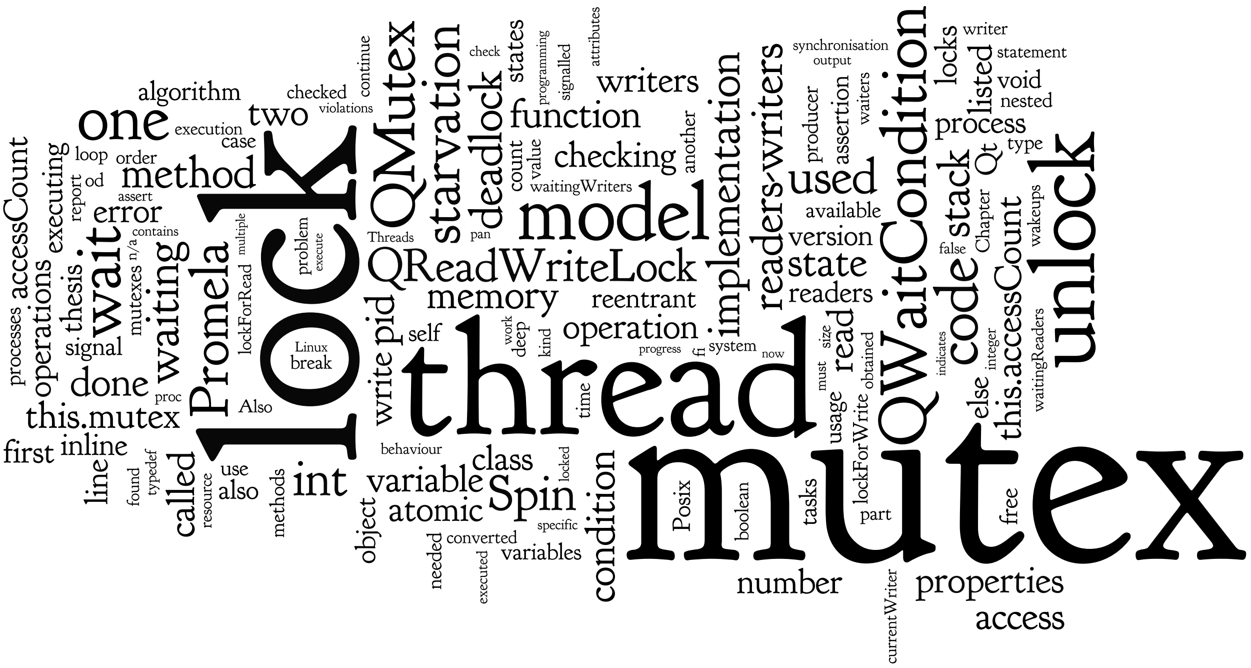

Verifying Reentrant Readers-Writers

Bernard van Gastel

Supervised by
dr. Sjaak Smetsers
and
prof. dr. Marko van Eekelen

Master's Thesis
Computer Science
Thesis Number 623
March 2010

Radboud University Nijmegen

This thesis is licensed under the
Creative Commons Naamsvermelding 3.0 Nederland
license.

The figure on the front page is a word cloud containing the most used words in this thesis (excluding common English words). The bigger the word the more used. The figure was created using the website `http://www.wordle.net`.

The figure is licensed under the *Creative Commons Attribution 3.0* license.

Abstract

The readers-writers algorithm is a widely used mutual exclusion mechanism in concurrent programming. Several versions of the algorithm exist. This thesis examines the algorithm used by Nokia's Qt framework. The algorithm is checked for deadlock and starvation issues using the Spin model checker. Previous work found a deadlock in the algorithm, and proposed and verified a corrected algorithm. The used model was limited. In this thesis, we reverify the results using a more detailed model. While verifying the algorithm for absence of starvation, no issues were found in the algorithm itself. However, the condition variables used by Qt's implementation have algorithmic starvation issues which makes the readers-writers implementation prone to starvation. A new algorithm for a starvation-free condition variable is presented and verified. This condition variable is constructed out of a starvation-prone condition variable, and is therefore applicable to similar situations were only a starvation-prone condition variable is available.

Contents

Chapter 1

Introduction

There is an increasing interest in concurrent programming with current development of consumer priced multicore processors. Yet the reliability of concurrent algorithms is hard to determine as it is hard to reproduce race conditions and to reason about these algorithms [26]. These difficulties are caused by the concurrency used. In this thesis an industrial-used *reentrant readers-writers* algorithm and an implementation thereof are verified. The algorithm is a *mutual exclusion algorithm.* Such algorithms are used to serialise access to resources (memory, peripherals, etc). The implementation of the algorithm is part of Nokia's popular Qt toolkit, used in programs like Google Earth, Opera and Skype [41].

The algorithm is verified using the widely used formal method *model checking* [7, 24]. Model checking can be performed automatically on a model. User-specified properties are checked to be valid for a given model. If a counterexample for such a property is found, a trace is produced by the model checker. Such a trace indicates the exact order of events leading to the violation of the property. Model checking suffers from the state-space explosion problem and requires a closed and finite system.

This thesis continues with earlier work[1] on increasing the reliability of the readers-writers algorithm [16, 17]. A deadlock was found and a corrected version of the algorithm was verified. Although positive results were found, the work left room for improvement. The model was very abstract: important supporting classes were not modelled and parts of the model were not entirely accurate. Although the absence of deadlocks was verified, the absence of starvation was not verified. Also, only a limited amount of safety properties were checked.

To introduce the subject matter, an overview of multithreading and mutual exclusion, including readers-writers, is given in Chapter 2. Qt's implementation of the algorithm is discussed in detail in Chapter 3. Abstract versions of operating system primitives used in the supporting classes of the algorithm are modelled in Chapter 4, along with an example verification of a concurrent stack.

This thesis's contribution is the improvement on the issues in the earlier work

[1] For my course Research 2, I converted the Uppaal models of [16] and [17] to Spin and reverified the result, as Spin is better suited for this kind of problems [24]. For the course I handed in the last draft of my contribution to [18] prior to the editing process with the other authors. Additionally I gave a 15 minute presentation of Research 2 about my own contributions. For clarity reasons, this work is also included in this thesis.

mentioned above. We model QT's readers-writers implementation more accurately and model QT's *conditional variable* on which the algorithm depends. Also, using the model of QT's condition variables, we reverify the earlier found deadlock in Chapter 5. The corrected readers-writers algorithm is presented in Chapter 6 and reverified, again using the model of QT's condition variables (Section 6.2). Before starvation can be considered, the abstract operating system primitives must be adjusted to be free of starvation (Section 6.4). Next, the implementation and is checked for starvation issues (Sections 6.5). In Chapter 7, the cause of the deadlock is analysed. The algorithm is successfully verified for absence of starvation (Section 7.1). Based on this result it is shown that QT's condition variables are flowed: they are inherently prone to starvation and must be replaced. A new condition variable build on a starvation-prone condition variable is presented and is verified to be free of starvation. Using the new starvation-free condition variable, the readers-writers implementation is also verified free of starvation. Finally, related and future work is discussed in Chapter 8, and concluding remarks are found in Chapters 9.

Part of the results of this thesis are also reported in [18]. The sources of all models and a tool for easy batch checking of PROMELA models are available online at `https://www.bitpowder.com/~bvgastel/research/masterthesis`.

Chapter 2

Background of concurrency

Computers are able to execute multiple tasks *concurrently.* This enables applications to become more responsive and increase performance. With the recent development of consumer multi-core and multi-processor machines, multithreaded programming is used to fully exploit the maximal performance of a machine. The down side of concurrent programming is increased complexity. In practice many errors are introduced. In this chapter we will introduce the concept of *threading* and how a program can use it on Unix systems. We explain common pitfalls which are introduced by threading. Next we describe a number of synchronisation methods, both for access to resources and for synchronisation between threads. To get some idea how these mechanisms are used, an implementation of a common data type is described.

2.1 Threading

Computers execute multiple tasks concurrently. These tasks are called *processes.* The operating system executes a specific process a limited amount of time before switching to another process. Eventually the process is scheduled again for execution. The process is not aware of a context switch. This is called *preemptive multitasking.* The switching may occur even when the process has not finished execution yet. Processes function as a whole unit, isolated from other processes. Their memory space is separate from other processes, i.e. processes can only read and change their own memory. Inter Process Communication (IPC) mechanisms offer structured ways to communicate with other processes. IPC can be used to exchange information, draw to the screen (managed by a windowing process), get updates on system events, etc. If a program generates an unrecoverable error, e.g. reading from a non existent memory location, the process is terminated. But a process is isolated from the errors occurring in other processes. The notable exception is if two processes communicate with each other and one of them crashes. This of course influences the other process too.

Modern operating systems support running multiple tasks concurrently in a specific process. These tasks are called *threads.* They are executed and preempted in the same way as processes. The primary advantage is that the threads reside in the same memory space. Therefore costly IPC mechanisms can be avoided.

```
int       pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                void *(*start_routine)(void*), void *arg);
void      pthread_exit(void *value_ptr);
int       pthread_join(pthread_t, void **);
pthread_t pthread_self(void);
```

Figure 2.1: The most important function prototypes of the Posix.1c standard.

But if one of the threads makes an unrecoverable error, the whole process is terminated, including all its threads. Depending on the operating system which is used, there are several *scheduling policies* for threads. These policies regulate the order of execution of threads.

In this thesis we use the Portable Operating System Interface [for Unix] (Posix) standard. This standard defines a platform-independent Application Programming Interface (API) for all sorts of operations, including process creation, signal delivery, timers, pipes, etc. A special part of the standard, called Posix.1c, defines functions for thread creation and manipulation. The most important functions are listed in Figure 2.1. The specification also includes standards for thread scheduling and thread synchronization. To create a thread, the function `pthread_create` is used. The first argument of the function is a thread control structure, allowing later manipulation of the thread. The second argument contains attributes, including scheduling behaviour, priority, stack size, etc. The last two arguments specify a function and an argument to that function. The specified function is executed in the newly created thread. It is also possible to wait for another thread to finish execution with the `pthread_join` function. Good introductions to Posix multithreaded programming are available online, e.g. [40].

If threading is used, problems can arise when a resource, i.e. data structure or peripherals, is accessed concurrently. Traditionally, each task (process or thread) expects that a resource is changed by itself, but not by other tasks. E.g. a method that reads some attributes of an object, calculates some sort of successor and stores the result. If two such operations execute concurrently, it is possible that both threads read the same information, both calculate the next values, and both write their results at the same time. The effect of one of the operations is lost. Even simple objects are prone to these problems. To avoid this kind of errors, a synchronisation mechanism must be implemented. Such mechanism controls access to a shared resource, e.g. a data structure. This effectuates that only one operation can run concurrently on a shared resource. But such mechanisms introduce another class of potential errors. These are described in the next section.

