in the lock invariant of each lock instance that is created through a DbNamedReadWriteLock. However, half of this permission is also stored in the lock invariant of the DbNamedReadWriteLock, and the other half is stored in DbNamedReadWriteLock.common(). This indicates that 1.5 permission to access readWriteFlag.val is specified. This obviously cannot be correct.

A theorized solution to this problem is a similar solution to the limitation discussed in Section 8.1.2. Here, instead of keeping track of a fractional readInv() permission, we keep track of a fractional readWriteFlag.val permission, ensuring that only the first reader to obtain the read lock can obtain a full write permission for readWriteFlag.val in the tryLock() method, and only the last reader to release the read lock can obtain a full write permission for readWriteFlag.val in the unlock() method.

There, a full write permission is needed to alter the value of the readWriteFlag. Other read access does not require the full write access, and therefore could do with a fractional permission.

Again, due to time constraints, this limitation was not addressed in this thesis.

8.1.4 Each lock instance has their own Subject

As the title suggests, each lock instance currently creates a new Subject() object in their constructor. Ideally, this subject is passed along to be shared across all lock instances, at least under a DbLockManager instance. Due to time constraints and the relatively lower importance of this limitation, this was not realised. This limitation is less important, because in the current usage of the system, the subject's invariant is not defined. However, if this specification were to be the basis of a fully fleshed out specification, including a properly specified Subject.inv(), each lock instance effectively guards their own piece of memory, which does not overlap with identically named lock instances (though it should!).

8.2 Intermediate 3

8.2.1 Usages of database model predicates are incorrectly specified

In DbNamedLock.lock(), the dbModel.perm_all(lockName) predicate is held (see Listing 30 and Listing 35). However, only one such perm_all(lockName) predicate may be held over the DbModel instance. In the current specification, it is implicitly assumed that each DbNamedLock instance somehow has access to this perm_all() predicate. This specification successfully verifies, because it is possible to have a single DbNamedLock (with a given lockName), a single DbLockManager and a single DbModel. In this special case, the specification suffices.

In fact, in any single-threaded scenario where a DbModel is initialised, this specification suffices, since the calling thread has the DbModel.perm_all() permissions. However, if these DbNamedLock instances are considered to be in separate threads (this is not checked by VerCors with the current specification), then this 'global' DbModel.perm_all() predicate is no longer present. Therefore, it is not possible to call lock(), unlock() or tryLock(), since the precondition of holding DbModel.perm_all(lockName) is not satisfied.

Some of the permissions present in perm_all(lockName), specifically the permissions over heldArray[lockName] and readWrite[lockName], are used in the specification of the DbModel class to indicate the results of calling the insert and delete methods. Because of this, the permissions present in the perm_all() predicate cannot simply be put into the DbModel's lock invariant, though ideally that is where they should belong in order to solve this limitation.

This limitation is also present in intermediate 4, as no clear solution to this was found.

8.2.2 Missing postcondition in DbLockManager.releaseLock()

The releaseLock() method of the DbLockManager class currently does not specify anything about its behaviour. This could be coupled more strongly to the database behaviour, though it is not strictly necessary for proper system behaviour.

8.2.3 DbLockManager.next_uuid is not unique

Between DbLockManager instances, uuids aren't actually unique with the way the specification was written. In the actual implementation, a random UUID is generated upon creating a Lock instance. In this case, uniqueness of any two generated UUIDs is assumed. In the specification, a supposedly unique uuid is handed down by the DbLockManager. This supposedly unique uuid is incremented each time a new lock is created. This means that if there are two DbLockManager instances with similar (unspecified) starting values for next_uuid, multiple Lock instances may have the same UUID.

The initial value for the next_uuid field was left unspecified. In principle, this means that, while consecutively created Lock instances within the same DbLockManager do have consecutive uuid's, it is vague what the actual value of this next_uuid field is. One DbLockManager instance may start at an initial value of -10.000, whereas the next starts at an initial value of 1.000.000.

Overlap may still exist, but is assumed to never happen (as the probability of two generated UUID's being the same is astronomically small anyway). This assumption is explicitly stated upon the callsite of dbModel.insert(): When a lock is inside tryLock, the following assume statement is encountered:

```
//@ assume !(uuid \in dbModel.lockData[lockName].uuidSeq);
```

This effectively states that the lock's uuid is not currently present within the lock rows for the lock's lock name. Since the assumption that UUID's are unique can be made, this is not a big problem, and this solution suffices.

8.3 Intermediate 4

At the time of writing, unfolding and folding the predicates described in Section 7.4.6.3 has partially been tested, but not all of it. During testing, all methods and their specification were removed to speed up verification. Furthermore, altering the specification of the DbLockManager methods to suit the new predicate and function structure can be left until it is established that the predicates work. In particular, the following things were achieved with regards to the specification structure described in Section 7.4.6:

- When holding newCommon(), it is possible to unfold all of the opaque predicates within and establish write permissions to all fields within.
- The truth value of each boolean function is preserved when unfolding the opaque predicates which they are part of.
- The information contained within each boolean function can be extracted by unfolding half of the opaque predicates indicated in its precondition, asserting the contents of the function, with appropriate replacements for the readPair/writePair/failFast function calls and finally unfolding the second half of the opaque predicates indicated in the precondition.
- pairPermsOpaque() can be folded again after unfolding newCommon() and all opaque predicates within.

The following led to a problem, for which a solution has been theorized, but not yet applied:

- Folding pairContents() after unfolding pairPermsOpaque() fails, as VerCors has trouble establishing that pairContents() == (contents of pairContents()). The theorized solution for this issue is to create a function, which has the contents of pairContents() as a precondition, and has pairContents() as a postcondition. In effect, this helps VerCors establish pairContents() again.

The following points have yet to be done in order to reach a successful verification:

- Re-establish pairContents() after (partially) folding pairPermsOpaque().
- Successfully re-fold lockInstancePermsOpaque().
- Successfully re-fold newCommon().
- Establish newCommon() in the constructor.
- Rewrite the specification of the methods in DbLockManager to work with the new specification structure.

- Restore all other methods which were removed during the testing phase and confirm that this leads to a successful verification.

9 Results

Over the course of this thesis, a number of results have been achieved. These results are summarised in Table 3.

Intermediate #	Description of added features	Number of files	Total lines of code (including specification)	Verification time (in mm:ss)
Intermediate 1	Reentrant lock with BetterBe's code structure	7	516	±01:15
Intermediate 2	Read and write locks	8	847	±04:45
Intermediate 3	Database model	12	1348	±03:45
Intermediate 4	failFastLock optimisation	13	2080	-

Table 3: A summary of achieved results

9.1 Intermediate 1

A ReentrantLock specification has been written with a code structure similar to that of BetterBe's provided implementation.

This specification allows for the creation and deletion of lock instances during runtime, saving memory usage early on, but also further along when the system has been running for a long time. Furthermore, even creation and deletion of threads during runtime is possible, provided they share the same DbLockManager instance.

The specification for intermediate 1 was successfully verified using VerCors.

9.2 Intermediate 2

The specification of intermediate 2 consists of a ReentrantReadWriteLock. Read and write locks synchronize their state through the DbNamedReadWriteLock, and the lock administration was updated accordingly.

The specification for intermediate 2 was successfully verified using VerCors.

