Reliability of a read-write lock implementation

Bachelor thesis computer science

Bernard van Gastel
Supervised by Sjaak Smetsers en Marko van Eekelen
Radboud University, Nijmegen
B.vanGastel@student.science.ru.nl

February 7, 2008

Abstract

It is wiser to find out than to suppose.

– Samuel L. Clemens, known as Mark Twain [5]

Read-write locking is an important mechanism to speed up concurrent access to certain resources within the same program. The correctness of the implementation is important: for example no deadlock should occur and certain properties, like exclusivity for writers, must be guaranteed. To assert the reliability of a read-write lock implementation, it is modelled and checked in UPPAAL. After adjustments to the implementation no bugs were found.

Contents

1 Introduction **3**

2 Read-Write Lock **5**
2.1 Introduction . . . 5
2.2 Properties . . . 7
2.3 Implementation . . . 9

3 Modelling in UPPAAL **14**
3.1 UPPAAL . . . 14
3.2 Modelling Technique . . . 14
3.3 Modelling the Read-Write Lock . . . 15
3.4 Checking the Read-Write Lock . . . 19

4 Other implementations **21**

5 Future Work **23**

6 Conclusion **24**

Bibliography **25**

A Final Version - C Code **26**

B PVS model **30**

Chapter 1

Introduction

> *Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.*
>
> – Edsger Dijkstra [2]

Reliability of software is important. There are several methods to assert the reliability of software modules. A strong method means a greater degree of certainty of the reliability of the module. There are mainly three methods: testing, model checking and formal proof. To start, extensive testing of a module can give an idea of its reliability. Notwithstanding such testing, the above Dijkstra quote remains highly illustrative. Stated differently: if you do not find any bugs with testing, it is possible the tests were not extensive enough (thus there is still a possibility the program contains bugs). Testing is not a strong method, yet, because of the inherent costs of stronger methods, it is the method most widely used.

The second and a stronger method is model checking (for example, with the tool UPPAAL). To use this method, the software module must first be modeled, and thereafter certain properties can be checked in each possible state of the model. Such a model is an abstraction of details and includes only the essential parts of the module and is therefore less complex. Hence the model can only have less states then the original implementation. A level of abstraction should be reached so the model can have only a limited number of states hence all states can be checked in finite time. Otherwise extra limitations on the number of states must be imposed (but well chosen not to affect the essential part) because otherwise the model is uncheckable.

The third method, formal proofing, is used if there are many variables in the model (and hence the model bigger) and an explosion of states occurs (there a many possible states). With each added variable the number of states grows exponentially, possibly leading to a model not checkable in a limited period of time or leading to an infinite model which is per definition not checkable. By using formal proof techniques, the third technique, and optionally a proof assistant (e.g., PVS), it is possible to prove certain properties not deducible with model checking. Formal proof of correctness is the strongest method available to assert the reliability of software, however it is also the most expensive.

It is essential for both model checking and formal proof methods to make an abstract model. If there is no model or a too detailed model, it is hard to proof anything aside from trivial properties, since too much time is lost checking or proofing details not relevant for the properties of the software to be proved. Also the model must be sound: every property of the model must be a property of the implementation.

The code in this paper originates from the self-written open-source library PIToolkit. The library is written in Objective-C, an almost minimal superset of C to support object oriented programming. The Objective-C language does not have any own locking mechanism (although different Objective-C libraries all have their own, incompatible of each other, mechanisms). All multithreaded code is dependant on the POSIX multithread API's and all code is being used in 24/7 applications. Until writing this article, no errors in the read-write locking code where found during testing or running applications using the code. For readability, all Objective-C code is converted to plain C and appended to this thesis in appendix A.

First read-write locks, its properties and its implementation are described. In the next section the implementation is modelled and certain properties (mentioned in section 2) are checked with the model checker UPPAAL. Other implementations are looked into in the section thereafter.

Chapter 2

Read-Write Lock

One has attained to mastery when one neither goes wrong nor hesitates in the performance.

– Nietzsche [6]

2.1 Introduction

Read-write locks are an important mechanism for correct and fast execution of concurrency. Inside a computer multiple applications (called processes) work simultaneously. These processes are isolated from each other with respect to resources (e.g., memory or file). In the event a fault occurs within a process, it is isolated within this process (the notable exception being interdependent processes). Furthermore, in terms of executing speed or developer hours it can be efficient to execute multiple threads within a single process, the trade-off however is no isolation between threads. When two threads are working on the same resource within the same process, errors can occur [7], resulting in a crash of the program or worse in corrupted files (this also applies to interdependent processes).

Normally, a lock (also called a mutex) is used to ensure that only one thread will access a certain set of resources at a given time. A lock has two operations: `lock` and `unlock`. The `lock` operation is done before the critical section and the `unlock` operation after the critical section. The most basic lock can only be locked one time by a given thread (non-reentrant) and can be implemented with a boolean and an atomic test-set operation (only if a variable has a certain value replace it with a different value, all atomically, see section 5.2 of [8] for more information). In figure 2.1 a standard lock implementation is listed. Extensions are for example reentrancy and condition locking. Reentrancy allows nested lock operations by the same thread. Condition locking is a special kind of lock where a thread waits (stalls) until a certain condition is satisfied and automatically continues when notified (a 'notify' operation needs to be executed when the condition changes). An example usage is listed in 2.2. The examined read write lock uses a condition lock. Condition locks are extensively described in section 5.4 of [8].

A read-write lock functions differently from a normal lock: it either allows multiple threads to access the resource in a read-only way, or it allows one, and only one, thread at any given time to have full access (both read and write) to the resource [7, 8]. It is fundamentally different from both a normal mutex (since it allows multiple threads to obtain a read lock)

```
// lock = 0 -> unlocked
// lock = 1 -> locked
// testset(...) test for 0 and if 0 sets the variable to 1.
// testset(...) returns true if the test for 0 succeeded, otherwise false
void lock(int *lock) {
   while (!testset(lock));
}
void unlock(int *lock) {
   *lock = 0;
}
```

Figure 2.1: Standard lock solution adapted from section 5.2 of [8].

```
ConditionLock *l;
Item items[16];
int count = 0;

void push(Item item) {
   lock(l);
   while (count == 16)
     wait(l);
   items[count++] = item;
   if (count == 1) {
      unlockWithSignalAll(l);
   } else
      unlock(l);
}

Item pop() {
   lock(l);
   while (count == 0)
     wait(l);
   Item retval = items[--count];
   if (count == 15) {
      unlockWithSignalAll(l);
   } else {
      unlock(l);
   }
   return retval;
}
```

Figure 2.2: Condition lock example adapted from section 5.4 of [8].

```
Lock *read;
Lock *write;
int readCount = 0;
void readLock() {
  lock(read);
  readCount++;
  if (readCount == 1)
    lock(write);
  unlock(read);
}
void readUnlock() {
  lock(read);
  readCount--;
  if (readCount == 0)
    unlock(write);
  unlock(read);
}
void writeLock() {
  lock(write);
}
void writeUnlock() {
  unlock(write);
}
```

Figure 2.3: Standard read write lock solution, adapted from [7].

and the standard producer/consumer problem (both the consumer and producer need to write access) [8]. By using this kind of thread the program can be faster by allowing more concurrency. A standard implementation is stated in figure 2.3 (adapted from [7]). The standard approach has certain limitations (like non-reentrancy) the examined read-write lock has not.