2.2 Potential synchronisation pitfalls

There are roughly two kinds of problems for synchronisation mechanisms: *starvation* and *deadlock*. They both influence the program in which they manifest, but in different ways.

The first common pitfall is *deadlock*. This is a situation in which a number of threads can not make progress. E.g. two threads both need exclusive access to two resources before they can finish execution. But both threads have exclusive access to one of the two resources, disallowing the other one to finish executing.

So both threads are waiting for each other. This situation is called a *deadlock*. More generally: a state where multiple tasks are stalled forever due to activities of other tasks. A variant of deadlock is *livelock*. The difference with deadlock is that the threads still actively try to obtain exclusive access to the resource. But instead of being descheduled in anticipation of obtaining the access, tries continually to obtain the lock. This costs many processor cycles.

The other problem is called *starvation*. This happens if one of the threads is favoured over other threads consistently. E.g. if both threads require exclusive access, the access is always given to one thread. The other thread can not get any resources. Therefore it can not make any progress, i.e. it starves.

A good example illustrating deadlock and starvation is the Dining Philosophers Problem [42, 44]. This classical problem is a retold version by Tony Hoare of a problem described by Dijkstra.

2.3 Access synchronisation mechanisms

There are several options to regulate access to shared resources. The access synchronisation methods include *mutexes*, *semaphores* and finally *readers-writers locking*. In this section, we give a description of these methods, from low to high complexity. With all these methods there are two types of implementations: starvation-free locking and opportunistic locking. The first one tracks the order in which threads request the synchronisation primitive. The order in which access to a resource is requested, is maintained. This is starvation free. But this comes with a performance penalty: the queue has to be maintained and more switches between threads are needed. The latter implementation is called opportunistic locking: if a synchronisation primitive is available for locking the thread locks it, even if other threads are waiting for it. This implementation has higher performance but suffers from starvation.

Mutexes

The term mutex stands for mutual exclusion. If a task wants to access a shared resource, it should execute a `lock` method. This method only returns, if and only if no other tasks are accessing the resource. When the tasks is done with the resource it should call an `unlock` method. A thread waiting on the resource can then proceed. A mutex can be implemented as a boolean with an operations defined on it. The operation tests the boolean for a specific value, and if it matches the boolean is set to a new value. This operation must be atomic. The idea of this implementation is still in use today, and is called a *spin lock*. An example is implemented in Figure 2.2. In this example a imaginary keyword `atomic` is used to indicate a block is uninterruptible. This keyword is used to simulate an atomic hardware test-and-set operation. Spin locks are often implemented using these atomic hardware operations.

These spin locks have a major disadvantage: the busy-wait loop in the `lock` operation. This makes them processor intensive for heavily contended mutexes. To avoid this loop, operating system support for suspending tasks is needed. If a thread wants to acquire a mutex and the mutex is not available, the thread should be descheduled until the mutex is available. Such a task should be woken if the `unlock` method of the mutex is called. POSIX defines a pthread_mutex

```
boolean mutex = false;

void unlock(boolean *lock) {
  atomic {
    assert(*lock);
    *lock = false;
  }
}

void lock(boolean *lock) {
  boolean succes = false;
  while (!succes) {
    atomic {
      if (!*lock) {
        *lock = true;
        succes = true;
      }
    }
  }
}
```

Figure 2.2: Simple implementation of a mutex with a busy-wait loop. The `atomic` keyword denotes an atomic operation, not interruptible by anything else.

object supporting this behaviour. The object has a `lock` and `unlock` method. The `unlock` method must be called by the same thread as the `lock` method is called.

The Posix compliant threading library Native Posix Thread Library (NPTL) on Linux [35] is implemented by means of *futexes* [37, 34], a special kernel primitive. These futexes support descheduling if another thread has already obtained the mutex. Futexes consist of two parts: an integer in user-space and a thread queue in the kernel. The integer can be adjusted with atomic operations. If such an operation can not succeed, the thread is placed into the kernel queue. Another thread can wake the thread again, in its `unlock` method. An implementation in this manner yields considerable performance benefits. This is because the lock is mostly not contended and atomic operations are sufficient for the `lock` and `unlock` operations.

Reentrant mutexes allow a single thread to execute the `lock` operation multiple times, i.e. nest them. The thread is required to perform an equal amount of `unlock` operations. This is useful for functions operating on the same object, each beginning and ending with a lock operation. Allowing reentrancy enables these functions to call each other. This avoids the problem of the non-reentrant version where two lock operations result in a deadlock. Posix offers the options to store information in *thread local storage*. This is memory which is only available to *one* thread. Reentrant mutexes can be implemented easily by wrapping the mutex. The wrapper function counts the number of nested calls. It only calls the real mutex for the first `lock` call and the last `unlock` call, otherwise it only adjusts the reentrancy count. A real implementation based on this scheme will be slow because thread local storage is really slow in almost all implementations. Therefore support for reentrancy is mostly build directly into the mutexes.

Semaphores

Another locking mechanism are semaphores. Developed by Dijkstra [15] in 1965, these semaphores were the first real solution for concurrent programming. A formal analysis of semaphores and starvation issues is available in [31]. Semaphores are more flexible as mutexes and can also be used to implement mutexes. A semaphore is a non-negative counter with two associated functions, `P` (the imaginary Dutch word *prolaag*[1]) and `V` (the Dutch word *verhoog*). The `P` function tries to decrease the counter with one, *or* returns an error if the counter is not greater than zero. The `V` increases the counter with one. Both actions are executed atomically. Unlike mutexes, the functions `P` and `V` can be called from different threads.

The semaphore is initialised with a value. If this value is one, the semaphore is called a *binary semaphore.* Such semaphore can be used as a mutex. Also just as with mutexes, semaphores can support descheduling of threads if the semaphore can not be decreased. But unlike mutexes, semaphores can not be reentrant, because there is no notion of ownership by a specific thread. Semaphores are often used for regulating the number of tasks doing a specific job.

Readers-writers mutexes

For efficiency considerations readers-writers were introduced by Courtois et al [14] in 1971. This method distinguishes tasks that only access the resource without modifying it, called *readers*, from tasks requiring full access, called *writers.* When obtaining a lock, the programmer must indicate wether the thread only needs read access to the resource or full access. A thread that has obtained read access is called a *reader.* A *writer* is a thread that has obtained full access. This facilitates higher efficiency: *or* multiple readers and no writers *or* no readers and only one writer are allowed concurrently.

There are three types of readers-writers mutexes. The types are based on the starvation properties of threads obtaining the mutex. The first type states that if a thread has been given read access from the mutex, all other threads requesting read access should obtain it without delay. This is called the *first kind of readers-writers.* In this case writer starvation is possible: if readers are always busy, no write lock can be obtained. The *second kind of readers-writers* gives priority to writers. This type of mutex first executes all requests for obtaining a write lock, after which it executes the remaining read lock requests. But this type of mutex suffers from reader starvation. This occurs when there are always writers waiting to obtain the mutex. The last type is called the *third kind readers-writers.* This is a starvation-free version of readers-writers. Both readers and writers are guaranteed to execute.

As with plain mutexes, readers-writers mutexes can be reentrant too. But because there are two kinds of lock operations, there are different kinds of reentrancy. The most simple form is *weakly reentrant.* This only permits reentrant operations of the same kind as the first operation. If the first operation was a request for a read lock, only read locks are allowed. This can hamper modular programming. If a thread has already obtained a write lock one can not call for a method which obtains a read lock. The other form is *strongly reentrant.* This form is more interesting. It allows one to obtain a read lock after the same thread has already obtained a write lock. The inverse is not included, otherwise an obvious deadlock is possible. If two threads first request a read lock and then a write lock, both threads are deadlocked.

[1] The Dutch word for increase is *verhoog*, while the word for decrease is *verlaag.* As Dijkstra thought this was too confusing for non-Dutch speaking people, he invented the word *prolaag*, a composition of *probeer te verlagen* (try to decrease).

Summary of the properties of readers-writers mutexes:

- There is maximal *one* executing writer.
- No readers can execute when a writer is executing.
- For the first kind of readers-writers: if a read lock has been obtained, all requests for other read locks are executed.
- For the second kind of readers-writers: if a request for a write lock is pending, no new (non-reentrant) read lock requests are executed.
- For the third kind of readers-writers: The order in which the threads are given a lock is the same order as the lock were requested.

2.4 Thread synchronisation mechanisms

Another class are *thread synchronisation mechanisms*. These mechanisms regulate the scheduling of the threads. One thread can influence another thread.

Barriers

Barriers facilitate synchronisation between threads, by means of synchronisation points. When a thread reaches a barrier, it waits until all other threads also reach the barrier before continuing. These kind of barriers can be implemented in all sorts of ways. E.g. if programming a `for` loop with OpenMP, barriers are used by semantics of the language.

A special case of barriers is the joining of threads. This means that a thread can wait until another thread has finished execution. The joining of threads can easily be implemented with *semaphores*. If x threads needs to be joined, a semaphore with a count of 0 can be created. The threads each execute a `V` operation on the semaphore when the barrier is reached, which increases the count. The thread waiting on the other threads can execute the `P` operation x times. This thread blocks until all x threads have finished execution. The Posix library also supports this kind of joining. The function is called `pthread_join` and its prototype is listed in Figure 2.1. The second argument to the function is a memory location where the return code of the finished thread will be written.

Conditional Variables

Conditional variables, or *conditions*, offer a tight integration between access synchronisation and event notification. The concept was developed by C.A.R. Hoare [23] and P. Brinch Hansen [12] in 1974, in a somewhat different form called *monitors*. Such monitor is an object used by multiple threads. All methods are protected by a mutex, so at any moment just one method of the object can be executed. Monitors also provide *conditions*. These are declared in the class definition. Two methods operate on the conditions: `wait` and `signal`. Conceptually a thread executing the `wait` method for condition c waits for the condition c to be true, by executing `wait c`. Such a thread unlocks the mutex, waits until it is signalled for condition c, regains the mutex and continues execution. Another thread can signal that condition c is true by signalling on condition c. This translates to executing the following code: `signal c`.

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
                                  pthread_mutex_t *restrict mutex);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

Figure 2.3: Function operating on pthread_cond.

Depending how the signalling is handled there are two sorts of monitors: *blocking* and *non blocking*. The first was as proposed originally by Hoare. The latter was used in the Mesa programming language developed by Xerox PARC in the late 1970s. The blocking type transfer the control directly to the signalled thread. Non-blocking does not transfer control, but enqueues the signalled thread in a high priority queue for obtaining the mutex.

The Posix variant of condition variables differs from monitors that the condition variables are created explicitly, as there are no objects available in C. A condition variable has to be used in conjunction with a mutex. There are three operations available: `pthread_cond_wait`, `pthread_cond_signal` and `pthread_cond_broadcast`. The function prototypes are listed in Figure 2.3. The `signal` function works as expected. The `broadcast` function wakes all waiting threads. The `wait` function has an extra argument: the mutex on which the condition functions.