9.3 Intermediate 3

The specification of intermediate 3 consists of a distributed ReentrantReadWriteLock specification, which has a code structure similar to that of BetterBe's provided implementation.

A database model was written which faithfully follows the behaviour of the real database system, in which insertion and deletion of rows was supported.

In order to emulate the database behaviour, a custom List implementation was implemented and specified with methods to add, remove and retrieve elements, and a method to copy an array.

The specification for intermediate 3 was successfully verified using VerCors.

While working on the verification, adding in files progressively, limits on performance of the created specification were found. These had to be addressed in intermediate 4, as even more complexity was added in.

9.4 Intermediate 4

The specification of intermediate 4 did not get fully finished. However, significant steps were taken to improve the runtime of the verification and get closer to a working verification.

A DbNamedLock specification was written, in which the failFastLock was used to improve the performance characteristics of this lock implementation. A structure for the specification constructs was created and prototyped, and most of the kinks therein were ironed out. Many of the predicates which ballooned the state space into an unmanageably large size were simplified, improved upon and/or refactored into boolean functions and opaque predicates, which meant that most of the contents thereof were hidden at the points where it mattered.

A specification and implementation was written for a ReentrantReadWriteLock. This object was used as the failFastLock implementation. It was fully specified and verified.

9.5 Miscellaneous

Over the course of writing this thesis, 36 bugs, feature requests and other issue reports have been submitted to the VerCors GitHub page. Of these issues, 28 have been resolved already.

Furthermore, during the course of this thesis and the project preceding it, valuable experience was gained in how injectivity checks influence the verification of software which is not designed with verification in mind. VerC

9.6 Conclusion

While performance was eventually a big bottleneck in the development of this project, the results of the first three intermediates and the performance improvements encountered while switching over to (partially) opaque predicates and pure boolean functions from inline predicates show that there is potential in this approach.

VerCors is a toolset which has some hurdles which are yet to be overcome. However, great leaps are being taken by the VerCors team to overcome these hurdles and make applying VerCors to industrially sized projects more and more feasible.

For now, applying VerCors to such projects is only feasible under several assumptions and limitations, and requires some simplification and rewriting of the code in order to be possible at all. However, since this thesis was started, many of the features which were missing are now implemented at least to some extent, so usability of the tool is rapidly improving.

10 Bibliography

[1] "Utwente-fmt/vercors: The Vercors Verification Toolset for verifying parallel and concurrent software." https://github.com/utwente-fmt/vercors (accessed Mar. 29, 2023).

[2] M. Abadi, C. Flanagan, and S. N. Freund, "Types for Safe Locking: Static Race Detection for Java," *ACM Trans. Program. Lang. Syst.,* vol. 28, no. 2, pp. 207–255, March 2006. [Online]. Available: https://doi.org/10.1145/1119479.1119480.

[3] T. R. Allen and D. A. Padua, "Debugging Fortran on a shared memory machine," January 1987. [Online]. Available: https://www.osti.gov/biblio/5735086.

[4] T. Ball, "The concept of dynamic analysis," *ACM SIGSOFT Software Engineering Notes,* vol. 24, no. 6, pp. 216-234, 1999.

[5] M. Ben-Ari, *Principles of the Spin model checker*. Springer Science & Business Media, 2008.

[6] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn, "The VerCors Tool Set: Verification of Parallel and Concurrent Software," Cham, 2017: Springer International Publishing, in Integrated Formal Methods, pp. 102-110.

[7] T. Bultan, F. Yu, and A. B. Can, "Modular verification of synchronization with reentrant locks," in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, 2010, pp. 59-68.

[8] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent Control with "Readers" and "Writers"," *Commun. ACM,* vol. 14, no. 10, pp. 667–668, October 1971. [Online]. Available: https://doi.org/10.1145/362759.362813.

[9] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv, "Local Reasoning for Storable Locks and Threads," in *Programming Languages and Systems*, Berlin, Heidelberg, Z. Shao, Ed., 2007: Springer Berlin Heidelberg, pp. 19-37.

[10] R. Hhnle and M. Huisman, "Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools," in *Computing and Software Science: State of the Art and Perspectives*, B. Steffen and G. Woeginger Eds. Cham: Springer International Publishing, 2019, pp. 345-373.

[11] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM,* vol. 12, no. 10, pp. 576–580, October 1969. [Online]. Available: https://doi.org/10.1145/363235.363259.

[12] M. Naik and A. Aiken, "Conditional Must Not Aliasing for Static Race Detection," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, 2007: Association for Computing Machinery, in POPL '07, pp. 327–338. [Online]. Available: https://doi.org/10.1145/1190216.1190265. [Online]. Available: https://doi.org/10.1145/1190216.1190265

[13] R. H. B. Netzer and B. P. Miller, "What Are Race Conditions? Some Issues and Formalizations," *ACM Lett. Program. Lang. Syst.,* vol. 1, no. 1, pp. 74–88, March 1992. [Online]. Available: https://doi.org/10.1145/130616.130623.

[14] P. W. O'Hearn, "Resources, concurrency, and local reasoning," *Theoretical Computer Science,* vol. 375, no. 1, pp. 271-307, 2007. [Online]. Available: https://doi.org/10.1016/j.tcs.2006.12.035.

[15] J. Shirako, N. Vrvilo, E. G. Mercer, and V. Sarkar, "Design, Verification and Applications of a New Read-Write Lock Algorithm," in *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, New York, NY, USA, 2012: Association for Computing Machinery, in SPAA '12, pp. 48–57. [Online]. Available: https://doi.org/10.1145/2312005.2312015. [Online]. Available: https://doi.org/10.1145/2312005.2312015

[16] M. Singhal and N. G. Shivaratri, *Advanced Concepts in Operating Systems*. USA: McGraw-Hill, Inc., 1994, pp. 18-19.

[17] B. van Gastel, "Verifying reentrant readers-writers," *Master's thesis, Radboud Universiteit, Nijmegen, Netherlands,* 2010.

[18] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model Checking Programs," *Automated Software Engineering,* vol. 10, no. 2, pp. 203-232, April 2003. [Online]. Available: https://doi.org/10.1023/A:1022920129859.

[19] K. Winblad, K. Sagonas, and B. Jonsson, "Lock-Free Contention Adapting Search Trees," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, New York, NY, USA, 2018: Association for Computing Machinery, in SPAA '18, pp. 121–132. [Online]. Available: https://doi.org/10.1145/3210377.3210413. [Online]. Available: https://doi.org/10.1145/3210377.3210413

Appendix A Implementation details of abstracted parts

A.1 Database behaviour

Since verification of database functionality is not possible in VerCors, a workaround had to be found for this. Ultimately, the choice was made to transform this into a Java model of the database's functionality, which could be fully specified. For this reason, the exact implementation details regarding the database functionality were left out. The main functionality of the database is explained in Section 4.2, and the queries used to perform these operations are described in Section 6.1.8, in which a justification for the choice of database model is given. There, the behaviour of the database, and the corresponding database model behaviour, is listed.