2.2 Properties

The properties of the examined read write lock are described below.

Reentrant. A thread may call one of the lock methods multiple times, even when the thread has not fully unlocked the lock. A thread may only call one of the methods multiple times (not interleaved) until the lock is fully unlocked. This property is important for modulair programming. A function obtaining a lock can use other functions which also obtain the same lock. Figure 2.4 contains an example of this usage.

A read lock can be obtained after a write lock is already obtained by the same thread. When a write lock is obtained, this lock is also regarded as a read lock. Following this property a thread holding a write lock can execute reentrant (see previous property) read locks on the lock. The following sequence of lock operations is valid: `write lock, read lock, unlock, unlock`. This property is also important for modulair programming. It is possible to use functions requiring a read lock in a method which uses a

```
int determinant(Matrix *a) {
    readLock(a);
    // ... calculate Determinant ...
    unlock(a);
    return determinant;
}

// gives back the inverted matrix
Matrix *invertMatrix(Matrix *a) {
    readLock(a);
    int i = determinant(a);
    // ... really inverse matrix ...
    unlock(a);
    return inverseMatrix;
}
```

Figure 2.4: Example of the use of a reentrant lock.

write lock. The sequence of first a read lock and secondly a write lock (called upgradable read locks) is not possible because the chance of a deadlock situation is large (e.g. two threads with the same sequence). Figure 2.5 contains an example of the use of read locking a lock already write locked earlier.

Write locks have priority over read locks. There should be no situation where a certain thread cannot get a lock because of scheduling order. Each thread should make near constant progress and never stall. Read-write locks have two kinds of starvation, one with each kind of lock operation. Write lock priority results in the possibility of reader starvation: when constantly there is a thread waiting to acquire a write lock, threads waiting for a read lock on the same lock will never progress and visa versa. Most implementations give priority to write locks over read locks. This is mostly because write locks are more important, smaller, exclusive and occurs less. In the literature, this property is known as 'the second readers writers problem' [7].

The properties below are fundamental to the correct working of all read-write locks, not just the implementation used in this paper. These properties will be used in later sections to assert the reliability.

Exclusive access for threads with a write lock. It is important to only grant one and only one thread full access (by means of a write lock) and disallow any access from other threads. This is the same property as stating that while a thread has obtained a read lock no other thread may have obtained a write lock.

Deadlock free. No correct sequence of read locks, write locks and unlocks may result in a state where the program stalls, called a deadlock. Since it is possible to combine several locks and correct sequences of lock operations (obtaining a read or write lock, or releasing a lock), without resorting to complex synchronization , resulting in a program which can deadlock dependant on the scheduling order of the threads. An example of this is shown in figure 2.6. Even through this is not the scope of this thesis, solutions for

```
int determinant(Matrix *a) {
    readLock(a);
    // ... calculate Determinant ...
    unlock(a);
    return determinant;
}

// inverts the matrix in place
void invertMatrix(Matrix *a) {
    writeLock(a);
    int i = determinant(a);
    // ... really inverse matrix ...
    unlock(a);
}
```

Figure 2.5: Example of the use of a write lock being treated as a read lock.

this problem are available [7, 8]. Restricting the deadlock free property to the scope of this thesis, no deadlock may occur originating from the implementation of the read-write lock while executing correct sequences of lock operations.

2.3 Implementation

The examined read write lock has certain attributes/variables. The code of the implementation is listed in figure 2.7. The lock implementation is based on a condition lock, so the first attribute is a condition lock. It has a global lock count `lockCount` which count the number of locks (including reentrant locks) obtained from this lock. Also it tracks, by means of the `writeLockedBy` variable, if and which thread has the write lock. A write lock can only be obtained when the global lock count is zero (`lockCount == 0`) or the thread already has obtained a write lock (a reentrant write lock request, `writeLockedBy ==` current thread identifier). While a write lock request is pending (recorded in the variable `writersWaiting`), no read lock may be obtained because of the property to prioritize write requests. Also a read lock can only be obtained when no write lock is obtained from the lock. When requesting a read or write lock and one of the conditions obtaining is not met, the thread waits until the condition is satisfied. This is done by using the condition lock. During the course of this thesis an error in the implementation was found with help of the UPPAAL tool (see chapter 3) and the implementation is adjusted to correct the error. When a reentrant read lock is requested (by a thread already holding a read lock) while a write lock is pending a deadlock occurs. The read lock request cannot succeed because a write lock request is pending and the write lock request cannot succeed because the other thread still has a read lock. Example code for this deadlock is listed in figure 2.8. The solution is to let a reentrant read lock to be obtained always but a new read lock to be obtained only when no writers are waiting nor any thread has a write lock. Therefore the read write lock needs to maintain per thread state information. The implementation is adjusted accordingly.

The new version keeps track of the reentrant lock count (and the corresponding thread identifier) for each thread which obtained a lock. Therefore the new implementation is more

```
Lock *lockOne;
Lock *lockTwo;
void threadOne() {
    lock(lockOne);
    sleep(10);
    lock(lockTwo);
    unlock(lockTwo);
    unlock(lockOne);
}

void threadTwo() {
    lock(lockTwo);
    sleep(10);
    lock(lockOne);
    unlock(lockOne);
    unlock(lockTwo);
}
```

Figure 2.6: Example of a deadlock caused by multiple locks obtained in the wrong order.

complex. A write lock can only be obtained when if no locks have been obtained and thus the per thread information list is empty. A seperate `lockCount` is not needed and eliminated. Just like the old version the code tracks which thread has a write lock if obtained, so no read nor write locks by other threads can be obtained when this variable is set. To adhere to the property to prioritize write lock requests, just as in the old version, a counter with the number of waiting threads for a write lock is kept so no read lock can be obtained when the counter is above zero (unless it is a reentrant read lock). The simplified code (for just one lock) is listed in figure 2.9, while the total code is listed in appendix A.

```
ConditionLock *l;
int lockCount = 0;
ThreadID writeLockedBy = NONE;
int writersWaiting = 0;

void readLock() {
  lock(l);
  while (writersWaiting > 0 ||
            (writeLockedBy != NONE && writeLockedBy != currentThreadID))
    wait(l);
  lockCount++;
  unlock(l);
}

void writeLock() {
  lock(l);
  writersWaiting++;
  while (lockCount > 0 && writeLockedBy != currentThreadID)
    wait(l);
  writersWaiting--;
  writeLockedBy = currentThreadID;
  lockCount++;
  unlock(l);
}

void unlock() {
  lock(l);
  lockCount--;
  if (lockCount == 0) {
    writeLockedBy = NONE;
    unlockWithSignalAll(l);
  } else
    unlock(l);
}
```