The condition variable should always be used in a loop which reevaluates the condition used to determine if the thread got signalled. Otherwise certain scheduling scenario's exists where a thread continues execution while the the condition the thread just waited for is unsatisfied. For example, two threads are signalled based on a condition. Statements executed by the first scheduled thread could make the condition unsatisfied. The second thread should not continue execution. But as the second thread is already scheduled for execution it should revalidate the condition variable to prevent wrongful continuation. An example of this is shown in Section 4.4. Also this leads inherently to a possible starvation issue, as the order in which the threads waits on a condition variable can not be used to prevent starvation. Depending on the algorithm which uses condition variables it is possible that a thread can not escape the loop described above, as every time the thread is scheduled the condition remains not satisfied and the thread will wait again for the condition. To avoid these issues one have to maintain some sort of order oneself in the algorithm using condition variables.

2.5 Example: a concurrent stack

We can demonstrate the use of a number of Posix synchronisation mechanisms with a concurrent stack with a fixed maximum size. This stack can contain maximal `STACK_SIZE` number of integers. It supports two operations: `push` and `pop`. The `push` operation adds an item to the stack. If the stack is full it blocks on the condition variable `isFull`. If the operation succeeds it signals the condition `isEmpty`. The `pop` operation performs the inverse: it pops an item from the stack. If the stack is empty it blocks on the condition variable `isEmpty`. It the operation succeed it signals the condition `isFull`. The code is listed in Figure 2.4.

```
#define STACK_SIZE 5
typedef struct int_stack IntStack;
struct int_stack {
  pthread_mutex_t m;
  pthread_cond_t isEmpty;
  pthread_cond_t isFull;
  unsigned int size;
  int item[STACK_SIZE];
};

void push(Stack *s, int value) {
  pthread_mutex_lock(&s->m);
  while (s->size == STACK_SIZE)
    pthread_cond_wait(&s->isFull, &s->m);
  s->item[s->size] = value;
  s->size++;
  pthread_cond_signal(&s->isEmpty);
  pthread_mutex_unlock(&s->m);
}

int pop(Stack *s) {
  pthread_mutex_lock(&s->m);
  while (s->size == 0)
    pthread_cond_wait(&s->isEmpty, &s->m);
  s->size--;
  int retval = s->item[s->size];
  pthread_cond_signal(&s->isFull);
  pthread_mutex_unlock(&s->m);
  return retval;
}
```

Figure 2.4: Simple implementation of a stack of integers with a Posix mutex and conditions for synchronisation.

2.6 Alternative approaches

A good overview of the problems with threads and possible solutions can be found in [26]. As task queueing systems are omitted from the overview, they are described next. Task queueing systems decouples tasks to be executed and the threads running them. There exist several implementations, most notably *Grand Central Dispatch* (GCD), created by Apple. This implementation is described next. Tasks can be dispatched to specific work queues. Tasks in the same queue are executed in order, but tasks from different queues can execute parallel to each other. GCD automatically schedules the tasks for execution. The number of concurrent executing tasks is managed by the operating system. Based on the load of the machine and the activity of other application, extra execution threads are started or stopped. The main difference with other systems such as the Executor class of Java, is that GCD can enqueue tasks if an external event is triggered, for example: GUI events, arrival of network data, timers or signals. This way polling loops can be avoided. This dispatch system can also be used for certain access synchronisation problems. E.g. mutual exclusion for an object can be implemented by enqueueing all operations on that object to *one* specific queue. All operations from that queue are executed in order. Therefore these operations can not access to the object concurrently.

Chapter 3

QT's readers-writers explained

In this chapter QT's implementation of a readers-writers lock is presented. Qt is a development framework, available for WINDOWS, MAC OS X and LINUX. In this thesis the 4.3.5 version of QT for LINUX is considered. This is due to the availability of the sources of LINUX, so the entire system can be analysed. QT implements a weakly reentrant readers-writers with writers preference, in the class QReadWriteLock. The implementation is dependent on *mutexes* and *condition variables*. Both are available in the QT framework, as respectively QMutex and QWaitCondition. We start with explaining these classes.

3.1 QT's mutex

The implementation of QMutex is highly optimized. It has two parts: a platform-independent part and platform dependent. The mutex starts platform independently. This part implements a lock with an atomic operation. If obtaining the spin lock fails, control is handed over to the platform dependent part. This part deschedules the thread. The unlock method checks if there are other threads waiting on the lock. If there are threads waiting, it signals one of those.

The implementation of QMutex uses a private object, of class QMutexPrivate, to hide the platform-dependent part. This part is accessible through an indirection `d`. The class is listed in Figure 3.1. The `self` method returns an unique identifier for thread calling the method. The `wait` deschedules the current thread until it receives a signal. The method returns a boolean, with a value of true indicating the wait succeeded without problems. The `wakeUp` method signals a waiting thread. QMutex can be initialised as recursive or non recursive mode, the boolean attribute `recursive` indicates this. `contenders` is a QAtomic object, implementing atomic operations on an integer. The integer count of this variable indicates the number of threads contending for the lock. The `count` variable indicates the number of locks the thread identified by `owner` has obtained. At last there are some platform specific variables. The boolean `wakeup` indicates a thread has to wakeup. This variable is needed to counter spurious wakeups of `pthread_mutex_wait` many vendors warn about. Also this counters the situation when a `wakeUp` is performed, before the `wait` it should wake up is called. This situation is explained clearly in [36]. At last follows a POSIX mutex called `mutex` and a POSIX condition variable called `cond`.

```
class QMutexPrivate {
public:
    QMutexPrivate(QMutex::RecursionMode mode);
    ~QMutexPrivate();

    ulong self();
    bool wait(int timeout = -1);
    void wakeUp();

    const bool recursive;
    QAtomic contenders;
    ulong owner;
    uint count;

#if defined(Q_OS_UNIX)
    volatile bool wakeup;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
#elif defined(Q_OS_WIN32)
    HANDLE event;
#endif
};
```

Figure 3.1: Attributes of the QMutex object.

```
void QMutex::lock() {
    ulong self = 0;
#ifndef QT_NO_DEBUG
    self = d->self();
#endif
    if (d->recursive) {
        self = d->self();
        if (d->owner == self) {
            ++d->count;
            Q_ASSERT_X(d->count != 0, "QMutex::lock", "Recursion counter overflow");
            return;
        }
    }

    bool isLocked = d->contenders.fetchAndAddAcquire(1) == 0;
    if (!isLocked) {
#ifndef QT_NO_DEBUG
        if (d->owner == self)
            qWarning("QMutex::lock: Deadlock detected in thread %ld", d->owner);
#endif

        // didn't get the lock, wait for it
        isLocked = d->wait();
        Q_ASSERT_X(isLocked, "QMutex::lock", "Infinite wait has timed out.");

        // don't need to wait for the lock anymore
        d->contenders.deref();
    }
    d->owner = self;
    ++d->count;
    Q_ASSERT_X(d->count != 0, "QMutex::lock", "Overflow in recursion counter");
}

void QMutex::unlock() {
    Q_ASSERT_X(d->owner == d->self(), "QMutex::unlock()",
               "A mutex must be unlocked in the same thread that locked it.");

    if (!--d->count) {
        d->owner = 0;
        if (!d->contenders.testAndSetRelease(1, 0))
            d->wakeUp();
    }
}
```

Figure 3.2: Platform independent part of the implementation of QMutex::lock, Qt version 4.3.5.

The working of the implementation is not clear. There are three observations to be made about the `lock` method of QMutex, listed in Figure 3.2. First of all the boolean `isLocked` at line 15 is indicating its inverse. This hampers the understanding of the code. Secondly, the check if it is a recursive call to the mutex, at line 8 and 18, is not atomic. This means a potential error could occur if the memory where the variable `owner` is located, is not correctly processor aligned. On some processors non aligned memory is updated in two steps. This can introduce all sorts of access synchronisation errors. Third, the code at line 28 is not clear. This code is needed because of an unneeded case distinction in the `unlock` method, as listed in Figure 3.2. The working of the code is more clear if line 28 of the `lock` method is removed and line 40 of the `unlock` method is changed to: `if (!d->contenders.deref())`. This has the same workings, as the `deref` method decreases the integer with one and returns a boolean indicating if the integer has reached zero after the subtraction. If the integer has not reached zero, more threads are waiting and one needs to be signalled.

Also, for a relatively simple mutex a lot of code is needed. Both a mutex and a condition variable are needed in the Unix case, the mutex is only needed to use the condition lock. Also for non reentrant locks a special mode exists in Linux and FreeBSD called `PTHREAD_MUTEX_ADAPTIVE_NP`. This mode tries to obtain a mutex with an atomic operation and if this fails this mode will deschedule the thread. This mode functions the same as the QMutex optimisations. But using this mode instead of the optimisations avoid the cost of a explicit condition variable. The case a thread needs to be descheduled can probably be handled more efficient internally. Also, from the code it is not clear why semaphores are not used to wake up waiting threads. As semaphores are more reliable (not suffering from spurious wake-ups), the boolean `wakeup` and much code in `wait`, listed in Figure 3.3, is not needed.

```
bool QMutexPrivate::wait(int timeout) {
    report_error(pthread_mutex_lock(&mutex), "QMutex::lock", "mutex lock");
    int errorCode = 0;
    while (!wakeup) {
        if (timeout < 0) {
            errorCode = pthread_cond_wait(&cond, &mutex);
        } else {
            struct timeval tv;
            gettimeofday(&tv, 0);

            timespec ti;
            ti.tv_nsec = (tv.tv_usec + (timeout % 1000) * 1000) * 1000;
            ti.tv_sec = tv.tv_sec + (timeout / 1000) + (ti.tv_nsec / 1000000000);
            ti.tv_nsec %= 1000000000;

            errorCode = pthread_cond_timedwait(&cond, &mutex, &ti);
        }
        if (errorCode) {
            if (errorCode == ETIMEDOUT)
                break;
            report_error(errorCode, "QMutex::lock()", "cv wait");
        }
    }
    wakeup = false;
    report_error(pthread_mutex_unlock(&mutex), "QMutex::lock", "mutex unlock");
    return errorCode == 0;
}

void QMutexPrivate::wakeUp() {
    report_error(pthread_mutex_lock(&mutex), "QMutex::unlock", "mutex lock");
    wakeup = true;
    report_error(pthread_cond_signal(&cond), "QMutex::unlock", "cv signal");
    report_error(pthread_mutex_unlock(&mutex), "QMutex::unlock", "mutex unlock");
}
```