A.2 Watchdog functionality

When a client which is holding a lock crashes, hangs for an extended amount of time or loses database connectivity, the client will not release their lock in a timely fashion. To avoid deadlocks in this scenario, locks which are held for an extended amount of time without getting regularly updated are automatically timed out and thus released. A client holding a lock must start a special 'watchdog' thread, whose sole task is to periodically update the lock and signal the client when the lock is lost. This prevents the lock from timing out, and prevents the working thread from continuing work on a critical section, despite the lock not being held. In this research, the assumption is made that eventually, all locks are released. Because of this, deadlock detection is not necessary, and the part of the provided implementation which handled deadlock detection was removed from the implementation.

The situation where this functionality might fail is as follows:

1. Thread A obtains the write lock.
2. Thread A starts a watchdog thread, whose sole task is to update the lock row.
3. Due to system load or thread scheduling issues, the watchdog thread fails to update the lock row in a timely manner. Because of this, the database considers the lock to be lost.
4. Thread B obtains the write lock (as the lock was timed out)
5. Thread A still hasn't been notified of the lock loss, and therefore continues writing to the protected resource despite no longer holding the lock. Simultaneously, Thread B is writing to the same resource, as it is holding the write lock.

In this situation (and many related situations), the read/write rules introduced in Section 2.4 are violated.

While this situation is admittedly rare, it may still occur. Because of the assumption made in Section 5.1, this situation is out of scope for this thesis.

Appendix B Overview of changes from Capita Selecta code until Intermediate 3

B.0 Java6Lock from Capita Selecta

First, the specification outline of the Capita Selecta is given.

First, the helper classes are mentioned, then the Lock, Thread and Main classes are discussed in detail.

B.0.1 The Subject class

The Subject class represents the resource which is protected by the lock. It has a predicate inv(), which represents the permissions given out, when the owner of the lock has access to the protected resource. The inv() predicate is unspecified in VerCors. This allows us to use it within the pre- and postconditions of the tryLock and release methods, without explicitly mentioning what the permissions regarding the protected resource are exactly.

B.0.2 The AtomicInteger class

In the AtomicInteger class, the following methods are present:

- set(int newVal) – sets the internal value of this AtomicInteger to the new value
- get() – Gets the current value of the AtomicInteger
- compareAndSet(int expected, int newValue) – If the current value of the AtomicInteger equals expected, sets the value to newValue.
- Constructors – There are two constructors; one without a parameter which sets the initial value to 0, and another which takes an integer as parameter and sets the initial value to its value.

B.0.3 The Lock class

The Lock class represents a single Lock. It has the following methods:

- tryLock() – Makes an attempt to obtain the lock
- release() – Releases the lock, when held
- currentThread() – Returns the identifier of the current thread (represents Thread.currentThread().getId() in Java)

This class allows us to lock and unlock locks, as well as obtain permissions to the protected resource (represented by the Subject class, described in Section B.0.1).

The Lock class has the following fields:

- AtomicInteger count – This field indicates the reentrancy level of the lock.
- AtomicInteger owner – This field holds the identifier for the thread holding the lock.

Aside from these two fields, there are a number of added ghost fields. These fields only exist within the specification, therefore they don't have any effect on the actual execution of the Java code.

- int T – This field represents the total number of threads that can concurrently access this class.
- int[] `held` – This field represents the current state of the lock: For each thread, it keeps track of the reentrancy level of the lock for that thread.
- int holder – This field keeps track of the current lock holder.

- Subject subject – This field represents the protected resource for this lock. When the lock is held, permissions regarding the subject are obtained (through a predicate inv()), and when the lock is released, these permissions are released again.

The Lock class has three inline predicates:

- initialised(int N): This predicate indicates that the value for T is precisely N.
- common(): This predicate specifies permissions on the fields of the Lock class and specifies the length of the `held` array, which is T.
- lockset_part(int current_thread): This predicate indicates permission on the `held` array indexed at current_thread. Furthermore, it indicates that if the lock is held by this thread (that is, if `held`[current_thread] > 0), the count and owner fields reflect this (by having the correct values), and exclusive write permission is held over the values of the count and owner fields, when the object invariant of the Lock class is obtained.

Furthermore, there's the object invariant, which indicates 'valid state', that is:

- The value of T is larger than 0
- `held` has the correct length (T)
- The right amount of permissions are held over the values of the owner and count variables (1\2 if the lock is held, 1 otherwise – the owning thread has the other 1\2 through the lockset_part predicate)
- The holder variable has a valid value ($0 \leq \text{holder} < \text{T}$)
- If holder is -1, the lock is not held
- For all $0 \leq i < T$: Read permission for `held`[i] is specified, if i is not the holder, `held`[i] == 0, `held`[i] is strictly non-negative, and if `held`[i] == 0, then owner.val does not point to i.

B.0.4 The Thread class

The Thread class represents one Thread, which has access to an array of Lock objects. This Thread can attempt to lock and unlock locks through the tryLock(int lock_id) and release(int lock_id) methods.

The Thread class has the following fields:

- int T – the number of threads in the thread pool
- int L – the number of Locks
- Lock[] locks – the array of Lock instances.
- ghost int current_thread – This field represents the Thread's unique identifier (obtained by Thread.currentThread().getId() in Java code). This is assumed to be bound to a specific range ($0 \leq current_thread < T$), and each Thread instance is assumed to have a unique value for current_thread. This allows for multiple Threads to share the same array of locks, as each Lock has support for T threads through the `held` array described above.

The Thread class has two predicates:

- common(): This predicate indicates that the Thread class is properly initialised, and permissions over the fields of the Thread class are specified. The bounds for current_thread (0 <= current_thread < T) and the length of the locks array (L) is checked, the locks array is initialised (locks[i] != null) with unique locks (locks[i] == locks[j] ==> i == j). Furthermore, for all i, locks[i].common() and locks[i].initialised(T) is specified, so the T values for the Thread and locks agree with each other, and the locks are each also properly initialised.

- lockset(bag<int>S): This predicate checks a few more injectivity constraints, namely:
    - The locks' count variables are unique
    - The locks' owner variables are unique
    - The locks' `held` ghost variables are unique

  Furthermore, locks[i].lockset_part(current_thread) is specified, so each lock is in a valid state with regards to this Thread, and finally, locks[i].`held`[current_thread] agrees with the given bag<int> S, that is, if lock i is supposed to be held reentrantly x times, i also shows up in S a total of x times. This is represented by locks[i].`held`[current_thread] == (i \in S).

B.0.5 The Main class

The Main class represents a program, which initialises an array of Locks and Threads, and then calls tryLock() and release a number of times from different Threads. The purpose of this class is to showcase that the specification is complete, that is, it does not leak permissions or fail to verify simple locking scenarios.

Values for T and L are chosen. In principle, these are arbitrary, but for the sake of performance, they are chosen to be fairly low.

The Locks are initialised, using this value for L to determine the array length, and the value for T is passed along to the constructor of Lock().

Then, the Threads are initialised. The current_thread ghost variable is the same as the thread index, such that threads[i].current_thread == i for all values of i. The locks array is passed along as a parameter to each thread class, such that the locks are shared between threads.

Finally, a number of trylock and release calls are made, from different threads, to different locks, and the validity of the resulting state is confirmed through a number of assert statements.

B.1 Intermediate 1

The first intermediate is a Reentrant Lock. The largest change compared to the Java6Lock version is that the code is now structured similarly to the BetterBe implementation. The main difference is that now, locks can be created and deleted as required. This is handled by the DbLockManager class. Furthermore, locks have a name, and locks with the same name are considered to be equivalent, and should be aliases of each other (to some extent, to be extended in later intermediates).