Figure 2.7: Simplified code (for one lock) of the faulty implementation.

```
void threadOne() {
    readLock();
    sleep(20);
    readLock();
    // never reached due deadlock
}

void threadTwo() {
    sleep(10);
    writeLock();
    // never reached due deadlock
}
```

Figure 2.8: Code resulting in a deadlock when using the original version of the lock (using the simplified syntax of figure 2.7.

```
struct ThreadInfo {
   ThreadID id;
   unsigned int count;
   struct ThreadInfo *next;
};
ConditionLock *l = ...;
struct ThreadInfo *threads = NULL;
int writersWaiting = 0;
ThreadID writeLockedBy = NONE;
void newLock(ThreadID id, struct ThreadInfo **threadsPtr) {
        struct ThreadInfo *new = malloc(sizeof(struct ThreadInfo));
        new->id = id;
        new->count = 1;
        new->next = *threadsPtr;
        *threadsPtr = new;
}
int incLock(ThreadID id, struct ThreadInfo *thread) {
        if (thread == NULL)      return -1;
        if (thread->id == id)    return ++thread->count;
        return incLock(id, thread->next);
}
int decLock(ThreadID id, struct ThreadInfo **threadPtr) {
        struct ThreadInfo *current = *threadPtr;
        if (current == NULL)     return -1;
        if (current->id == id) {
                if (current->count > 1) return --current->count;
                *threadPtr = current->next;
                free(current);
                return 0;
        }
        return decLock(id, &current->next);
}
void readLock() {
   lock(l);
   // try to increase the reentrant lock count
   int retval = incLock(currentThreadID, threads);
   if (retval == -1) {
      // new thread (thread has not locked this lock)
      while (writeLockedBy != NONE || writersWaiting > 0)
            wait(l);
      newLock(currentThreadID, &threads);
   }
   unlock(l);
}
void writeLock() {
   lock(l);
   if (writeLockedBy == currentThreadID) {
      incLock(currentThreadID, threads);
   } else {
      writersWaiting++;
      while (threads != NULL)
         // there are other threads holding the lock, wait
         wait(l);
      writersWaiting--;
      newLock(currentThreadID, &threads);
      writeLockedBy = currentThreadID;
   }
   unlock(l);
}
void unlock() {
   lock(l);
   unsigned int count = decLock(currentThreadID, &threads);
   if (count == 0) {
      writeLockedBy = NONE;
      unlockWithSignallAll(l);
   } else
      unlock(l);
}
```


Figure 2.9: Simplified code (for one lock) of the correct implementation.

Chapter 3

Modelling in UPPAAL

> *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*
>
> – Brian W. Kernighan

3.1 UPPAAL

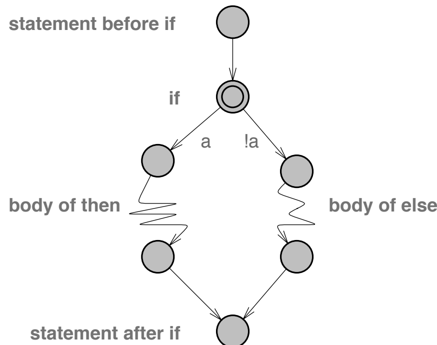
> Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.).
>
> The tool is developed in collaboration between the Department of Information Technology at Uppsala University, Sweden and the Department of Computer Science at Aalborg University in Denmark.

The above quote is from the UPPAAL website. UPPAAL is a tool written in Java to check models for certain properties. The model can be created in the GUI of UPPAAL, the properties are lines of text. A model is a final state automata. It consists of states, transitions and variables. A final state automata in UPPAAL is called a process. Multiple processes can be combined in UPPAAL into one UPPAAL file. The transitions can have conditions (called guards) if the transition can occur. Also they can alter the variables. A transition can be synchronized with another transition in another process (in the diagrams synchronization statements are the statements with an `?` or `!` suffix). In this case both transitions must occur at te same time. In this manner the two processes can communicate. Properties checked checked by UPPAAL are called queries and are specified in a specific query language.

3.2 Modelling Technique

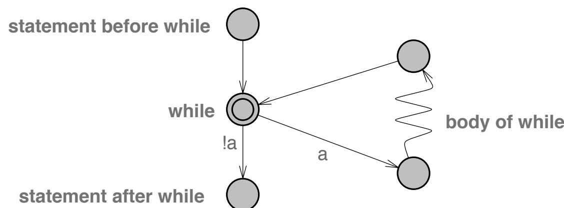
The following conversion method is used to create an UPPAAL model of the lock code. In case of an `if` control structure the code chooses to execute the `then` part if the condition of the `if` is true, otherwise the `else` part. An `if` control structure is represented in UPPAAL by two transition from a certain state, with transition conditions. The transition condition is

the same as in the `if` statement, one of them the complement of the condition. Statements in the body of the `then` or `else` part are represented by states and updates in transitions.




```
statement before if;
if (a) {
        body of then;
} else {
        body of else;
}
statement after if;
```

The `while` statement executes the body of the `while` while the condition is satisfied. A `while` statement can be modelled by state with two transitions, one from the state (to another state but eventually) to itself and one to another state, if the condition is not satisfied and the execution of the code continues with the next statement. The first (looping) transition contains the condition on which the while continues and leads to a state which executes the body of the while.




```
statement before while;
while (a) {
        body of while;
}
statement after while;
```

Non control statement execution can be modelled by an update to variables by means of a state transition. In UPPAAL each transition can have an effect when executed, like updating variables. Several updates can be combined in one transition (in UPPAAL a function needs to be created for this).

3.3 Modelling the Read-Write Lock

To model check, a model of a thread and the condition lock are made. Below are certain aspects of the model explained. The model of a thread is shown in figure 3.5 and the model of the lock is shown in figure 3.1. The different parts of the implementation are a straightforward conversion as described earlier, almost all variable names are the same. The different functions of the code are between the 'begin' and 'end' states. The different parts of the read write lock implementation are combined into one with a start state and several control variables (the control code used is listed in figure 3.4). These control variables added to the model constrain the possible operations: a thread is valid if and only if it adheres to the properties stated in section 2. A thread should only execute valid sequences of operations. A thread can obtain a write lock several (possibly zero) times and thereafter it can obtain a read lock several (possibly zero) times, all interleaved with several (possibly zero) times an unlock operation (note: if after n read lock operations n unlock operations are done, new write lock operations

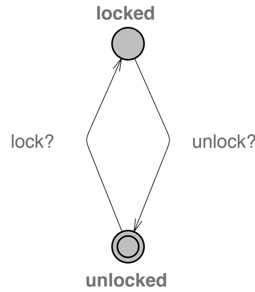


Figure 3.1: UPPAAL model of the condition lock, where the read write lock is based on.