Figure 3.3: Unix specific part of the implementation of `QMutex::wait` and `QMutex::wakeUp`, Qt version 4.3.5.

```
struct QWaitConditionPrivate {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int waiters;
    int wakeups;
};

bool QWaitCondition::wait(QMutex *mutex, unsigned long time) {
    if (! mutex)
        return false;

    if (mutex->d->recursive) {
        qWarning("QWaitCondition: cannot wait on recursive mutexes");
        return false;
    }

    report_error(pthread_mutex_lock(&d->mutex), "QWaitCondition::wait()", "mutex lock");
    ++d->waiters;
    mutex->unlock();

    int code;
    forever {
        if (time != ULONG_MAX) {
            struct timeval tv;
            gettimeofday(&tv, 0);

            timespec ti;
            ti.tv_nsec = (tv.tv_usec + (time % 1000) * 1000) * 1000;
            ti.tv_sec = tv.tv_sec + (time / 1000) + (ti.tv_nsec / 1000000000);
            ti.tv_nsec %= 1000000000;

            code = pthread_cond_timedwait(&d->cond, &d->mutex, &ti);
        } else {
            code = pthread_cond_wait(&d->cond, &d->mutex);
        }
        if (code == 0 && d->wakeups == 0) {
            // many vendors warn of spurios wakeups from
            // pthread_cond_wait(), especially after signal delivery,
            // even though POSIX doesn't allow for it... sigh
            continue;
        }
        break;
    }

    Q_ASSERT_X(d->waiters > 0, "QWaitCondition::wait", "internal error (waiters)");
    --d->waiters;
    if (code == 0) {
        Q_ASSERT_X(d->wakeups > 0, "QWaitCondition::wait", "internal error (wakeups)");
        --d->wakeups;
    }
    report_error(pthread_mutex_unlock(&d->mutex), "QWaitCondition::wait()", "mutex unlock");
    mutex->lock();

    if (code && code != ETIMEDOUT)
        report_error(code, "QWaitCondition::wait()", "cv wait");

    return (code == 0);
}

void QWaitCondition::wakeOne() {
    report_error(pthread_mutex_lock(&d->mutex), "QWaitCondition::wakeOne()", "mutex lock");
    d->wakeups = qMin(d->wakeups + 1, d->waiters);
    report_error(pthread_cond_signal(&d->cond), "QWaitCondition::wakeOne()", "cv signal");
    report_error(pthread_mutex_unlock(&d->mutex), "QWaitCondition::wakeOne()", "mutex unlock");
}

void QWaitCondition::wakeAll() {
    report_error(pthread_mutex_lock(&d->mutex), "QWaitCondition::wakeAll()", "mutex lock");
    d->wakeups = d->waiters;
    report_error(pthread_cond_broadcast(&d->cond), "QWaitCondition::wakeAll()", "cv broadcast");
    report_error(pthread_mutex_unlock(&d->mutex), "QWaitCondition::wakeAll()", "mutex unlock");
}
```

Figure 3.4: UNIX specific implementation of the QWaitCondition object, QT version 4.3.5.

3.2 QT's condition variables

The QWaitCondition class works about the same as the conditional variable part of QMutex. The implementation is listed in Figure 3.4. As with the QMutex class, it uses an indirection to hide its implementation details (attribute `d` from class QWaitCondition). It has three methods: `wait`, `wakeOne` and `wakeAll`. As expected, the `wait` function deschedules the thread until a 'wake' method is called by another thread. The `wakeAll` wakes all waiting threads, while `wakeOne` signals just one thread, both in no special order.

As one can see, the class depends on pthread_mutex, pthread_cond (appearing as attribute types) and on QMutex (passed as an argument to the method `QWaitCondition_wait`). QWaitCondition is designed in this manner to counter spurious wake-ups, the same problem as occurring in QMutex. The variable `wakeups` is needed to keep track of the number of threads allowed to wake up and is bound by the number of waiting threads, as contained in the variable `waiters`. Precisely formulated: $0 \leq$ `wakeups` $\leq$ `waiters`. The pthread_mutex used in QWaitCondition is needed because QMutex does not use a pthread_mutex in the optimised case, and such a mutex is needed if calling the `pthread_cond_wait` function.

As mentioned in the previous chapter, waiting on conditional variables should occur in a loop reevaluating the condition to prevent race conditions. This also applies for these condition variables. In code using these condition variables, there are two loops: the inner loop checking the `wakeups` variable in `QWaitCondition::wait` and the outer loop in which `wait` is called. The inner loop is used to counter spurious wakeups. But the outer loop can also counter these spurious wakeups. So using only the outer loop is already sufficient. Therefore also the variable `wakeups` is not needed.

3.3 QT's readers-writers

With the basics explained we can continue with the class QReadWriteLock. This class implements a weakly reentrant readers-writers algorithm with writers preference. The relevant code of the implementation is listed in Figure 3.5. The structure QReadWriteLockPrivate contains the attributes of the class. These are accessible via an indirection named `d`. The attributes `mutex` (of type QMutex), `readerWait` (of type QWaitCondition) and `writerWait` (of type QWaitCondition) are used to synchronize access to the other administrative attributes, of which `accessCount` keeps track of the number of locks acquired (including reentrant locks) for this lock. A negative value is used for write access and a positive value for read access. The attributes `waitingReaders` and `waitingWriters` (both int's) indicate the number of pending threads requesting read respectively write permission. If some thread owns the write lock, `currentWriter` contains a HANDLE to this thread; otherwise `currentWriter` is a null pointer.

It has three methods: `lockForRead`, `lockForWrite` and `unlock`. All three methods first lock the basic mutex QReadWriteLock is build upon. This is done via the constructor of the wrapper class QMutexLocker, e.g. line 19 in Figure 3.5. Unlocking this mutex happens implicitly in the destructor of this wrapper (when the function returns).

```
struct QReadWriteLockPrivate {
  QReadWriteLockPrivate()
  : accessCount(0),
    currentWriter(0),
    waitingReaders(0),
    waitingWriters(0)
  { }

  QMutex mutex;
  QWaitCondition readerWait,
                 writerWait;

  Qt::HANDLE currentWriter;
  int accessCount,waitingReaders,
      waitingWriters;
};

void QReadWriteLock::lockForRead() {
  QMutexLocker lock(&d->mutex);
  while (d->accessCount < 0 ||
         d->waitingWriters) {
    ++d->waitingReaders;
    d->readerWait.wait(&d->mutex);
    --d->waitingReaders;
   }
  ++d->accessCount;
  Q_ASSERT_X(d->accessCount>0,
        "...","...");
}

void QReadWriteLock::lockForWrite() {
  QMutexLocker lock(&d->mutex);
  Qt::HANDLE self =
      QThread::currentThreadId();
  while (d->accessCount != 0) {
    if (d->accessCount < 0 &&
        self == d->currentWriter) {
      break; // recursive write lock
    }
    ++d->waitingWriters;
    d->writerWait.wait(&d->mutex);
    --d->waitingWriters;
  }
  d->currentWriter = self;
  --d->accessCount;
  Q_ASSERT_X(d->accessCount<0,
        "...","...");
}

void QReadWriteLock::unlock() {
  QMutexLocker lock(&d->mutex);
  Q_ASSERT_X(d->accessCount!=0,
        "...","...");
  if ((d->accessCount > 0 &&
          --d->accessCount == 0) ||
      (d->accessCount < 0 &&
         ++d->accessCount == 0)) {
    d->currentWriter = 0;
    if (d->waitingWriters) {
      d->writerWait.wakeOne();
    } else if (d->waitingReaders) {
      d->readerWait.wakeAll();
    }
   }
}
```

Figure 3.5: The QReadWriteLock class of Qt 4.3.5.

A lockForRead operation may only continue if the lock is not currently locked for writing and there are no waiting writers. It checks these conditions and otherwise waits on the readerWait condition variable. Otherwise it can continue and increases the accessCount variable. A write lock can only be obtained when the lock is completely released (d->accessCount == 0), or the thread already has obtained a write lock (a reentrant write lock request, d->currentWriter == self). Otherwise it waits on the writerWait condition variable. The unlock method decreases the accessCount variable. If the lock count reaches zero, it wakes a thread waiting on a write lock if available. Otherwise all threads waiting on a read lock are signalled. In this way preference is given to threads waiting on a write lock. Threads are signalled by executing the wakeOne method on writerWait (in case of thread waiting on a write lock) or executing the wakeAll method on readerWait (in case of threads waiting on a read lock).

The code could be polished a bit. Some of the administrative attributes can be expressed in terms of the others. The distinction between readers and writers in the accessCount variable is unneeded, as currentWriter already contains the same information. Also the variables waitingReaders and waitingWriters are redundant, both are contained in the QWaitCondition objects: respectively readerWait and writerWait. The QReadWriteLock uses one QMutex object and two QWaitCondition objects. In total three pthread_mutex objects are used and three pthread_cond objects. If the readers-writers was implemented with POSIX primitives, only one mutex and two condition variables are needed. However, this thesis continues with the original code, except for the messages in the assertions which were, of course, more informative.

Chapter 4

Modelling the Posix basics

Spin is an explicit state model checker with support for assertions and Linear Temporal Logic (LTL), including liveness properties. Spin converts a model written in the specification language Promela to a checker written in C. By compiling and running the checker, properties can be verified; e.g. see [10, 24, 7].

In my Bachelor thesis (which also ended up in [17]) Uppaal was used for modelling the system. An advantage of Uppaal is its intuitive and easy to use graphical interface. However, the switch to Spin was made mainly because of two reasons. First, the input language Promela resembles C. Almost all language constructs used in the Qt case study of readers-writers are available in Promela. This allows for a direct and clear conversion of the code to a model. Second, compiled models generated by Spin appear to be more efficient for this case study than equivalent models specified in Uppaal. This allows one to verify larger models.

A few general notes can be made about modelling code in Promela. Promela is not a (general-purpose) programming language, and therefore it lacks some features that are found in common language like C/C++ and Java. For instance, there are no functions that return values in Promela. For simple non-recursive procedures, one can use the `inline` construct instead. Moreover, Promela does not support object oriented programming. In this thesis the attributes of objects will be represented as structs. Non-static methods are converted as an inline construct, having the object it operates on as an explicit argument called `this`. Also the function names of C++ methods needs to be converted. Because colons are not allowed in `inline` names in Promela, those characters are converted to an underscore.

A feature of Spin is the ability to embed C code directly. With a couple of special Promela statements C code can be inserted in the model and is executed atomically in the model. Spin tracks the memory used by these statements and include the memory regions in the state space. One can easily convert source code to a Promela model by wrapping all C code in the proper Promela statements. This method is not applicable to this case study: the mutexes are system calls which modify memory outside the process space. The content of these (kernel) memory regions can not be rolled back by Spin as the state space is explored. So we have to model the whole implementation in Promela.

The readers-writers algorithm we looked into uses the pthread_mutex and pthread_cond components of the Posix Thread Library. This library is part of the operating system. Creating a code based model of these components would require the treatment of OS dependent details making the whole system too complex. Instead this thesis will use abstract versions of these components. We first model pthread_mutex in Promela, followed by pthread_cond. To get more feeling with Promela and Spin, the integer stack example of Figure 2.4 is converted to Promela and checked with Spin.