More details on the changes and the specification details can be found below.

B.1.1 AtomicInteger

Compared to the Java6Lock version, incrementAndGet() and decrementAndGet() are added.

B.1.2 DbNamedLock

The DbNamedLock class contains a number of fields which correspond to similarly named fields in the Java6Lock. Their meaning is also similar to the Java6Lock versions. Their meaning is described in Section B.0.3.

- final AtomicInteger instanceLockCount (count in Java6Lock)
- final AtomicInteger owningThread (owner in Java6Lock)
- Subject subject
- int holder
- int T
- int[] heldArray (`held` in Java6Lock)

Note that instanceLockCount and owningThread are now made final. They were considered read-only after initialisation either way, so this simply made the specification a little bit simpler, as all Permission predicates relating to these fields could safely be removed.

Aside from these fields, there is now also the field final int lockName – this represents the name of the lock. It is not used in the DbNamedLock class itself, but will be used in the DbLockManager. This is further discussed in Section B.1.4. There's also a corresponding pure method getName(), which returns the lockName.

The predicates common(), initialised(N) and lockset_part(current_thread) from the Java6Lock (Section B.0.3) are essentially unchanged.

Below, a list of changes to the methods are listed.

- Constructor: The constructor's contract has one slight addition: a lockName is passed as a parameter, which is stored in the lockName field. This is also added as a postcondition. Other than that, it is identical to the Java6Lock.
- tryLock(): This method closely follows the Java6Lock version, with one change: The AtomicInteger methods are no longer inlined. Other than this, the specification and implementation are very similar.
- Unlock(): This method was called release() in the Java6Lock. The specification and implementation is more or less identical to that of the Java6Lock.
- getThreadId(): This method was called currentThread() in the Java6Lock. Its specification is identical, and the implementation is left missing (except during testing).
- There are some (unsupported) other Lock methods, such as lockInterruptibly(), newCondition() and tryLock(long time, TimeUnit unit). They each throw an UnsupportedOperationException.

- getName() – A pure method which simply returns the lockName of the Lock object.
- lock() – A method which repeatedly calls tryLock(), until the lock is obtained. This is done in a while(true) loop, with a return statement when tryLock() returns true for the first time. Its specification is copied over from tryLock()'s \result ==> … postconditions.

B.1.3 Pair

The Pair class is a helper class which is essentially a structure to hold two things:

- final AtomicInteger, which keeps track of the amount of references of a lock which the LockManager has given out, and
- final DbNamedLock, which is the lock to keep track of.

It has two pure methods getFirst(), which returns the AtomicInteger, and getSecond(), which returns the DbNamedLock.

It has one inline predicate common(int T) , which specifies write permission to the value field of the AtomicInteger, DbNamedLock.common() and DbNamedLock.initialised(T).

B.1.4 DbLockManager

The DbLockManager class is responsible for keeping track of the DbNamedLocks in existence. It does this through two methods: createLock and destroyLock.

The fields of the DbLockManager class are as follows:

- ghost int T – This field indicates the maximum amount of threads that can operate on the locks on the same time. Thread identifiers are checked to be 0 <= thread_id < T.
- final int maxLockName – This field indicates the maximum allowed lock name. Similarly to T, in reality, this is unbound, but to make specification possible, it's capped at a certain value. Lock names are checked to be 0 <= lockName < maxLockName.
- final Pair[] pairs – This field holds an array of Pairs. The length of this array is maxLockName, so there is space for a Pair object for each lock name (see Section B.1.3).
- boolean isStopped – This field indicates whether the lockManager is stopped or not. A stopped lockManager can't create new (references to) locks, but existing locks can still be destroyed.

Below, the methods of this class are listed.

- Constructor – This initialises all fields. T and maxLockName are set to a certain value and the pairs array is initialised to be an all-null array of length maxLockName. isStopped is set to false initially.
- createLock(int lockName) – If the DbLockManager is stopped, an IllegalStateException is thrown. If pairs[lockName] == null, no Lock with this lockName exists within this lock manager. A new lock is created, as well as an AtomicInteger to keep track of its reentrancy. A Pair with the count and lock is created and stored in pairs[lockName]. After creation of this element (or if the element already existed), the count is incremented and the lock is returned.
- destroyLock(NamedLock lock) – This method calls destroyLock((DbNamedLock) lock). It has a weaker postcondition, as the NamedLock interface does not specify any fields, so the resulting state of these non-existing fields can't be specified.
- destroyLock(DbNamedLock lock) – This method checks that this lock exists within the lockManager (by checking that lock.getName() is in the pairs array). If not, it throws an

IllegalStateException. If it is a valid lock, it decrements the reference count for this lock. If that count was 0, an IllegalStateException is thrown. Finally, if this was the last reference for the lock, then pairs[lockName] is set to null, in effect deleting the Pair instance, and with it, the Lock instance.

  - Note: An important assumption is made here, that after calling destroyLock(), the lock passed as a parameter can no longer be accessed from the Thread calling destroyLock. Otherwise, it may be possible for multiple Lock objects with the same name to exist. This situation could cause unprotected concurrent access, as different Lock objects have no way to communicate with each other in intermediate 1. This behaviour is solved with the introduction of the database model (intermediate 3), but normal operation never accesses the lock after calling destroyLock() in any intermediate.

- stop() – This method sets the isStopped flag to true. If it was already set to true, an IllegalStateException is thrown, as a DbLockManager can only be stopped once.

B.2 Intermediate 2

The second intermediate adds read/write functionality.

B.2.1 *DbLockManager*

Below, the changes in the fields are given.

- pairs – This array has been expanded to be twice the size – maxLockname * 2. The first half of the array is reserved for write locks, the second half for read locks.
- final int maxLockIndex = maxLockName * 2; - This field did not exist in Intermediate 1. This field is added to accommodate the change in pairs.length described above.
- T – this field is made final. May be changed for intermediate 1 and possibly the Capita Selecta as well. In the specifications for intermediate 1 and Java6Lock the field was effectively final, but not marked as such, meaning permissions over the field had to be specified. A small change, but it simplifies the specification slightly.

The changes in predicates are listed below.

- pureSupport is renamed to common_part, and some of the injectivity checks have been moved into the lock invariant. Only the injectivity/non-null/state checks that are required for the specification of methods are specified here. The other state-specific stuff was moved into the lock invariant, since all methods are synchronized.
- The lock invariant is expanded in Intermediate 2, adding in a lot of the injectivity checks that were previously in common() and pureSupport (now called common_part), as well as adding some new ones:
    - An additional check for i == lockName + (lockType ? 0 : maxLockName) is added: This ensures that the first half of the array are write locks, and the second half are read locks
    - Uniqueness of pairs, locks, lock counts and owning threads are specified
    - Initialisation of locks are specified through lock.initialised(T) and lock.common()

The changes in the methods are given below.