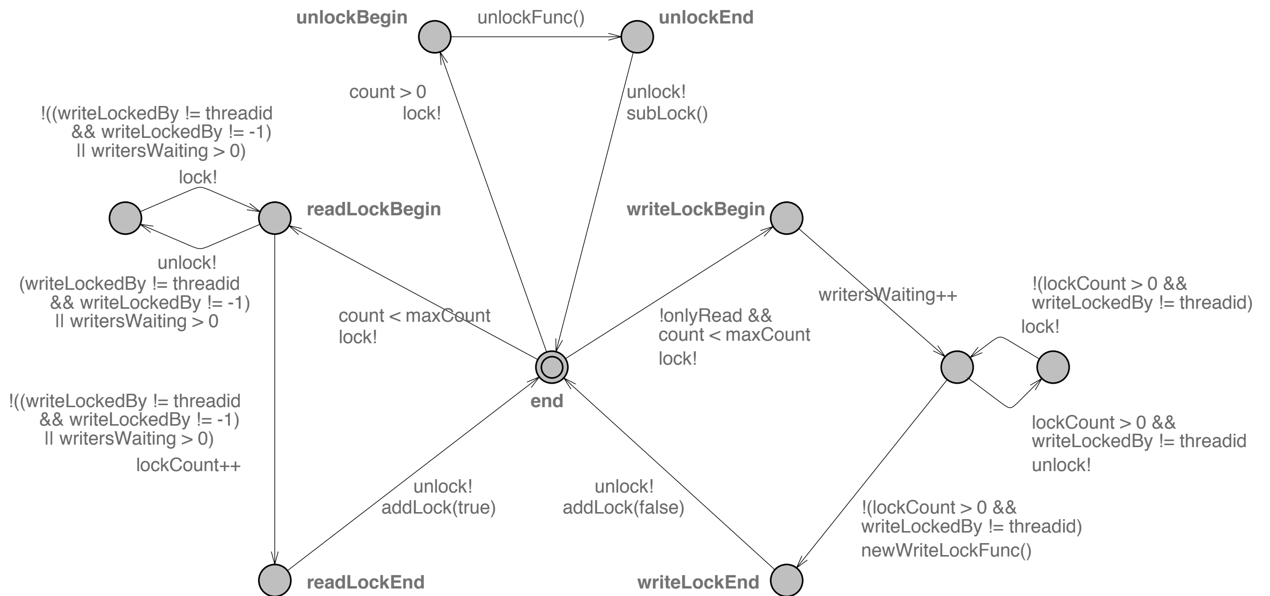


Figure 3.2: UPPAAL model of the original version of a thread with as identifier the variable '`threadid`'. The functions used are listed in figure 3.6 and 3.4.

may occur). To prevent a write lock request when a read lock is already obtained, the variable `onlyRead` must be false in the thread model for the transition to obtain a write lock. This variable is set when a read lock is obtained and unset when the correct number of unlocks are executed (equal to the number of read locks obtained). This way only valid sequences of locking operations will be model checked.

To test the model UPPAAL must simultaneously check multiple thread models and just one condition lock model. All possible valid scheduling orders of the threads must be checked. This non-deterministic behaviour of the code can be modeled by multiple state transitions which can occur at a given time. UPPAAL will check each possible combination of transitions. This way all possible scheduling orders are checked.

To communicate between the model of the condition lock and the model of the thread, synchronization on transitions is done. The condition lock has only two states and in this way only one thread model can be in a critical section, just like the implementation. If obtaining

```
int lockCount = 0;
int writeLockedBy = -1;
int writersWaiting = 0;

// real functions
void newWriteLockFunc() {
        writersWaiting--;
        lockCount++;
        writeLockedBy = threadid;
}

void unlockFunc() {
        lockCount--;
        if (lockCount == 0) {
                writeLockedBy = -1;
        }
}
```

Figure 3.3: Functions used in the original model of the thread, see 3.3.

```
// control part to only facilitate correct sequences of locking operations
int maxCount = 5;
int count = 0;
bool onlyRead = false;
int resetFlagOn = 0;

void addLock(bool setOnlyRead) {
        if (!onlyRead && setOnlyRead) {
                resetFlagOn = count;
        }
        onlyRead = onlyRead | setOnlyRead;
        count++;
}

void subLock() {
        count--;
        if (count == resetFlagOn)
                onlyRead = false;
}
```

Figure 3.4: Functions used in the both models of the thread, see 3.3.

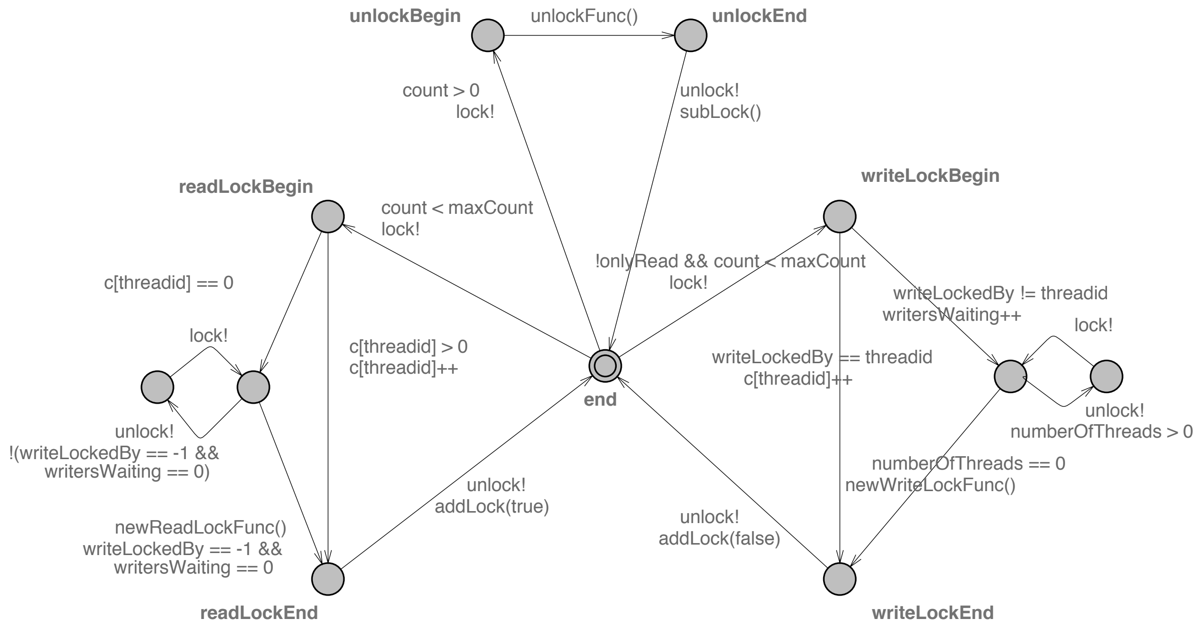


Figure 3.5: UPPAAL model of the adjusted version of a thread with as identifier the variable '`threadid`'. The functions used are listed in figure 3.6 and 3.4.

```
int numberOfThreads = 0;
int c[16];        // info per thread, index is threadid, contents is lock count for the thread
int writeLockedBy = -1; // -1 = none
int writersWaiting = 0;

// model of real thread rwlock functions
void newReadLockFunc() {
   c[threadid] = 1;
   numberOfThreads++;
}

void newWriteLockFunc() {
   writersWaiting--;
   c[threadid] = 1;
   numberOfThreads++;
   writeLockedBy = threadid;
}

void unlockFunc() {
   c[threadid]--;
   if (c[threadid] == 0) {
      numberOfThreads--;
      writeLockedBy = -1;
   }
}
```