As mentioned in Chapter 3, we consider the Unix version of the Qt library. If using the 2.6 version of the Linux kernel, the default behaviour for Posix components is opportunistic and prone to starvation. Other Unix platforms, like Mac OS X, implement starvation-free mutexes. Later on these starvation-free mutexes are considered, as we look into absence of starvation. However, we will begin with the opportunistic behaviour, as it applicable to all Unix platforms.

4.1 Modelling pthread_mutex

We start with modelling the basic pthread_mutex component. The two main functions of this component are pthread_mutex_lock and pthread_mutex_unlock, which both can be specified easily in Promela; see Figure 4.1. The lock itself is represented as a single boolean, named locked, initially set to false. The pthread_mutex_lock function is an atomic operation that waits until locked is false before setting it to true, even if the expression is contained in an atomic block. Waiting can be expressed in Promela just by using a boolean expression as a statement. If, during the execution of the model such a statement is encountered, the corresponding computation branch will be suspended until the expression becomes true. The pthread_mutex_unlock function resets locked to false. To check for incorrect use, an assertion is added to the code verifying that no lock is released if it has not been obtained before. By wrapping the locked variable in a typedef (named pthread_mutex_t), it is possible to use this pthread_mutex component in the same manner as in the original C++ code.

```
typedef pthread_mutex_t {
  bool locked = false
};

inline pthread_mutex_unlock(this) {
    assert(this.locked);
    this.locked = false;
}
inline pthread_mutex_lock(this) {
  atomic {
    !this.locked;
    this.locked = true;
  }
}
```

Figure 4.1: Abstract model in Promela of the non-reentrant pthread_mutex.

4.2 Modelling pthread_cond

We now model pthread_cond. This component allows a thread owning a mutex to wait until some condition is satisfied and another thread notifies this thread about it. While waiting the thread releases the mutex obtained by the thread, and if signalled reacquires the mutex. When another running thread completes a task and determines that a waiting thread can now continue, it can wake

```
typedef pthread_cond_t {
  byte waiters = 0;
  chan cont = [0] of {bit};
};

inline pthread_cond_signal(this) {
  atomic {
    if
    :: this.waiters > 0 ->
          this.waiters--;
          this.cont?_;
    :: else
    fi;
  }
}

inline pthread_cond_broadcast(this) {
  atomic {
    do
    :: this.waiters > 0 ->
          this.waiters--;
          this.cont?_;
    :: else -> break;
    od;
  }
}

inline pthread_cond_wait(this,mutex) {
  atomic {
    this.waiters++;
    pthread_mutex_unlock(mutex);
    this.cont!1;
  }
  pthread_mutex_lock(mutex);
}
```

Figure 4.2: Abstract starvation prone model of pthread_cond in Promela.

up the waiting thread by signalling the corresponding condition. Actually, two kinds of signals are available in pthread_cond: `pthread_cond_signal` for waking one thread and `pthread_cond_broadcast` for waking all threads. Our abstract version of pthread_cond uses a basic synchronisation mechanism of Promela: (synchronous) rendez-vous channels. The `pthread_cond_wait` function uses a send operation on the rendez-vous channel `cont`. The thread invoking this method will be blocked until another thread execute a receive operation. The contents of the message sent over this channel are irrelevant, only the timing of the message matters. On the receiver side this is specified by using an anonymous write-only variable named _, resulting in the Promela statement `cont?_`. On the sender side an arbitrary value of 1 is chosen. The resulting statement becomes `cont!1`. Before waiting on the channel the wait function has to release the mutex and, after continuing, to acquire the mutex again. To be able to wake all the waiting threads, the condition keeps track of the number of waiting threads in the variable `waiters`. The value of this variable equals the number of receive operations that have to be performed if all threads are signalled. For correctness atomic blocks are used to limit the interleaving of processes (otherwise the test `waiters > 0` and `waiters--` could be interrupted). Just like pthread_mutex the variables are wrapped in a new type pthread_cond_t. The Promela model is listed in Figure 4.2.

The correctness of the `pthread_cond_broadcast` function depends on the atomicity of the body of the function, otherwise too many threads can get a signal. Therefore the send and receive operations can not be exchanged. This is due to the semantics of the atomic block in combination with the send and receive operation on a rendezvous channel. From the Promela manual page on `atomic`: 'If an atomic sequence contains a rendezvous send statement, control passes from sender to receiver when the rendezvous handshake completes' [38]. Thus an `atomic` block is no longer atomic if it contains a rendezvous send statement. As the atomic block in the broadcast function contains such a send operation. The function can therefor be interrupted. This would make the function behave incorrectly. A receive operation on channel c can only be interrupted if no other process is blocking on sending a message to channel c. In our model of pthread_cond, for each receive operation a send operation is always already blocking. For each increment of the `waiting` variable a send operation tries to execute but is blocked, as the body of the `pthread_cond_wait` method is wrapped inside an atomic block. This manner of using receive operations does not suffer from the same atomically problems as send operation and can be safely used inside a `atomic` block.

4.3 Modelling the concurrent stack example

The stack of integers example of Figure 2.4 is easily converted to Promela. The result is listed in Figure 4.3. Because of the lack of pointers, pointers are converted to `struct`s, declared global or on the stack. Accessing a member m of a structure s can be done by `s.m`. The `while` loop in C is converted to a `do ... od` loop in Promela. This loop can be thought of as a `while (1)` loop in C, as the loop has to be aborted explicitly by a `break` statement. Each loop can have multiple condition blocks (a block prefix by `::`). A condition block is only executable if the first statement is executable (so a block starting with `false;` is never executable). Normally, one of the executable blocks is chosen non-deterministically for execution. In the translated code, only one of the blocks should be executable to match the semantic of the `while` statement in C. A similar construction is needed if converting `if` statements from C to Promela. By adding a second condition block with as condition the negation of the condition of the original `while` loop, only one of the two blocks is executable. This can be stated shorter by using the `else` keyword, Spin calculates the negation automatically[1].

```
typedef IntStack {
  pthread_mutex_t m;
  pthread_cond_t isEmpty;
  pthread_cond_t isFull;
  byte size = 0;
  int item[STACK_SIZE];
};

inline push(s, value) {
  pthread_mutex_lock(s.m);
  do
  :: s.size == STACK_SIZE ->
     pthread_cond_wait(s.isFull, s.m);
  :: else -> break;
  od;
  s.item[s.size] = value;
  s.size++;
  pthread_cond_signal(s.isEmpty);
  pthread_mutex_unlock(s.m);
}
```

```
inline pop(s, retval) {
  pthread_mutex_lock(s.m);
  do
  :: s.size == 0 ->
     pthread_cond_wait(s.isEmpty, s.m);
  :: else -> break;
  od;
  s.size--;
  retval = s.item[s.size];
  pthread_cond_signal(s.isFull);
  pthread_mutex_unlock(s.m);
}
```

Figure 4.3: Conversion of the concurrent stack of Figure 2.4 to Promela.

4.4 Verifying the concurrent stack example

To be able to model check this example we have to model typical usage of the stack somehow. The test setup includes multiple consumers and multiple producers. A model of such a setup is listed in Figure 4.4. The producers and consumers loop forever, doing their job: pushing and popping the integer 1 to and from the stack. Next we can continue with checking the example.

[1] As noted in [39], a few exceptions exist. The exceptions deals with message queues and nesting of `do` and `if` loops, but do not apply here.

```
IntStack s;

active[PRODUCERS] proctype producer() {
  do
  :: push(s, 1);
  od;
}
active[CONSUMERS] proctype consumer() {
  int retval;
  do
  :: pop(s, retval);
  od;
}
```

Figure 4.4: Model of usage of a stack.

There are three kinds of properties which can be checked, each by invoking SPIN differently. The absence of deadlock property is checked implicitly when running the verifier for assertion violations. Each time a non-end state is encountered and no transitions out of the state are valid an ‘invalid end state’ error is reported. The second type of properties we check are safety properties, which are valid in each state of the model (specified as LTL formulas beginning with the [ ] operator). Most of the informal correctness properties specified in Section 2.3 are of this type. The last type are liveness properties, guaranteeing that each process can make progress in some sort. SPIN has special support for liveness properties, called *progress states*, although they can also be checked with LTL properties.

For the stack example the output of a run verifying absence of deadlocks and assertion violations is listed in Figure 4.5 (the megabytes listed are actually mebibytes). It shows the number of visited states, the number of transitions taken, the memory usage, and unexecuted lines of code in the model. There are no problems listed.

```
(Spin Version 5.2.3 -- 25 November 2009)
        + Partial Order Reduction
        + Compression

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 136 byte, depth reached 115770, errors: 0
  1003870 states, stored
   388596 states, matched
  1392466 transitions (= stored+matched)
   167725 atomic steps
hash conflicts:     16618 (resolved)

Stats on memory usage (in Megabytes):
  157.008       equivalent memory usage for states (stored*(State-vector + overhead))
   46.674       actual memory usage for states (compression: 29.73%)
                state-vector as stored = 21 byte + 28 byte overhead
   64.000       memory used for hash table (-w23)
  457.764       memory used for DFS stack (-m10000000)
  568.248       total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 232 11 11 ]
unreached in proctype producer
        line 46, state 40, "-end-"
        (1 of 40 states)
unreached in proctype consumer
        line 54, state 40, "-end-"
        (1 of 40 states)

pan: elapsed time 1.92 seconds
pan: rate 522848.96 states/second
```

Figure 4.5: Output of SPIN for the stack example, with 5 producers, 5 consumers and a stack size of 5.

```
pan: assertion violated (s.size<5) (at depth 360)
pan: wrote stack.spin.trail
pan: reducing search depth to 359
        ...
pan: wrote stack.spin.trail
pan: reducing search depth to 69

(Spin Version 5.1.7 -- 23 December 2008)
        + Partial Order Reduction
        + Compression

Full statespace search for:
        never claim                 - (none specified)
        assertion violations        +
        cycle checks               - (disabled by -DSAFETY)
        invalid end states        +

State-vector 80 byte, depth reached 365, errors: 10
     1080 states, stored
      504 states, matched
     1584 transitions (= stored+matched)
      460 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.119       equivalent memory usage for states (stored*(State-vector + overhead))
  458.335       actual memory usage for states (unsuccessful compression: 383619.54%)
                state-vector as stored = 444963 byte + 36 byte overhead
   64.000       memory used for hash table (-w23)
    0.003       memory used for DFS stack (-m69)
  522.154       total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 66 14 23 ]
unreached in proctype producer
        line 57, state 43, "-end-"
        (1 of 43 states)
unreached in proctype consumer
        line 66, state 44, "-end-"
        (1 of 44 states)

pan: elapsed time 0.01 seconds
```

Figure 4.6: Output of SPIN for the stack example, with an error introduced. Executed with 2 producers, 1 consumer and a stack size of 5.