- Added in createLock(int lockName, boolean lockType) and createLock(DbNamedReadWriteLock readWriteLock, int lockName, boolean lockType) – specification of these is similar to createLock(int lockName), with additional claims over the lockType and readWriteLock state. Note that when calling createLock(readWriteLock, lockName, lockType), no guarantee is given that \result.readWriteLock == readWriteLock. This is intentional, to ensure that locks between readWriteLocks with the same name alias with each other. It is, however, assumed that both the read and write locks operate on the same readWriteLock object. This is not (yet) specified. It is however required in order to ensure that read/write locking works as intended.
- destroyLock: Added requirement that pairs[i].getFirst().val > 0. This is part of the lock invariant. In the code, an IllegalStateException would be thrown if this were to happen (it shouldn't).

There's one more overall change. Instead of indexing the pairs array with lock.getName(), we index it with lock.getName + (lock.getType() ? 0 : maxLockName). This is to ensure that read and write locks are both supported in the specification.

B.2.2 DbNamedReadWriteLock

This class was added in Intermediate 2. It holds a reference to a readLock and a writeLock. Both are instantiated lazily, that is, they are only instantiated upon calling readLock resp. writeLock.

The fields of the DbNamedReadWriteLock class are:

- Final int lockName: The name of the lock
- Final DbLockManager manager: The lock manager this lock belongs to
- Final AtomicInteger readWriteFlag: A flag representing the current state of the ReadWriteLock: -1 is unlocked, 0 is read locked, 1 is write locked
- DbNamedLock readLock – the read Lock
- DbNamedLock writeLock – the write Lock

The predicates of the DbNamedReadWriteLock class are listed below.

- lock_invariant: This simply holds Perm(readWriteLock.val, 1\2), to ensure that read permission is always available, but write permission is only available as needed through the next predicate:
- common(): Holds 1\2 permission for readWriteLock.val, so the value can be read, but only written to when holding the object invariant. Establishes the following:
    - lockManager != null
    - 0 <= lockName < maxLockName
    - lockManager.common_part(lockName) and lockManager.common_part(lockName + maxLockName) hold, for write resp. read locks to ensure that when a readLock/writeLock exists in the lockManager, they're properly instantiated
    - If the readLock is not null, it's the same lock as in the lockManager, and its name and type are correct.
    - Idem for writeLock

The methods are listed below.

- Constructor: This instantiates the final fields. lockName and lockManager are given parameters, and readWriteFlag is initialised to -1. The specification establishes this, and that readLock and writeLock both start as null. Common() is established.
- readLock(): Requires common(), ensures that readLock != null. If it was already non-null, the readLock is unchanged.
- writeLock(): Similar to above, but for write Locks.
- close(): Resets the lock back to its initial state.
- getReadWriteFlag(): Returns the current value of readWriteFlag.
- casReadWriteFlag(): Does compareAndSet for readWriteFlag.

B.2.3 DbNamedLock

Below, the changes in the fields are given.

- T is made final
- heldArray is made final
- ghost bag<int> holders is added, to keep track of which threads have a read lock
- subject is made final
- final boolean lockType added to keep track of which type of lock this is (true = write, false = read)

- final DbNamedReadWriteLock readWriteLock added – this is the reference to the readWriteLock, used for synchronising the Read and Write locks

B.3 Intermediate 3

B.3.1 DBRow

This class represents a single row in the database. It contains a number of final fields which represent the database fields.

- final int lockName: The name of the lock
- final int uuid: The UUID of the lock
- final boolean lockType: The type of the lock (true for write, false for read)

The DBRow class contains just the constructor, which initialises the object with a value for each of the final fields, which are given as parameters.

B.3.2 ListDBRow

This class represents a list of DBRows. Its implementation is a simplified version of the java.lang.ArrayList<E> implementation. The class does not use generics, as generics are not currently supported by VerCors, so it's implemented to be a list of DBRow objects.

It has three fields and one ghost field:

- initialCapacity: The initial capacity of the underlying array
- size: The current amount of elements in the list
- arr: The array containing the elements of the list
- ghost uuidSeq: A seq<int> containing all the uuids of elements in the list, in the same order as in the list.

It also has two predicates:

- common(): This predicate indicates write access to size, uuidBag, arr and indicates that the fields of the list is in a valid state. This means the following things are checked:
    - arr is not null and has a strictly positive length
    - The cardinality of uuidSeq is equal to the value of size
    - size is in a valid range (0 <= size <= arr.length)
- common_arr(): This predicate indicates write access to all elements of arr and indicates that the arr array is in a valid state. The following things are checked:
    - All elements with an index in the range [0 .. size):
        - Are not null
        - Are unique and have unique uuids:
          (\forall int i = 0 .. size, int j = 0 .. size; i != j; arr[i] != arr[j] && arr[i].uuid != arr[j].uuid)
        - Have unique uuids
        - The uuids of the elements match with the contents of uuidSeq:
          (\forall int i = 0 .. size; arr[i].uuid == uuidSeq[i]
    - All elements from size to arr.length are null
- total(): This predicate combines common() and common_arr() to indicate that the list is properly initialised. The predicates are split up in order to check for injectivity constraints in the DbModel class (see Section B.3.3).

The methods of the ListDbRow are listed below.

- Constructor: This initialises the list as empty, with an initial capacity of 10, and a size of 0. It establishes total().

- arrayCopy(DBRow[] src,int srcPos,DBRow[] dest,int destPos,int length): This method has the same behaviour as System.arraycopy(). Copies elements src[srcPos..srcPos+length] to dest[destPos..destPos+length]. If src and dest are the same array, it behaves as if all elements of src are first copied to a temporary array, and then written to dest. This is specified in the postcondition using
  ensures (\forall int i = destPos .. destPos + length; {:dest[i]:} == \old(src[srcPos + (i - destPos)]));
  Written differently, this says dest[destPos + i] == src[srcPos + i] for i∈[0 ..length].
  This method has a complex specification, as a result of the Permissions being difficult to manage when src and dest point to the same array. Only read permissions are required for src[srcPos..srcPos+length], but write permissions are needed for dest[destPos..destPos+length]. In particular, if there is overlap between these ranges, careful attention must be paid to which parts of the dest array write permissions are specified over as opposed to specifying 1\2 permissions (as the other half is already obtained by the Permission predicate for the src array in the part of the ranges which overlap.
  In all scenarios described below, 1\2 permissions were specified for the range src[srcPos..srcPos+length]. This means that if there is any overlap between the ranges of the src and dest array, the other 1\2 (but no more!) must be specified over the corresponding range in the dest array to obtain the write (equivalent to 1) permission. If more than 1 permission is specified, this leads to an unsatisfiable precondition, as it is not possible to specify more than a full write permission.
  Different Permission predicates were required for the following scenarios:
  - src == dest && srcPos <= destPos && srcPos+length > destPos
    In this scenario, the ranges src[destPos..srcPos+length] and dest[destPos..srcPos+length] point to the same DBRows, so for the dest range, only 1\2 permissions are specified. For the range dest[srcPos+length..destPos+length], write permissions are specified.
  - src == dest && srcPos <= destPos && srcPos+length <= destPos
    In this scenario, there is no overlap in the ranges src[srcPos..srcPos+length] and dest[destPos..destPos+length]. Write permission is specified for the range dest[destPos..destPos+length].
  - src == dest && srcPos > destPos && destPos+length > srcPos
    In this scenario, the ranges src[srcPos..destPos+length] and dest[srcPos..destPos+length] point to the same DBRows, so for the dest range, only 1\2 permissions are specified. For the range dest[destPos ..srcPos], write permissions are specified.
    src == dest && srcPos > destPos && destPos+length > srcPos
    In this scenario, there is no overlap in the ranges src[srcPos..srcPos+length] and dest[destPos..destPos+length]. Write permission is specified for the range dest[destPos..destPos+length].
  - src != dest
    In this scenario, no overlap is possible, as it is (implicitly!) assumed that the memory locations which src and dest point to have no overlap.
- add(DBRow elem): Adds a row to the list. If necessary (when size >= arr.length), it expands the list to double its current capacity. To preserve uniqueness of the array locations, this DbRow must not already be in the list. Furthermore, to preserve uniqueness of the uuids present in the list, the DbRow must have a uuid which is not present in the uuidBag (and therefore also not in the list). It inserts the uuid of the newly added row into the uuidBag.