Figure 3.6: Functions used in the model of the adjusted thread, see 3.5.

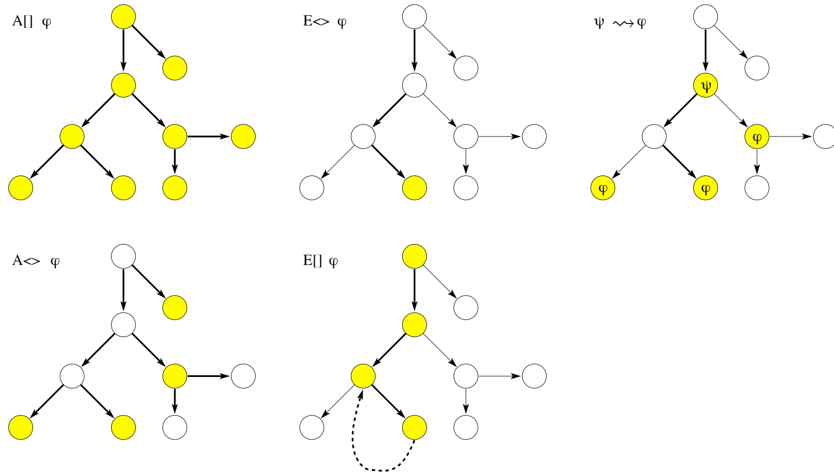


Figure 3.7: Different types of UPPAAL queries, used from [1]. The yellow states are the states in which the condition specified in the query must satisfy, otherwise the statement (consisting of the operator appended with the condition) is evaluated to false.

a lock fails the model retries until obtaining the lock succeeds, this is equivalent with sleeping of a thread in the implementation.

To limit the number of states (and prevent a state explosion) threads are modelled in restrained way. Every correct combination of read lock, write locks and unlocks is possible however only until a depth of 5 (read and/or write) locks. As the implementation of the lock no decision is made based on the depth of reentrant locks, this is a valid limitation. The number of threads is limited to 3 concurrent threads. This is also a reasonable limitation because every combination of states with different threads should occur. Wait/continue cycles which do not result in state changes are not executed so the model is easier checked. This also occurs in the code by the notification on the condition lock construct. The limits in the model of the thread are on the transitions to `readLockBegin`, `writeLockBegin` and `end`.

3.4 Checking the Read-Write Lock

UPPAAL has 5 different types of queries, listed in figure 3.7. These type of queries can be checked and validated. For example deadlock can be checked by a `A[]` query, stating that each state always must not deadlock. Most queries used are `A[]` queries, called 'safery properties'. [1] As many properties of section 2.2 should be modelled. The properties checked and its queries are shown below.

- No deadlocks. This is checked by a single query. All possible states of the model are checked if there is still a transition possible. This property does not exclude a deadlock by using multiple locks (like the example in figure ??).
  `A[] not deadlock`

- Exclusive access for thread with a write lock. The query checks that if a thread has a write lock obtained then the read/write lock data structure must have administrated

the thread identifier of the thread as the current thread that has the write lock. If two threads both has obtained a write lock, one of them is not registered. Since this property validates this is not the case.

```
A[] (thread1.end and (not thread1.onlyRead) and thread1.count > 0)
   imply writeLockedBy == thread1.threadid
```

- Consistency checks (e.g., thread count ≥ 0). Many consistency checks were checked, but not listed.

During model checking the deadlock query was not satisfied for the first version of the code. UPPAAL gives a state of the model when the deadlock occurs. This error is described in section 2.3. The second version of the code satisfied the deadlock query as well as the other queries.

Chapter 4

Other implementations

If there is a wrong way to do something, then someone will do it.

– Edward A. Murphy Jr.

The properties of the lock noted in section 2.2 distinguish this type of lock from other read write locks. Other such locks looked into are standard examples from books (they are often not reentrant), the implementation of Qt (base library of KDE, Google Earth, Mathematica and others) by Trolltech (read lock after write lock is not valid) and the Posix Read/Write lock (read lock after write lock is not valid). The properties of each implementation are listed in figure 4.1.

While analysing the source code of Trolltech's Qt 4.3, exact the same error was found as the error noted in section 2.3. Its documentation showed it was a valid situation and the deadlock possibility was not mentioned. The bug in the implementation was reported to Trolltech and will be fixed in the next release of Qt, version 4.4. The Posix version suffers from the same error, but mentions there is a special error code for the situation, although users are not explicit warned in the main documentation. Other implementation differs significantly (no condition lock) or are too large to analyse (like Java) so no useful comparison can be made.

Which	1	2	3	4	5	6
PIToolkit (this implementation)	×	×	×	×		
Qt 4.3	×			×	×	
Standard implementation in figure 2.3	only read					
Posix Threads Read/Write Lock	only read			×	×	×

Number	Legend
1	Reentrant
2	Read lock after write lock is valid
3	One unlock method
4	Writers get priority
5	Deadlock when reentrant requesting a read lock while a write lock is pending
6	Deadlock noted in the documentation

Figure 4.1: Properties of other implementations.

Chapter 5

Future Work

The required techniques of effective reasoning are pretty formal, but as long as programming is done by people that don't master them, the software crisis will remain with us and will be considered an incurable disease. And you know what incurable diseases do: they invite the quacks and charlatans in, who in this case take the form of Software Engineering gurus.

– Edsger Dijkstra [3]

To proof a certain piece of software correct is time costly. To assist and minimize errors a proof assistant, like PVS, can be used. To use a tool like PVS a piece of software must be translated in the modelling language used by PVS. PVS has a functional modeling language with a number of limitation on mutual recursive data types. Because of the functional language and its limitations it is not trivial to model the concurrency of the imperative code. A solution is to model threads as a list of actions, a state (waiting, running), a thread identifier (an integer) and a count of lock recursion of the thread. Valid actions for the thread are `readLock`, `writeLock` and `unlock`. They must occur in a valid order. A model should abstract from all other statements in the code of threads, so no work is spent on non-relevant details (just like the UPPAAL model). The state is 'running' (first action in the list not yet tried), 'waiting' (first action tried but failed) or 'error' (occurs when unlocking an unlocked lock). The model described was made, but due time constrains the model was not proved. Also the model is not optimal to use in a proof assistant as PVS and should be adjusted. It is left for future work. The model is listed in appendix B.