To show the output when an error occurs, an error is introduced in the example. The loop in the `push` function is changed to a conditional statement. Also an extra assertion is added just after this conditional statement: an assertion checking if the size is below the maximum size of the stack. In PROMELA one can state this as `assert(s.size < STACK_SIZE)`. Running the verifier again yields the output listed in Figure 4.6 (the megabytes listed are actually mebibytes[2]). At the beginning SPIN indicates there is an assertion violation and a trail was written to `stack.spin.trail`. In this file, SPIN records how the violation occurred. SPIN also has the ability to replay these steps in debug mode. The output in this mode is listed in Figure 4.7. First SPIN lists all running processes. Following is the order in which the statements were executed. It finishes with a final system state of all global variables and the line numbers the running processes are at. After close inspection it is clear what caused the assertion violation. Between a consumer signalling a producer and the producer enqueueing a new integer, another producer has enqueued another integer. To avoid this situation, a woken thread has to reevaluate the condition causing the wait. A loop is necessary[3].

[2] A mebibyte is $2^{20} = 1,048,576$ bytes, and is established by the *International Electrotechnical Commission* (IEC).

[3] If starvation-free mutexes and condition are used, a loop is unnecessary. By verifying this example with such versions (introduced later in this thesis) one can see this for oneself.

```
Starting producer with pid 0
Starting producer with pid 1
Starting consumer with pid 2
  1:  proc  1 (producer) line  17 "pthread_mutex.basic.abs" (state 1)
      [(!(s.m.locked))]        <merge 0 now @2>
  1:  proc  1 (producer) line  18 "pthread_mutex.basic.abs" (state 2)
      [s.m.locked = 1]
       ......
 69:  proc  0 (producer) line  24 "pthread_mutex.basic.abs" (state 35)
      [assert(s.m.locked)]        <merge 40 now @36>
 69:  proc  0 (producer) line  25 "pthread_mutex.basic.abs" (state 36)
       [s.m.locked = 0]        <merge 40 now @40>
 70:  proc  1 (producer) line  17 "pthread_mutex.basic.abs" (state 13)
      [(!(s.m.locked))]        <merge 0 now @14>
 70:  proc  1 (producer) line  18 "pthread_mutex.basic.abs" (state 14)
      [s.m.locked = 1]        <merge 23 now @18>
spin: line  24 "stack.spin", Error: assertion violated
spin: text of failed assertion: assert((s.size<5))
 71:  proc  1 (producer) line  25 "stack.spin" (state 23)
      [assert((s.size<5))]
spin: trail ends after 71 steps
#processes: 3
    s.m.locked = 1
    s.isEmpty.waiters = 0
    s.isFull.waiters = 0
    s.size = 5
    s.item[0] = 1
    s.item[1] = 1
    s.item[2] = 1
    s.item[3] = 1
    s.item[4] = 1
 71:       proc  2 (consumer) line  62 "stack.spin" (state 41)
 71:       proc  1 (producer) line  26 "stack.spin" (state 24)
 71:       proc  0 (producer) line  54 "stack.spin" (state 40)
3 processes created
```

Figure 4.7: Trace of Spin for the stack example with a bug. Executed with 2 producers, 1 consumer and a stack size of 5.

Unless otherwise stated, all experiments in this thesis are executed on a Sun Fire X4440, with 16-cores and 128 GiB memory. Also, in Spin the maximal stack depth to be searched must be specified, since a stack for the depth-first search is allocated in advance. In this thesis, the unused part of the stack is subtracted from the actual memory usage to obtain the memory usage in which the example can be verified. A 64-bits verifier was used. If less as 4 GiB is needed for verifying a certain case, it is possible to use even less memory by compiling the verifier as 32-bits executable, but for a more fair comparison the results from the 64-bits version are used unless otherwise noted.

Memory usage in mebibytes[4] if checking for assertion violations and deadlocks:

Memory Usage (in MiB)		Stack Size			
		1	4	7	10
1	1	4.39	4.40	4.41	4.43
	4	6.04	11.19	18.36	27.09
	7	748.75	4471.74	7005.64	10012.96
4	1	4.45	4.66	4.70	4.92
	4	129.35	96.78	143.39	178.27
	7	39651.62	43053.70	40621.57	50061.57
7	1	12.23	19.68	16.68	18.00
	4	5483.77	6171.12	2869.96	3471.56
	7	n/a	n/a	n/a	n/a
Producers	Consumers				

[4] A mebibyte is $2^{20} = 1,048,576$ bytes, and is established by the *International Electrotechnical Commission* (IEC).

Reached depth if checking for assertion violations and deadlocks:

Stack Depth (in steps)		Stack Size			
		1	4	7	10
1	1	67	208	486	790
	4	1997	3574	6557	9403
	7	53518	96584	162906	228975
4	1	1222	1711	2385	3109
	4	46495	21557	28806	35611
	7	1344364	481112	585697	687034
7	1	34738	35279	33737	37098
	4	1327782	518108	308797	377983
	7	n/a	n/a	n/a	n/a
Producers	Consumers				

Runtime in hours, minutes and seconds if checking for assertion violations and deadlocks[5]:

Run Time (in HH:MM:SS)		Stack Size			
		1	4	7	10
1	1	0:00:05	0:00:03	0:00:03	0:00:04
	4	0:00:03	0:00:03	0:00:04	0:00:04
	7	0:00:43	0:03:31	0:06:02	0:09:07
4	1	0:00:03	0:00:03	0:00:03	0:00:03
	4	0:00:10	0:00:07	0:00:10	0:00:11
	7	0:50:07	0:48:40	0:41:49	0:50:45
7	1	0:00:04	0:00:03	0:00:04	0:00:04
	4	0:05:59	0:06:15	0:03:01	0:03:18
	7	n/a	n/a	n/a	n/a
Producers	Consumers				

In the next chapter we continue with modelling and verifying readers-writers.

[5] As other cores on the machine were used by other programs, the runtime measurement is not entirely accurate across the different runs.

Chapter 5

Verifying Qt's readers-writers

Now that we have abstract versions of the relevant Posix components, we can model the readers-writers available in Qt. The QReadWriteLock class implements this algorithm. As explained in Chapter 3, this class depends on the QMutex and QWaitCondition classes. Before we can model the QReadWriteLock class we must first model those dependancies.

5.1 Modelling Qt's basics

The implementation of the QMutex class appears to be rather complex to model, due to optimisations that have been performed. As a consequence, the code base is large. Modelling this part faithfully is outside the scope of this thesis. Instead we will use pthread_mutex to provide the locking mechanism, because it has the same functional behaviour as the QMutex class is supposed to have. Hence QMutex is a wrapper around around pthread_mutex. The Promela model is listed in Figure 5.1.

```
typedef QMutex {
   pthread_mutex_t mutex;
};

inline QMutex_lock(this) {
   pthread_mutex_lock(this.mutex);
}
inline QMutex_unlock(this) {
   pthread_mutex_unlock(this.mutex);
}
```

Figure 5.1: QMutex class.

The implementation of QWaitCondition, on the other hand, is much shorter, and can therefore be converted to Promela straightforwardly. Again, as with the stack example, the attributes of this class are wrapped in a Promela `typedef`. Each of the statements of the original implementation, listed in Figure 3.4, can be easily translated in Promela statements. The translation is done in the same manner as with the stack example in Section 4.3. `while` loops are translated to `do ... od` loops, conditional `if` statement are translated to `if` statements with two explicit branches. The result is listed in Figure 5.2.

An abstract version of QWaitCondition can also be constructed. Looking at the documentation [43], one can see the class has the same specification as pthread_cond. To construct an abstract model of QWaitCondition one can just change

```
typedef QWaitCondition {
   pthread_mutex_t mutex;
   pthread_cond_t cond;
   int waiters = 0;
   int wakeups = 0;
};

inline QWaitCondition_wakeOne(this) {
   pthread_mutex_lock(this.mutex);
   this.wakeups = min(this.wakeups + 1,
                      this.waiters);
   pthread_cond_signal(this.cond);
   pthread_mutex_unlock(this.mutex);
}

inline QWaitCondition_wakeAll(this) {
   pthread_mutex_lock(this.mutex);
   this.wakeups = this.waiters;
   pthread_cond_broadcast(this.cond);
   pthread_mutex_unlock(this.mutex);
}
```

```
inline QWaitCondition_wait(this, m) {
   pthread_mutex_lock(this.mutex);
   this.waiters++;
   QMutexUnlock(m);
   do
   :: this.wakeups == 0 ->
      pthread_cond_wait(this.cond,
                        this.mutex);
   :: else > 0 ->
      break;
   od;
   this.waiters--;
   this.wakeups--;
   pthread_mutex_unlock(this.mutex);
   QMutexLock(m);
}
```

Figure 5.2: Concrete model in PROMELA of QWaitCondition

the name of the typedef and of the functions of the pthread_cond component. The only real difference is that the pthread_mutex lock and unlock method calls must be converted to the lock and unlock method calls of the QMutex. The abstract model of QWaitCondition is trivial, constructing the PROMELA model is left to the reader.

5.2 Modelling QT's readers-writers lock

Now we have modelled all the components on which the QReadWriteLock class depends, we can convert the QReadWriteLock itself to PROMELA. All class attributes can be expressed directly (the type Qt::HANDLE is converted to the PROMELA type pid, both identifying a specific thread). In Figure 5.3 the variables of the class and the code of lockForRead are listed, on the left the original C++ code, and on the right the conversion in PROMELA. Methods are converted to inline definitions. The QMutexLocker is a convenience wrapper around a mutex, obtaining the mutex when the object is constructed and releasing the mutex implicitly (via its destructor) when the object is deallocated. When used as a local (stack) object, QMutexLocker obtains the lock during its initialisation and releases the lock when this local object gets out of scope. This implicit destructor invocation is converted to an explicit call of QMutexUnlock. The rest of the PROMELA code should be self-explanatory. The full code is listed in Figure 5.4.