- get(int i): Retrieves element arr[i]. Requires that 0 <= *i* < size. This method is pure.
- size(): Retrieves the current value of size. This method is pure.
- fastRemove(DBRow[] es,int i): Removes element es[i] from the list, and shifts all elements after index i one position to the left. This method requires write permissions on es[j] for all j∈{i ..size}, if i < \old(size) - 1, as the elements to the right of i must be shifted one position to the left. Otherwise, if *I* == \old(size) - 1, write permission is only needed for es[\old(size) - 1], as no elements need to be shifted in this case.
- remove(int uuid): Removes and returns the first element arr[i] for which arr[i].uuid == uuid. Returns null if no such element existed. Updates uuidBag accordingly.

B.3.3 DbModel

The DbModel class has the following fields:

- final int maxLockName: The maximum allowed lockName. Initialised through a constructor parameter. This indicates the length of each of the following arrays.
- final ListDbRow[] lockData: This field contains one List<DbRow> for each lock name. This allows for easy indexing of rows by lock name.
- ghost final int[] heldArray: This array keeps track of the amount of different (distributed) clients that are holding a lock, again indexed by lock name.
- ghost final int[] readWrite: For each lock, keeps track of whether the lock is held exclusively/write mode (1), shared/read mode (0) or is unlocked (-1).

The predicates of the DbModel class are:

- common(): Indicates that all three arrays (lockData, heldArray and readWrite) are non-null and have length maxLockName. Also indicates that the DbModel instance is committed (the lock invariant has been established).
- perm_part(int lockId): Indicates that lockId is a valid lock identifier (0 <= lockId < maxLockName), indicates permissions to access each of the lockData, heldArray and readWrite arrays at index lockId, and indicates that lockData[lockId] != null.
- perm_all(int lockId): This predicate combines perm_part(lockId) with [1\2]lockData[lockId].common() and [1\2]lockData[lockId].common_arr(). Note that both common() and common_arr() are scaled by [1\2] here. The other half is stored in the lock invariant, which can be found below. This is done to ensure that the fields of the underlying list can be accessed in read mode in the specification, but write permission is available inside the methods (since the methods are synchronized).
- The lock invariant: The invariant indicates that the DbModel is in a valid state. The following are specified:
    - common() holds
    - [1\2] permissions are specified for each element of the heldArray and readWrite ghost variables – this ensures that upon entering a method which has a requires clause for perm_part(lockId), a full write permission is available for heldArray and readWrite at this index, and a read permission is available for all other elements.
    - [1\4] permissions are specified for all elements of the lockData array. This ensures that write permission is never available after exiting the constructor, as at most 3\4 permissions are held, even when holding perm_part(lockId). The list should never be replaced in its entirety, only its elements can ever be written to.

- All elements of lockData are non-null and unique (i != j ==> lockData[i] != lockData[j]), and have unique underlying arrays (i != j ==> lockData[i].arr != lockData[j].arr).
- We hold [1\2]lockData[i].common() and [1\2]lockData[i].common_arr() for each index i. Note that the other [1\2] is held by perm_all(lockId), to ensure complete write permission for lockData[lockId]'s fields and arr elements. There is read permission for other indices of lockData, as the perm_all is always held for at most one lock name.
- All elements of lockData[*].arr[*] are unique, meaning that for each unique combination of integers i,j, we have unique heap locations for lockData[i].arr[j], or in more mathematical terms: (\forall int i = 0 .. maxLockName, int j = 0 .. maxLockName, int k = 0 .. lockData[i].size, int l = 0 .. lockData[j].size; i != j ==> lockData[i].arr[k] != lockData[j].arr[l])
- All elements of heldArray are strictly non-negative. If heldArray[i] == 0, the lock is unlocked. Otherwise, the lock is locked by heldArray[i] different threads.
- All elements of readWrite∈[-1,0,1]. Here, -1 represents unlocked, 0 represents read locked, and 1 represents write locked.
- heldArray[i] == 0 if and only if readWrite == -1
- readWrite[i] == 1 ==> heldArray[i] == 1. When a write lock is held, it can only be held by one thread.
- heldArray[i] == lockData[i].size. There are exactly as many threads holding the lock as there are database rows for that lock.
- readWrite[i] == 0 ==> !lockData[i].arr[j].lockType – If a lock is marked as read-locked, there can only be read lock rows in the database model.
- readWrite[i] == 1 ==> lockData[i].arr[j].lockType – If a lock is marked as write-locked, there can only be write lock rows in the database model (in fact, there can only be one).

The methods of the DbModel class are:

- Constructor: Establishes the lock invariant and gives out perm_all(i) permissions for each index i∈{0..maxLockName}. In the postcondition, this is split out into perm_part(i), [1\2]lockData[i].common() and [1\2]lockData[i].common_arr() in order to insert injectivity checks between each of these predicates. Initialises the lockData, heldArray and readWrite fields as arrays of length maxLockName. Initialises all elements of lockData as empty lists, and marks all locks as not held by setting all elements of heldArray to 0 and all elements of readWrite to -1.
- checkReadWrite(int lockName, boolean lockType): Indicates whether or not it is legal to insert the requested lock into the database, taking into account the read and write exclusivity rules. A write lock cannot be inserted if this lock is already held (in read or write mode). A read lock cannot be inserted if this lock is already held in write mode. Otherwise, the lock row can be inserted.
- insert(int lockName, int uuid, boolean lockType, int thread_id): If the read and write exclusivity rules are respected, as checked by checkReadWrite(), inserts a row into the corresponding lockData[lockName] list. This represents inserting a row into the database. Returns 1 if the insertion happened successfully, or 0 otherwise.
- delete(int lockName, int uuid): Deletes the row with the corresponding lockName and uuid, if present. If the row was successfully deleted, returns 1. Returns 0 otherwise.

B.3.4 DbNamedLock

The DbNamedLock has the following fields:

- int heldCount – This field was previously called heldArray. In intermediate 3, lock instances are no longer shared between threads. Therefore, only a single thread may ever be holder of this lock instance, and an integer suffices. The field is no longer final.
- The holders field is removed. Since only one thread can hold the lock, it does not make sense to define ghost state to keep track of lock holders.
- int instanceLockCount – This field is no longer an AtomicInteger, as only one thread may access the lock instance.
- final int owningThread – Again, since exactly one thread is owner of this Lock instance, owningThread indicates which thread may access the lock's tryLock() and release() methods. For this reason, this field is now an int (instead of an AtomicInteger), as its owner will never change.
- final DbModel dbModel – This field refers to the database model.
- The readWriteLock field is removed, since having a reference in DbNamedLock is no longer needed to synchronize read and write locks. Instead, the DbModel is responsible for this.
- final int uuid – This field is added. Each lock instance has a unique uuid. This uuid is used to uniquely identify a row in the database.