Chapter 6

Conclusion

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

– C.A.R. Hoare [4]

The results of this paper consists of finding a serious bug in two implementations of a read write lock (my own and Trolltech's Qt), one of which is widely used (Qt). After reporting the bug is being fixed. Writing of concurrent code is hard. Testing concurrent code can not guarantee the absence of deadlocks or other bugs. Concurrent code should be modelled in a tool like UPPAAL, so the reliability is better known. Modelling in UPPAAL is not costly, it is relatively simple and the results are encouraging.

By correcting the implementation and checking the model a reliable read write lock is created. This read write lock is reentrant and allows a read lock to be obtained after a write lock has already been obtained by the same thread. This combination of properties is unique. Because it is based on a wide-available non-reentrant condition lock, the read write lock can be implemented on a broad range of platforms and languages.

Bibliography

[1] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[2] Edsger W. Dijkstra. The humble programmer (ewd 340). Commun. ACM, 15(10):859–866, 1972.

[3] Edsger W. Dijkstra. Answers to questions from students of software engineering. circulated privately, November 2000.

[4] Charles Antony Richard Hoare. The emperor's old clothes. Commun. ACM, 24(2):75–83, 1981.

[5] Merle Johnson. More Maxims of Mark. 1927.

[6] Friedrich Nietzsche. Daybreak Thoughts on the Prejudices of Morality. 1881.

[7] Abraham Silberschatz and Peter Baer Galvin. Operating System Concepts, 5th Ed. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[8] William Stallings. Operating Systems, Internals and Design Principles, 5th Ed. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

Appendix A

Final Version - C Code

```
// ConditionLock.h - Header
typedef char BOOL;
#define YES 1
#define NO 0

struct ConditionLock {
    ...
};
typedef struct ConditionLock ConditionLock;
ConditionLock *conditionlockNew();
void conditionlockDealloc(ConditionLock * obj);

BOOL conditionlockWriteLock(ConditionLock *lock, BOOL blocking);
void conditionlockUnlock(ConditionLock *lock);
void conditionlockUnlockWithSignalAll(ConditionLock *lock, BOOL signalAll);

// RWLock.h - Header
#define RWLOCK_MAXREADERS 16

struct ThreadInfo {
    pthread_t         threadID;
    pword_u             count;
};

struct RWLock {
    ConditionLock      *lock;
    // limited to RWLOCK_MAXREADERS so it has
    // O(1) in place of O(n) with n = number of threads
    struct ThreadInfo      threads[RWLOCK_MAXREADERS];
    unsigned char      numberOfThreads;
    unsigned char      writersWaiting;
    unsigned char      writeLockedBy;
    // if max -> not writelocked, if < max -> numberOfThreads == 1
};
typedef struct RWLock RWLock;

RWLock *rwlockNew();
void rwlockDealloc(RWLock * obj);

// locking methods,
// if blocking is false always returns true,
// if blocking is true returns if locking operation succeeded
BOOL rwlockReadLock(RWLock *lock, BOOL blocking);
BOOL rwlockWriteLock(RWLock *lock, BOOL blocking);
void rwlockUnlock(RWLock *lock);

// RWLock.c - Implementation
#include "PRWLock.h"

int indexOfThread(struct ThreadInfo threads[], pint_u numberOfThreads,
                                                    pthread_t threadID) {
    int i;
    for (i = 0; i < RWLOCK_MAXREADERS && numberOfThreads > 0; i++) {
        if (threads[i].count > 0) {
            if (threads[i].threadID == threadID)
                return i;
            numberOfThreads--;
        }
    }
    return RWLOCK_MAXREADERS;
}

// returns a free spot in threads, otherwise returns RWLOCK_MAXREADERS
// also increases the number of threads
int freeSpot(struct ThreadInfo threads[], pchar_u *numberOfThreadsPtr) {
    int i;
```

```
    for (i = 0; i < RWLOCK_MAXREADERS; i++) {
        if (threads[i].count == 0) {
            (*numberOfThreadsPtr)++;
            return i;
        }
    }
    return RWLOCK_MAXREADERS;
}

// set spot to a certain value
void setSpot(struct ThreadInfo threads[], pchar_u current,
                                        pthread_t threadID, pchar_u count) {
    assert(current < RWLOCK_MAXREADERS);
    threads[current].threadID = threadID;
    threads[current].count = count;
}

RWLock *rwlockNew() {
    RWLock *obj = (RWLock*)malloc(sizeof(RWLock));
    if (obj) {
        obj->lock = conditionlockNew();
        obj->numberOfThreads = 0;
        obj->writersWaiting = 0;
        obj->writeLockedBy = RWLOCK_MAXREADERS;
        int i = 0;
        for (i = 0; i < RWLOCK_MAXREADERS; i++) {
            obj->threads[i].count = 0;
        }
    }
    return obj;
}

void rwlockDealloc(RWLock * obj) {
    if (obj == NULL)
        return;
    conditionlockDealloc(obj->lock);
    free(obj);
}

BOOL rwlockReadLock(RWLock *self, BOOL blocking) {
    if (!conditionlockLock(self->lock, blocking))
        return NO;

    BOOL retval = YES;
    pthread_t threadID = pthread_self();

    pint_u current = indexOfThread(threads, numberOfThreads, threadID);

    if (current < RWLOCK_MAXREADERS) {
        // found (current thread has already a lock)
        threads[current].count++;
    } else {
        // new thread (thread has not locked this lock)
        while ( !(retval = numberOfThreads < RWLOCK_MAXREADERS && // no free spots
            writeLockedBy == RWLOCK_MAXREADERS && // not write locked
            writersWaiting == 0) & blocking) {     // no writers waiting
               conditionlockWait(self->lock);
        }
        if (retval)
            setSpot(threads, freeSpot(threads, &numberOfThreads),
                threadID, 1);
    }
    conditionlockUnlock(self->lock);
    return retval;
}
```

```
BOOL rwlockWriteLock(RWLock *self, BOOL blocking) {
    if (!conditionlockLock(self->lock, blocking))
        return NO;

    BOOL retval = YES;
    pthread_t threadID = pthread_self();

    if (writeLockedBy < RWLOCK_MAXREADERS &&
                threads[writeLockedBy].threadID == threadID) {
        threads[writeLockedBy].count++;
    } else {
        writersWaiting++;
        // if there are others wait
        while (!(retval = numberOfThreads == 0) & blocking) {
                conditionlockWait(self->lock);
        }
        writersWaiting--;
        if (retval) {
            setSpot(threads, writeLockedBy = freeSpot(threads, &numberOfThreads),
                threadID, 1);
        }
    }
    conditionlockUnlock(self->lock);
    return retval;
}

BOOL rwlockUnlock(RWLock *self) {
   conditionlockLock(self->lock, YES)

    pthread_t threadID = pthread_self();

    pint_u current = indexOfThread(threads, numberOfThreads, threadID);

    assert(current < RWLOCK_MAXREADERS);

    threads[current].count--;
    unsigned int count = threads[current].count;

    if (count == 0) {
        threads[current].threadID = 0;
        numberOfThreads--;
        writeLockedBy = RWLOCK_MAXREADERS;
    }
    conditionlockUnlockWithSignalAll(self->lock, count == 0);
}
```