5.3 Modelling usage of the readers-writers lock

In order to verify the model we will simulate all possible usages of the QReadWriteLock. For this reason we will define a number of threads, each (sequentially) executing a finite number of read and/or write locks, and matching unlocks, in a proper sequence (i.e. no unlocks if the lock is not obtained first by the thread, and no write lock requests if the thread already has obtained a read lock, or visa versa). Eventually each thread relinquishes all locks, so other threads are allowed to proceed. The variable maxLocks indicates how many locks a thread may request before it relinquishes all locks. We model these threads by PROMELA processes as shown in Figure 5.5. Here, THREADS indicates the num-

```
struct QReadWriteLockPrivate {
   QMutex mutex;
   QWaitCondition readerWait,
                  writerWait;
   Qt::HANDLE currentWriter;
   int accessCount,
     waitingReaders,
     waitingWriters;
};

void QReadWriteLock::lockForRead() {
  QMutexLocker lock(&d->mutex);
  while (d->accessCount < 0 ||
         d->waitingWriters) {
      ++d->waitingReaders;
      d->readerWait.wait(&d->mutex);
      --d->waitingReaders;
  }
  ++d->accessCount;
  Q_ASSERT_X(d->accessCount > 0,
                     "...", "...");
}
```

```
typedef QReadWriteLock {
   QMutex mutex;
   QWaitCondition readerWait;
   QWaitCondition writerWait;
   pid currentWriter = NT;
   int accessCount = 0;
   int waitingReaders = 0;
   int waitingWriters = 0;
};

inline QReadWriteLock_lockForRead(this) {
  QMutex_lock(this.mutex);
  do
  :: this.accessCount < 0 ||
        this.waitingWriters > 0 ->
      this.waitingReaders++;
      QWaitCondition_wait(this.readerWait,
                          this.mutex);
      this.waitingReaders--;
  :: else -> break;
  od;
  this.accessCount = this.accessCount + 1;
  assert(this.accessCount > 0);
  QMutex_unlock(this.mutex);
}
```

Figure 5.3: Part of QReadWriteLock (Qt 4.3 version) in C++ (left) and Promela (right).

```
typedef QReadWriteLock {
  QMutex mutex;
  QWaitCondition readerWait;
  QWaitCondition writerWait;
  int accessCount = 0;
  pid currentWriter = NT;
  int waitingReaders = 0;
  int waitingWriters = 0;
}

inline QReadWriteLock_lockForRead(this) {
  QMutex_lock(this.mutex);
  do
  :: this.accessCount < 0 ||
        this.waitingWriters > 0 ->
     this.waitingReaders++;
     QWaitCondition_wait(this.readerWait,
                         this.mutex);
     this.waitingReaders--;
  :: !(this.accessCount < 0 ||
        this.waitingWriters > 0) ->
     break;
  od;
  this.accessCount = this.accessCount + 1;
  assert(this.accessCount > 0);
  QMutex_unlock(this.mutex);
}

inline QReadWriteLock_lockForWrite(this) {
  QMutex_lock(this.mutex);
  pid self = _pid;
  do
  :: this.accessCount != 0 ->
     if
     :: this.accessCount < 0 &&
           self == this.currentWriter ->
        break;
     :: else
     fi;
     this.waitingWriters++;
     QWaitCondition_wait(this.writerWait, this.mutex);
     this.waitingWriters--;
  :: !(this.accessCount != 0) ->
     break;
  od;
  this.currentWriter = self;
  this.accessCount = this.accessCount - 1;
  assert(this.accessCount < 0);
  QMutex_unlock(this.mutex);
}

inline QReadWriteLock_unlock(this) {
  QMutex_lock(this.mutex);
  assert(this.accessCount != 0);
  if
  :: this.accessCount > 0 ->
     this.accessCount = this.accessCount - 1;
  :: this.accessCount < 0 ->
     this.accessCount = this.accessCount + 1;
  fi;
  if
  :: this.accessCount == 0 ->
     this.currentWriter = NT;
     if
     :: this.waitingWriters > 0 ->
        QWaitCondition_wakeOne(this.writerWait);
     :: else ->
        if
        :: this.waitingReaders > 0 ->
           QWaitCondition_wakeAll(this.readerWait);
        :: else
        fi;
     fi;
  :: else
  fi;
  QMutex_unlock(this.mutex);
}
```

Figure 5.4: The Promela model of the 4.3 version of the QReadWriteLock class.

```
active[THREADS] proctype user() {
   byte maxLocks;
   byte nest = 0;
   do
   :: maxLocks = MAXLOCKS;
      if
      :: do
         :: maxLocks > 0 -> nest++;
                            QReadWriteLock_lockForRead(rwlock);
         :: nest > 0 -> nest--;
                        QReadWriteLock_unlock(rwlock);
         :: maxLocks != MAXLOCKS && nest == 0 -> break;
         od;
      :: do
         :: maxLocks > 0 -> nest++;
                            QReadWriteLock_lockForWrite(rwlock);
         :: nest > 0 -> nest--;
                        QReadWriteLock_unlock(rwlock);
         :: maxLocks != MAXLOCKS && nest == 0 -> break;
         od;
      fi;
   od;
}
```

Figure 5.5: PROMELA process of QReadWriteLock usage.

ber of threads the model is checked with. Note that the do statement chooses one of the options non-deterministically. The readNest variable is used to exclude the case in which a (reentrant) write lock is performed after a read lock is already obtained. Both readNest and writeNest are used to control unlocking.

5.4 Verifying assertions and absence of deadlock

As stated before, deadlock detection is done implicitly when checking for assertions. Each state not marked as an end state and with no outgoing transitions is reported. Also all assertions in the model are checked. Besides the assertions that were present in the original code, there is one assertion in lockForWrite that has been added, to verify that no thread gets write access when readers are busy. Also debug output was added, so in case of an error, the trail would be easier to analyse.

Running our model resulted immediately in a deadlock. This occurs if checking the model with two threads and a reentrancy of two. The counter-example can also be found if the abstract model of condition variables is used. For clarity reasons the SPIN output using this model is listed in Figure 5.6. The debug output of the shortest trail is printed. For readability, the debug output is filtered to only include our own debug statements. Appended to the output are the values of all variables in the last state, and ends with a message in which state the processes are. The situation reported by SPIN occurs when a thread already having a read lock requests another one, while another thread is waiting for a write lock. The deadlock is clear: the first thread is never going to proceed with the reentrant read lock request because there is a writer waiting. The second thread is never going to proceed because the first thread can never release the lock. A change to the algorithm is needed to avoid this deadlock.

As errors are returned by the model checker, no useful information about the model checking can be summarized.

```
pan: invalid end state (at depth 188)
pan: wrote qreadwritelock43.usage.trail
...
pan: reducing search depth to 32
...
   0: enter lockForRead
   0: leave lockForRead
      1: enter lockForWrite
      1: waiting
   0: enter lockForRead
   0: waiting
spin: trail ends after 34 steps
#processes: 2
   rwlock.mutex.m.lockedBy = 255
   rwlock.mutex.m.count = 0
   rwlock.readerWait.waiters = 1
   rwlock.readerWait.wakeups = 0
   rwlock.readerWait.waiting = 1
   rwlock.writerWait.waiters = 1
   rwlock.writerWait.wakeups = 0
   rwlock.writerWait.waiting = 1
   rwlock.accessCount = 1
   rwlock.currentWriter = 255
   rwlock.waitingReaders = 1
   rwlock.waitingWriters = 1
   readers = 1
   writers = 0
 34:    proc  0 (user) line  19 "qwaitcondition.abs" (state 29)
 34:    proc  1 (user) line  19 "qwaitcondition.abs" (state 187)
```

Figure 5.6: Output of SPIN when checking for a deadlock.

Chapter 6

Verifying a deadlock-free readers-writers

As noted in the previous chapter, the implementation of the readers-writers lock in Qt 4.3 contains a deadlock. We need to adjust the model. A change to the algorithm is described, along with new models. This chapter then continues with checking the adjusted algorithm for starvation issues. Spin has special support for this kind of properties, called *progress states*. These are explained and used in the verification.

6.1 Adjusting the model of Qt's readers-writers

The solution to the deadlock stated in the previous chapter is to let a reentrant lock always proceed [16, 17]. To check if a lock request is a reentrant operation, for each thread the number of calls to the specific lock should be remembered. If this number is positive the lock operation should always succeed. In the original C++ code, an extra variable `count` of type QHash⟨Qt::HANDLE, int⟩ is introduced, mapping thread identifiers to numbers. In our translated model we represented this hash table by an integer array `count` in which `count[pid]` is the number of reentrant locks of thread `pid`. In Promela the array in declared with the statement `int count[THREADS]`.

Furthermore, we take this opportunity to change the strange use of the `accessCount` variable: the sign of the value of `accessCount` indicates whether active locks are read locks or write locks. This distinction between readers and writers appears to be superfluous. In fact, leaving out this distinction provides that our implementation is strongly reentrant. Moreover, we changed the name of the variable into `threadCount` to indicate it actually contains the number of different threads that are currently holding the lock. The new model of QReadWriteLock is listed in Figure 6.1.

We reported the deadlock to Trolltech. Recently, Trolltech released a new version of the thread library (version 4.4) in which the deadlock was repaired. However, the new version of the Qt library is still only weakly reentrant, not admitting threads that have write access to do a read lock. This limitation unnecessarily hampers modular programming.

```
typedef QReadWriteLock {
  QMutex mutex;
  QWaitCondition readerWait;
  QWaitCondition writerWait;

  int threadCount = 0;
  int waitingReaders = 0;
  int waitingWriters = 0;

  pid currentWriter = NT;
  int count[THREADS] = 0;
}

inline QReadWriteLock_lockForRead(this) {
  QMutex_lock(this.mutex);
  // check if this is a reentrant lock
  if
  :: this.count[_pid] == 0 ->
     do
     :: (this.currentWriter != NT ||
         this.waitingWriters > 0) ->
        this.waitingReaders++;
        QWaitCondition_wait
            (this.readerWait,this.mutex);
        this.waitingReaders--;
     :: else -> break;
     od;
     this.threadCount++;
     assert(this.waitingWriters == 0);
  :: else
  fi;
  this.count[_pid]++;
  ... update model variables ...
  QMutex_unlock(this.mutex);
}

inline QReadWriteLock_lockForWrite(this) {
  QMutex_lock(this.mutex);
  // check if this is a reentrant lock
  if
  :: this.currentWriter != _pid ->
     do
     :: this.threadCount != 0 ->
        this.waitingWriters++;
        QWaitCondition_wait
            (this.writerWait,this.mutex);
        this.waitingWriters--;
     :: else -> break;
     od;
     this.currentWriter = _pid;
     this.threadCount++;
  :: else
  fi;
  assert(this.threadCount == 1 &&
         this.currentWriter == _pid);
  this.count[_pid]++;
   ... update model variables ...
  QMutex_unlock(this.mutex);
}

inline QReadWriteLock_unlock(this) {
  QMutex_lock(this.mutex);
  this.count[_pid]--;
  // is it the last unlock by this thread?
  if
  :: this.count[_pid] == 0 ->
     this.threadCount--;
     // is it the last unlock of the lock?
     if
     :: this.threadCount == 0 ->
        this.currentWriter = NT;
        if
        // if available wake one writer,
        :: this.waitingWriters > 0 ->
           QWaitCondition_wakeOne
                        (this.writerWait);
        // otherwise wake all readers
        :: else ->
           if
           :: this.waitingReaders > 0 ->
              QWaitCondition_wakeAll
                        (this.readerWait);
           :: else
           fi;
        fi;
     :: else
     fi;
  :: else
  fi;
  ... update model variables ...
  QMutex_unlock(this.mutex);
}
```

Figure 6.1: Updated PROMELA model of readers-writers algorithm.

```
active[THREADS] proctype user() {
   byte readNest = 0;
   byte writeNest = 0;
   byte maxLocks; // number of lock operation remaining
   do
   :: maxLocks = MAXLOCKS;
      do
      :: maxLocks > 0 ->
         maxLocks--;
         if
         :: readNest == 0 ->
            writeNest++;
            QReadWriteLock_lockForWrite(rwlock);
         :: readNest++;
            QReadWriteLock_lockForRead(rwlock);
         fi;
      :: writeNest + readNest > 0 ->
         QReadWriteLock_unlock(rwlock);
      :: MAXLOCKS != maxLocks && writeNest + readNest == 0 ->
         break;
      od;
   od;
}
```

Figure 6.2: PROMELA process of QReadWriteLock usage.