The predicates of the DbNamedLock class are:

- Lock invariant – The lock invariant was altered to reflect the changes in the fields, as described above. This has the effect of removing *all* quantifiers from the lock invariant, which helps speed up verification.
    - heldCount – we now specify 1\2 permission over this field and indicate that heldCount >= 0.
    - dbModel – We specify that dbModel != null and that the lockName of this lock is within bounds of the dbModel's maxLockName (0 <= lockName < maxLockName)
    - instanceLockCount – We specify 1\2 permission over this field, rather than instanceLockCount.val, as was done previously.
    - heldCount == instanceLockCount: instanceLockCount now only keeps track of how many times *this* thread holds the lock, so the two fields can simply be linked like this.
    - Since read/write exclusivity is now checked by the database model, it is no longer necessary to link to a readWriteLock instance, and all quantifiers containing such a link was removed.
    - The quantifiers containing heldArray were removed. In particular, the link between the contents of heldArray and owningThread or holder were no longer needed, as during the lifetime of this Lock instance, only one thread is the owner (and possibly holder) of the lock. Instead of checking this in the lock invariant, it is checked in the methods whether the caller is the thread which owns the lock.
- common() – The common predicate now indicates the dbModel is non-null and is committed and indicates that the lockName is within bounds of the dbModel's maxLockName. Aside from this, the mentions to the removed and changed fields were removed, as they were not needed for specification.
- lockset_part() – Any mention of heldArray[thread_id] was replaced by heldCount, and the PointsTo predicate equating owningThread and thread_id was removed by a simple equation: owningThread == thread_id.

The methods of the DbNamedLock class are:

- Constructor – Two ghost parameters are added (uuid and current_thread), to initialise respectively the uuid and owningThread fields. The readWriteLock parameter was replaced with the dbModel parameter. The requirement is added that the given lockName is within dbModel's bounds (0 <= lockName < dbModel.maxLockName). The heldCount variable is initialised to a value of 0, as heldArray previously was. The postcondition now additionally specifies that the invariant is established, and the values of dbModel and uuid are properly stored in the created object.
- public int getOwningThread() – This returns the value of owningThread for use in the DbLockManager class. Simply returns the value of owningThread.
- private int getUUID() –The implementation actually generates a random UUID, but this behaviour is emulated by a given ghost parameter uuid which the method returns, similarly to the getThreadId() method defined in Section B.1.2. This method is used to initialise the uuid field in the constructor.
- tryLock()
    - Specification changes: The dbModel.common() and dbModel.perm_all(lockName) predicates are specified as pre- and postconditions of this method. This is to ensure we have the right permissions for the insert call done later.
    - Any reference to heldArray[current_thread] was replaced with heldCount.
    - If the heldCount was 0 and the lock was obtained (the returned value is true), the subject's invariant is obtained. It is implicitly assumed that for a read lock, this invariant can be split arbitrarily many times, such that it can be shared between read locks with the same name.
    - Implementation changes: A check is added to ensure that the caller is the owner of this lockInstance.
    - The distinction between the cases where readWriteLock is or is not null is removed, as the readWriteLock field is removed.
    - If the lock was already held, we simply increment the instanceLockCount and heldCount fields and return true. This is the case of a reentrant tryLock call.
    - If the lock was not held, the insert method is called on the database model. If this call returns 0, the lock attempt failed and false is returned. If the call returns a value larger than 1 (which is impossible with the current specification and implementation), false is returned.
    - If the call returns 1, this means one database row was added and the lock has successfully been obtained. In this case, instanceLockCount and heldCount are incremented.
- lock()
    - Added context_everywhere clause which indicates dbModel.common() and dbModel.perm_all(lockName). This ensures that the right permissions are present when calling tryLock.
    - Added requirement that current_thread == owningThread to ensure tryLock does not throw an exception.
    - Any reference to heldArray[current_thread] in the postcondition was changed to heldCount. The postcondition now specifies that heldCount will be incremented by one, and when the lock was obtained for the first time, subject.inv() is obtained.
    - The code structure was slightly altered to more accurately reflect the implementation. After each failed tryLock attempt, the thread sleeps for a certain

amount of time. This allows other threads the chance to finish their work and ensures the database is polled less frequently. In terms of specified behaviour, this is effectively identical, as we can more or less ignore the calls to Thread.sleep, aside from specifying that upon encountering a RunTimeException, we can no longer guarantee anything about program state (through a signals (RunTimeException e) true; clause). This can be done because Thread.sleep does not alter the state of the heap, and the permissions obtained by this method are not released upon calling it.
    - A boolean success is added, which is initialised to false and changed to true upon a successful tryLock call. The while loop (which previously was while (true) …) is changed to while (!success). This allows us to define a loop invariant:
        - (!success && heldCount == \old(heldCount)) || (success && heldCount == \old(heldCount) + 1)
        - success && \old(heldCount) == 0 ==> subject.inv()

      This loop invariant indicates that eventually, heldCount will be incremented by exactly one, as the postcondition states.
- releaseLock(boolean missingOk) – This method removes the database row corresponding to the lock instance, thereby releasing the lock so it can be obtained by other threads. If missingOk is set to false and dbModel.delete() returns 0, the lock was released while it was not held. Then, a LockLostException is thrown. In practice, this method is only called with missingOk being set to true, so this never occurs.
- unlock()
    - Specification changes: The dbModel.common() and dbModel.perm_all(lockName) predicates were specified in a context clause. This ensures that the correct permissions are held when finally releasing the lock for the final time.
    - Any reference to heldArray[current_thread] was changed to heldCount.
    - The clauses regarding readWriteLock were removed
    - The signals clause for IllegalStateException was changed. Now, it signals current_thread != owningThread.
    - Implementation changes: A check was added to check that current_thread == owningThread. If this is not the case, an IllegalStateException is thrown, as only the owningThread is allowed to call the release method on this lock instance. Other threads may obtain their own lock instances through the DbLockManager, as described in Section B.3.5.
    - As each lock instance is now single-threaded, we can simply decrement the heldCount and instanceLockCount variables. If the value of instanceLockCount reaches 0, we additionally call releaseLock(), which removes the database row corresponding to this lock.

B.3.5 LockInstanceArray

This is an inner class of the DbLockManager class. The purpose of this class is to hold an array of Pair objects. The reason this class exists is because support for nested arrays (Pair[][]) is lacking in VerCors at the time of writing. Having an inner class which contains an array as its only field allows for circumventing this limitation.

The LockInstanceArray class has one field: final Pair[] pairs. It holds an array of pairs.

It also has a constructor(int maxLockName) – Initialises the pairs array as an array of length maxLockName * 2. All elements of this array are null.