Appendix B

PVS model

```
List[T: TYPE]: THEORY
BEGIN
   pos(e: T, l: list[T]) : RECURSIVE nat =
          CASES l OF
                  null: 0,
           cons(hd, tl):
                    IF hd = e THEN 0 ELSE 1 + pos(e, tl) ENDIF
      ENDCASES
   MEASURE length(l)

   replace(l: list[T], i: below[length(l)], e: T) : RECURSIVE list[T] =
         CASES l OF
               null: null,
               cons(car, cdr):
                     IF i = 0
               THEN
                  cons(e, cdr)
               ELSE
                  cons(car, replace(cdr, i-1, e))
               ENDIF
         ENDCASES
      MEASURE length(l)

         flatten(l:list[list[T]]) : RECURSIVE list[T] =
                      CASES l OF
                            null: null,
                            cons(x,xs): append(x, flatten(xs))
                      ENDCASES
            MEASURE length(l)

   insert(l: list[T], i:below[length(l)], e: T) : RECURSIVE list[T] =
        IF i = 0 THEN
           cons(e, l)
        ELSE
         CASES l OF
                 null: null,
               cons(hd, tl): cons(hd, insert(tl, i-1, e))
         ENDCASES
      ENDIF
      MEASURE length(l)
END List

RWLockImp : THEORY
BEGIN
   % record
   RWLock : TYPE = [#
      numberOfThreads: nat,
      writeLockedBy: nat,
      writersWaiting: nat
      #]

   aLock: VAR RWLock

   % set
   Status: TYPE = {
      WAIT,
      RUN,
      SIGNAL,   % SPECIAL CASE OF RUN
      ERROR
      }

   % set
   Action: TYPE = %[[RWLock, Thread] -> LockCmdReturn]
      {READLOCK, WRITELOCK, UNLOCK}

   % record
```

```
Thread : TYPE = [#
   actions: list[Action],
   status: Status,
   threadId: nat,
   count: nat
   #]

% tuple
ActionReturn: TYPE = [
         thread: Thread,
         lock: RWLock
         ]

IMPORTING List[Thread]
IMPORTING list_props[Thread]

% model of real code
readLock(t: Thread, l : RWLock) : ActionReturn =
   IF t`status = ERROR THEN (t, l) ELSE
   IF t`status = RUN
   THEN
      % RUN
      IF t`count > 0 or (l`writeLockedBy = 0 and l`writersWaiting = 0)
      THEN
         (t with [actions := cdr(t`actions), count := t`count+1],
         l with [numberOfThreads := l`numberOfThreads + (IF t`count > 0 THEN 0 ELSE 1 ENDIF)])
      ELSE
         (t with [status := WAIT], l)
      ENDIF
   ELSE
      % WAIT
      IF (l`writeLockedBy = 0 and l`writersWaiting = 0)
      THEN
         (t with [actions := cdr(t`actions), count := t`count+1, status := RUN],
         l with [numberOfThreads := l`numberOfThreads + 1])
      ELSE
         (t, l)
      ENDIF
   ENDIF
   ENDIF

writeLock(t: Thread, l : RWLock) : ActionReturn =
   IF t`status = ERROR THEN (t, l) ELSE
   IF t`status = RUN
   THEN
      % RUN
      IF l`writeLockedBy = t`threadId or l`numberOfThreads = 0
      THEN
         (t with [actions := cdr(t`actions), count := t`count+1],
         l with [numberOfThreads := l`numberOfThreads + (IF l`writeLockedBy = t`threadId THEN 0 ELSE 1 ENDIF)])
      ELSE
         (t with [status := WAIT],
         l with [writersWaiting := l`writersWaiting+1])
      ENDIF
   ELSE
      % WAIT
      IF  l`numberOfThreads = 0
      THEN
         (t with [actions := cdr(t`actions), count := t`count+1, status := RUN],
         l with [numberOfThreads := l`numberOfThreads + 1, writeLockedBy := t`threadId, writersWaiting := l`writersWait
      ELSE
         (t, l)
      ENDIF
   ENDIF
   ENDIF
```

```
unlock(t: Thread, l : RWLock) : ActionReturn =
   IF (t`count > 0 AND not (t`status = ERROR))
   THEN
      (t with [count := t`count - 1],
      l with [numberOfThreads := l`numberOfThreads - (IF t`count = 1 THEN 1 ELSE 0 ENDIF), writeLockedBy := IF t`count
   ELSE
      (t with [status := ERROR], l)
   ENDIF

% end of model

% execute an action in a specific thread
executeThread(t: Thread, l: RWLock) : ActionReturn =
   IF not null?(t`actions) THEN
   CASES car(t`actions) OF
         READLOCK: readLock(t, l),
         WRITELOCK: writeLock(t, l),
         UNLOCK: unlock(t, l)
   ENDCASES
   ELSE
      (t with [status := ERROR], l)
   ENDIF

% BOOL indication if there are no actions left in the threads
noActionsInThreads(threads: list[Thread]) : RECURSIVE bool =
   CASES threads OF
         null: true,
         cons(car, cdr): length(car`actions) = 0 and noActionsInThreads(cdr)
   ENDCASES
MEASURE length(threads)

% check if one of the threads has an specific status
threadsStatusExists(threads: list[Thread], status: Status) : RECURSIVE bool =
   CASES threads OF
         null: false,
         cons(car, cdr): car`status = status or threadsStatusExists(cdr, status)
   ENDCASES
MEASURE length(threads)

% BOOL indicating if a specific thread is finished
threadDone(thread: Thread) : bool =
         length(thread`actions) = 0 and thread`status = RUN

% BOOL indicating if all threads are finished
threadsDone(threads: list[Thread]) : RECURSIVE bool =
   CASES threads OF
         null: true,
         cons(thread, cdr):  threadDone(thread) and threadsDone(cdr)
   ENDCASES
MEASURE length(threads)

% returns a `random'  number below the argument
shuffle(l: nat) : {n: nat | n < l}

% return the total number of actions in the threads
numberOfActionsInThreads(threads: list[Thread]) : RECURSIVE nat =
      CASES threads OF
          null: 0,
          cons(thread, tail): length(thread`actions) + numberOfActionsInThreads(tail)
      ENDCASES
      MEASURE length(threads) %length(threads)

% returns the number of threads that are runnable (status = RUN or SIGNAL)
%numberOfThreadsRunnable(threads: list[Thread]) : RECURSIVE nat =
%      CASES threads OF
%          null: 0,
```

```
%          cons(thread, tail): (IF thread`status = RUN or thread`status = SIGNAL THEN 1 ELSE 0 ENDIF) + numberOfThreads
%      ENDCASES
%      MEASURE length(threads) %length(threads)
% SAME AS length(activeThreads(arg))

% returns all threads that are runnable
activeThreads(threads: list[Thread]) : RECURSIVE list[Thread] =
      CASES threads OF
            null: null,
            cons(thread, tail): LET tail2 = activeThreads(tail) IN IF (thread`status = RUN or thread`status = SIGNAL) a
      ENDCASES
      MEASURE length(threads);

% checks if the lock is returned `empty'
unlocked(l: RWLock): bool =
       l`numberOfThreads = 0

% checks if all threads can be `run'
run(threads: list[Thread], l: RWLock) : RECURSIVE bool =
   (threadsDone(threads) and unlocked(l)) or (threadsStatusExists(threads, RUN) and
         (
         LET i = shuffle(length(activeThreads(threads))) IN
         LET e = nth(activeThreads(threads), i) IN
         LET retval = executeThread(e, l) IN
         run(replace(threads, i, retval`1), retval`2)
         ))
MEASURE lex2(numberOfActionsInThreads(threads), length(activeThreads(threads)))

% `runs' a specific thread
runnable(thread: Thread, l: RWLock) : RECURSIVE bool =
   (threadDone(thread) AND unlocked(l)) OR (thread`status = RUN and
         (
         LET retval = executeThread(thread, l) IN
         runnable(retval`1, retval`2)
         ))
MEASURE length(thread`actions)

% te bewijzen:
% - als writelock dan geen readlock die erdoorheen komt
% - max 1 writelock
% - na n locks en n unlocks een lege lock

aThread: VAR Thread
aThread2: VAR Thread

%exclusiveWriteLock: LEMMA
simpleTest: LEMMA % :)
   unlock(RUN, 1,
      readLock(RUN, 1,
         aLock with [count := 0, writeLocked := false, writersWaiting := 0, threadId := 0]
      )`2
   )`2
   =
   aLock with [count := 0, writeLocked := false, writersWaiting := 0, threadId := 0]

simpleTest2: LEMMA % :)
   unlock(RUN, 1,
      writeLock(RUN, 1,
         aLock with [count := 0, writeLocked := false, writersWaiting := 0, threadId := 0]
      )`2
   )`2
   =
   aLock with [count := 0, writeLocked := false, writersWaiting := 0, threadId := 0]

simpleTest3: LEMMA % :)
      LET lock = aLock with [count := 1, writeLocked := false, writersWaiting := 0, threadId := 0] IN
```

```
        LET thread1 = aThread with [status := RUN, actions := cons(UNLOCK, cons(UNLOCK, null)), threadId := 1] IN
        LET lockFinal = aLock with [count := 0, writeLocked := false, writersWaiting := 0, threadId := 0] IN
        LET threadFinal = aThread with [status := RUN, actions := cons(UNLOCK, null), threadId := 1] IN
        executeThread(lock, thread1) = (lockFinal, threadFinal)

  simpleTest4: LEMMA % :)
        LET lock = aLock with [count := 0, writeLocked := false, writersWaiting := 0, threadId := 0] IN
        LET thread1 = aThread with [status := RUN, actions := null, threadId := 1] IN
        LET threads = cons(thread1, null) IN
        run(lock, threads) %, {n: nat | n < length(threads)}) = true

  simpleTest5: LEMMA % :)
        LET lock = aLock with [count := 0, writeLocked := false, writersWaiting := 0, threadId := 0] IN
        LET thread1 = aThread with [status := RUN, actions := cons(READLOCK, cons(UNLOCK, null)), threadId := 1] IN
        LET threads = cons(thread1, null) IN
        run(lock, threads) %, {n: nat | n < length(threads)}) = true

  tail: VAR list[Action]
  lockCount: VAR nat
  writersWaiting: VAR nat
  aThreadId: VAR {x: nat | x > 0}

  % acties die kunnen gebeuren bij een lege lock
  runLemma: LEMMA
       FORALL (thread: Thread) :
       LET lock = aLock with [count := 0, writeLocked := false, writersWaiting := writersWaiting, threadId := 0] IN
       runnable(lock, thread) IMPLIES (thread`actions = null OR
                  writersWaiting = 0 AND thread`actions = cons(READLOCK, tail) OR
               thread`actions = cons(WRITELOCK, tail))

  % acties die kunnen gebeuren bij een gereadlockde lock
  runLemma2: LEMMA
       FORALL (thread: Thread) : lockCount > 0 AND
       LET lock = aLock with [count := lockCount, writeLocked := false, writersWaiting := writersWaiting, threadId := 0]
       runnable(lock, thread) IMPLIES ((writersWaiting = 0 AND thread`actions = cons(READLOCK, tail)) OR
               thread`actions = cons(UNLOCK, tail))

  % acties die kunnen gebeuren bij een gewritelockde
  runLemma3: LEMMA
       FORALL (thread: Thread) : lockCount > 0 AND
       LET lock = aLock with [count := lockCount, writeLocked := true, writersWaiting := writersWaiting, threadId := aThr
       runnable(lock, thread) IMPLIES (thread`threadId = aThreadId AND thread`actions = cons(READLOCK, tail)) OR
                  (thread`threadId = aThreadId AND thread`actions = cons(WRITELOCK, tail)) OR
               thread`actions = cons(UNLOCK, tail)

  actions: VAR list[Action]

  oneThreadProof: LEMMA
        %FORALL (actions1: list[Action]) :
        %LET lock = aLock with [count := 0, writeLocked := false, writersWaiting := 0] IN
        %LET thread1 = aThread with [status := RUN, actions := actions, threadId := 1] IN
        LET threads = cons(aThread, null) IN
        runnable(aLock, aThread) IMPLIES run(aLock, threads)

  twoThreadProof: LEMMA
        %FORALL (actions1: list[Action]) : FORALL (actions2: list[Action]) :
        %LET lock = aLock with [count := 0, writeLocked := false, writersWaiting := 0] IN
        %LET thread1 = aThread with [status := RUN, actions := actions1, threadId := 1] IN
        %LET thread2 = aThread with [status := RUN, actions := actions2, threadId := 2] IN
        LET threads = cons(aThread, cons(aThread2, null)) IN
        (runnable(aLock, aThread) AND runnable(aLock, aThread2) AND not (aThread`threadId = aThread2`threadId)) IMPLIES r

END RWLockImp
```