B.3.6 DbLockManager

The fields of the DbLockManager class are:

- ghost int next_uuid – Keeps track of the next uuid to give out to a lock instance to ensure uniqueness.
- private final DbModel dbModel – The database model which this DbLockManager is connected to.
- private final int maxLockName – This field is now private.
- private final LockInstanceArray[] lockInstanceArray – In intermediate 2, lock instances were shared between all threads. This meant that only one Lock instance per lock name was needed to support all T threads. The field Pair[] pairs contained these lock instances, as well as an AtomicInteger which kept track of how many threads within this lock manager had access to a reference to this lock instance (see Section B.1.3). However, in intermediate 3, each thread has its own set of lock instances. This means that each thread requires such an array of Pairs. Since support for two-dimensional arrays is limited in VerCors, an inner class LockInstanceArray (see Section B.3.5) was added. One instance of this class is needed for each thread.
- final int num_threads – Added variable to hold the value of T, so the lock instance array can be initialised

The predicates of the DbLockManager class are:

- Lock invariant
    - Added Perm(next_uuid, 1\2) – the other half is stored in general(). This ensures that the next_uuid can be incremented inside the createLock method, and the value of next_uuid can be accessed within the specification in read mode.
    - Specified the exact values of T (num_threads), maxLockName (10), maxLockIndex (2 * maxLockName)
    - Added reference to dbModel, with a non-null constraint and specified that dbModel.maxLockName == maxLockName.
    - lockInstanceArray – A lot of things have changed compared to the previous intermediate, so here's an overview of what is specified regarding this array (previously, some of these clauses were specified for the pairs array from intermediate 2, see Section B.2.1).
        - Specified the length (T),
        - Indicated that 1\2 permissions for each element are held,
        - All elements are non-null and are injective (including the pairs array contained within),
        - The pairs array for each element has the correct length (maxLockIndex),
        - 1\2 permissions to access elements of lockInstanceArray[i].pairs[j] are held,
        - The elements of each pairs array are injective, both within the pairs array and between pairs arrays of different elements: (\forall int i = 0 .. T, int j = 0 .. maxLockIndex, int k = 0 .. T, int l = 0 .. maxLockIndex; i != k || j != l ==> lockInstanceArray[i].pairs[j] != lockInstanceArray[k].pairs[l])
        - These unique elements of the pairs arrays all have unique and non-null first and second fields, and the contained lock instance also has a unique uuid
        - Each pair's lock instance has a uuid less than next_uuid

  - Each pair's lock instance has a lock type (read/write) corresponding to its index: The first half of the pairs array has write locks, the second half has read locks.
  - Perm(lockInstanceArray[i].pairs[j].first.val, 1\2) is specified for each i, j. This value is positive.
  - For the Lock instance contained within lockInstanceArray[i].pairs[j], the common() and initialised(T) predicates are specified.
- common_part(int current_thread, int i) – The current_thread parameter is added. This is needed to access the correct lock instance. Permission to access this lock instance is specified. All the other specified statements regarding the pair still hold, with a few additions: it is specified that the type is write for indices in the range [0 .. maxLockName) and read for indices [maxLockName .. maxLockIndex). Furthermore, it is specified that the pair's fields have both committed the lock invariant (and are thus properly initialised).
- general() – Specifies that the DbLockManager object is committed, that is, its object invariant is initialised. This predicate indicates 1\2 permission to access next_uuid. Furthermore, it specifies a few of the clauses also present in the lock invariant; that is, the value of T, maxLockIndex, maxLockName, dbModel != null, dbModel.maxLockName and the length of the lockInstanceArray (T). The reason that this was put in a separate predicate is that this 1\2 permission to access next_uuid can only be given out once, and in DbNamedReadWriteLock (see Section B.2.2), common_part is specified to be held twice (with differing indices). For this reason, general() was moved out of common_part and some parts were added.
- common() – This predicate contains general() and specifies the length of the lockInstanceArray and each pairs array within. Furthermore, permissions to access each element of these arrays are specified, all with 1\2 permission (the other half is stored in the lock invariant). Injectivity is specified for each of the elements of the lockInstanceArray, and injectivity is specified for each pairs array contained within, similarly to how it was done in the lock invariant. Finally, write permission is specified over the isStopped variable.

The methods of the DbLockManager class are:

- Constructor() – Changed specification, now establishes common(). The values for num_threads (and T), maxLockName, maxLockIndex were moved into common(), as well as initialisation and permission to access the lockInstanceArray. Initialises a new DbModel, values for all (ghost) fields and ensures proper initialisation for the lockInstanceArray.
- Constructor(DbModel dbModel) – Added constructor to initialise a DbLockManager with an existing DbModel object. Other than this, the constructor is identical to the one described above.
- lockKey(int lockName, boolean lockType, int maxLockName) – Added a ghost method which returns lockName + (lockType ? 0 : maxLockName). This returns the index of the given element in the Pair array.
- createLock(int lockName) – Added general() as a precondition, as it was split off from common_part(). Added new value of next_uuid to the postcondition. Added ghost parameter current_thread. Changed pairs[lockName] to lockInstanceArray[current_thread].pairs[lockName] to match the change in the fields.
- createLock(~~DbNamedReadWriteLock lock,~~ int lockName, boolean lockType) – Made the specification match that of createLock(int lockName), with one change: Anywhere

pairs[lockName] is indexed, instead, lockKey(lockName, lockType, maxLockName) is used. This ensures that the correct index is taken, regardless of the value of lockType.
In terms of implementation, there are little changes – the database model is passed along to the constructor of DbNamedLock, instead of the DbNamedReadWriteLock parameter. The current_thread and next_uuid parameter are also passed along. The next_uuid parameter is incremented after initialisation of the new Lock object.

- destroyLock – In intermediate 2, a ghost parameter i was added, which had the correct value for the index of the lock. In intermediate 3, this parameter was removed and lockKey(lockName, lockType, maxLockName) was used instead. A ghost parameter current_thread was added, to index the lockInstanceArray. Other than this, there were only syntactical changes to the specification. The implementation was also altered. Since Lock instances are no longer shared between threads, only the Lock's owner can call destroyLock. It is checked that current_thread == lock.owningThread. If this is not the case, an IllegalStateException is thrown. Other than this, there were only syntactical changes.
- getDbModel – A pure method was added to retrieve the dbModel.

B.3.7 DbNamedReadWriteLock

The fields of the DbNamedReadWriteLock class are:

- readWriteFlag – This field was removed, as synchronization between read and write locks is now done through the database model.

The predicates of the DbNamedReadWriteLock class are:

- Lock invariant – Since readWriteFlag was removed, the lock invariant (which only held permission to access readWriteFlag.val) was also removed.
- common(int current_thread) – current_thread parameter was added. Permission for readWriteFlag.val was removed. Furthermore, the parts of the state which indicated readLock == null ==> … and writeLock == null ==> … were removed, as they were incorrect. When a readLock is destroyed, it is not necessarily destroyed in the lockManager as well. This held only if the DbNamedReadWriteLock was holding the only reference to the lock.

The methods of the DbNamedReadWriteLock class are:

- Constructor – Removed reference to readWriteFlag. Added current_thread parameter for use in the common() predicate.
- readLock() – Added ghost parameter current_thread. Rather than use the old ghost parameter i in the call to lockManager.createLock(), now use the new parameter current_thread.
- writeLock() – Identical changes as to readLock() method.
- close() – Added common(current_thread) to the postcondition. Altered signals clause to indicate permissions and initialisation for all fields except for readLock and writeLock.