6.2 Verifying assertions and absence of deadlock

Before we can verify we have to adjust the usage model, as the algorithm is now *strongly* reentrant. So obtaining a read lock while already holding a write lock is possible. To fully verify the new algorithm we have to use the new algorithm in all possible and valid ways. The new usage model is listed in Figure 6.2.

After the adjustments to the model SPIN reports no invalid end states and thereby no deadlocks for 3 threads and a maximum of 7 lock operations. Also no assertion violations where found. An overview of the results is given below. More extended validation was not possible as no machine with more as 128 gibibyte of memory was available.

Memory usage in mebibytes if checking for assertion violations and deadlocks:

Max Locks	1	3	5	7
2 Threads	4.42	5.82	11.37	25.54
3 Threads	6.93	153.46	1405.38	5986.89
4 Threads	55.30	9974.17	n/a	n/a

Reached depth if checking for assertion violations and deadlocks:

Max Locks	1	3	5	7
2 Threads	557	5585	20201	43927
3 Threads	8636	321289	1957287	6921311
4 Threads	118050	13679353	n/a	n/a

Runtime in hours, minutes and seconds if checking for assertion violations and deadlocks[1]:

Max Locks	1	3	5	7
2 Threads	0:00:17	0:00:17	0:00:17	0:00:18
3 Threads	0:00:17	0:00:26	0:01:31	0:05:55
4 Threads	0:00:21	0:13:18	n/a	n/a

6.3 Verifying absence of starvation

In Section 2.3 we stated that the design decision to give preference to writers results in a possible reader starvation. Therefore it only makes sense to check for absence of writer starvation. In SPIN one can verify starvation properties by using *progress states*. A looping process obtaining and releasing write locks, but no read locks, is added and labelled with a *progress label*. When checking the model, it is verified that all execution cycles (i.e. an execution path on which the same state occurs twice) contain this progress label. The special process used in the verification is listed in Figure 6.3.

If checking the model for starvation, a trivial starvation issue in pthread_mutex is revealed. The root of the problem is the `lock` function. This method stalls if

[1] As other cores on the machine were used by other programs, the runtime measurement is not entirely accurate across the different runs.

```
active[1] proctype userWriter() {
   byte readNest = 0;
   byte writeNest = 0;
progress:
   QReadWriteLock_lockForWrite(rwlock);
   QReadWriteLock_unlock(rwlock);
   goto progress;
}
```

Figure 6.3: Extra process which checks absence of starvation of write locks.

the mutex is locked, otherwise it continues. But the duration of the stall is unspecified, although eventually the process will try again to lock the mutex. This is the same opportunistic scheduling behaviour of the standard mutexes found on some operating systems, and therefore this behaviour is also used by default by the readers-writers algorithm on those systems. This mutex is not starvation free, and moreover the readers-writers algorithm is also prone to starvation. We want to fully verify the algorithm under all conditions, so we want to verify the algorithm is free of starvation for systems with starvation-free mutexes (like FreeBSD and Mac OS X). If we want to verify Qt's readers-writers algorithm is starvation free for those systems we have to avoid the mentioned issue in the model of pthread_mutex. The necessary adjustments to the model of pthread_mutex are discussed in the next section.

6.4 Making the primitives starvation free

In order to eliminate the starvation issue mentioned in the previous section, pthread_mutex must be converted to a starvation-free version. The order in which threads call the `lock` method is remembered. Threads are only allowed to claim the mutex in the same order (FIFO), as opposed with the old opportunistic behaviour (see Section 2.3). Also pthread_cond must be converted, as a signalled thread reclaiming the mutex must get priority over threads requesting the mutex for the first time. Otherwise a trivial starvation issue can be found if checking the algorithm with the old model of pthread_cond[2].

To this end the new version of both modules combined implements a *non-blocking monitor* (see Section 2.4). To each mutex two queues of processes are added. The queues are modelled using buffered channels in Promela. These channels can contain a maximum amount of `THREADS` messages of the type `pid`. The first queue is to remember the order of the threads in which they call the `lock` function. The second contains the order in which signalled threads should continue. The `unlock` method transfers the lock to another thread if there are threads waiting. Threads in the second queue are given priority over threads in the first queue. A condition variable consists of a single queue. In this queue, the order in which the threads executed the `wait` method is stored. When a condition variable gets signalled a thread is transferred from the queue of the condition variable to the second queue of the corresponding mutex. Sleeping and waking specific threads is performed by waiting on a specific rendevouz commu-

[2] If a signalled thread is added to the normal queue for waiting on the mutex instead, starvation issues are still present. There exists a possibility that just before a thread is signalled another thread tries to claim the lock. After the mutex is unlocked, this first threads gets executed. This first thread then can obtain the resource (in this thesis a read or write lock), starving the woken thread. If this occurs repeatedly the signalled thread is starved.

nication channel. The specifics are in the file `schedular.abs`, see Section A.3. Two methods are available: `processSleep` to let the current thread sleep, and `processWake` to wake a specific thread. The models of both modules are listed in Figures 6.4 and 6.5. Now we can continue with checking for starvation issues in the algorithm itself.

```
typedef pthread_mutex_t {
  byte locked = false;
  chan queue = [THREADS] of {pid};
  chan requeue = [THREADS] of {pid};
};

inline pthread_mutex_lock(this) {
  atomic {
    if
    :: this.locked ->
       this.queue!_pid;
       processSleep();
    :: else ->
       this.locked = true;
    fi;
    assert(this.locked);
  }
}

inline pthread_mutex_unlock(this) {
  atomic {
    assert(this.locked);
    if
    :: nempty(this.requeue) ->
       this.requeue?p1;
       processWake(p1);
    :: empty(this.requeue) ->
       if
       :: nempty(this.queue) ->
          this.queue?p1;
          processWake(p1);
       :: empty(this.queue) ->
          this.locked = false;
       fi;
    fi;
  }
}

inline pthread_mutex_requeue_one(this, process) {
  atomic {
    assert(this.locked);
    this.requeue!process;
    assert(nempty(this.requeue));
  }
}

inline pthread_mutex_requeue_queue(this, aqueue) {
  atomic {
    assert(this.locked);
    do
    :: nempty(aqueue) ->
       aqueue?p1;
       this.requeue!p1;
    :: empty(aqueue) -> break;
    od;
  }
}
```

Figure 6.4: Starvation-free PROMELA model of pthread_mutex.

```
typedef pthread_cond_t {
  chan queue = [THREADS] of {pid};
};

inline pthread_cond_wait(this, m) {
  atomic {
    this.queue!_pid;
    pthread_mutex_unlock(m);
    processSleep();
    assert(m.locked);
  }
}





inline pthread_cond_signal(this, m) {
  atomic {
    if
    :: nempty(this.queue) ->
       this.queue?p1;
       pthread_mutex_requeue_one(m, p1);
    :: empty(this.queue)
    fi;
  }
}

inline pthread_cond_broadcast(this, m) {
  atomic {
    pthread_mutex_requeue_queue(m, this.queue);
    assert(empty(this.queue));
  }
}
```

Figure 6.5: Starvation-free PROMELA model of pthread_cond.

6.5 Verifying absence of starvation, again

Despite using the starvation-free Posix basics, the model still contains the possibility of writer starvation. This appeared when the model is checked for absence of progress, and Spin found an execution cycle with no progress states. The output of Spin is listed in Figure 6.7. This is the shortest counterexample found by Spin. The output is not obvious because both QMutex and QWaitCondition uses a pthread_mutex. The debug output in the model was changed slightly to be more informative, displaying also the current value of `count[_pid]`. A graphic representation of this cycle is shown in Figure 6.6.

The problem is caused by the `wait` method of QWaitCondition; see Figure 5.2. When a thread t calls `QWaitCondition_wait` it will suspend (by calling `pthread_cond_wait`) until some other thread s signals that t can continue its execution. However, at that time t has released the mutex `this.mutex`. Another thread can now lock this mutex (as occurring by calling `lockForWrite`) just before t does, effectively stealing the turn of t. When t obtains the mutex, it can not obtain a write lock, because s has already a write lock.

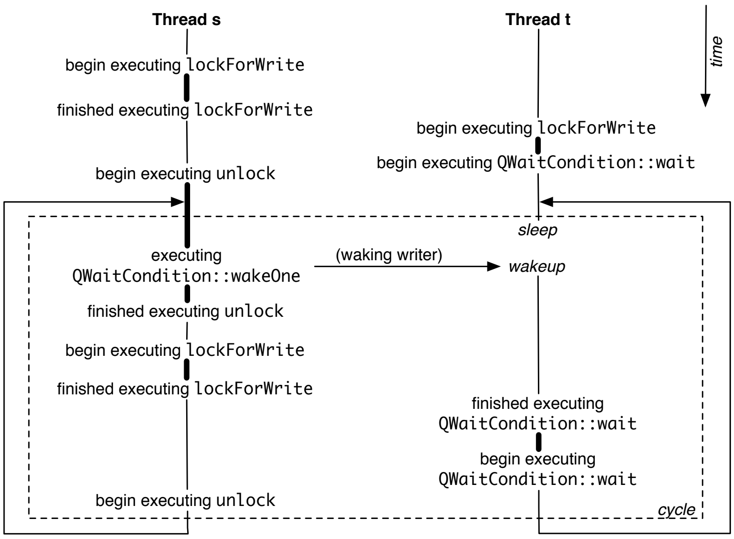
Thread s
Thread t
time
begin executing lockForWrite
finished executing lockForWrite
begin executing lockForWrite
begin executing QWaitCondition::wait
begin executing unlock
sleep
executing QWaitCondition::wakeOne
(waking writer)
wakeup
finished executing unlock
begin executing lockForWrite
finished executing lockForWrite
finished executing QWaitCondition::wait
begin executing QWaitCondition::wait
begin executing unlock
cycle

Figure 6.6: Graphical representation of the counterexample indicating a starvation problem. The thick black line indicates that the mutex `this,mutex` is locked.

```
Starting userWriter with pid 0
Starting user with pid 1
spin: couldn't find claim (ignored)
    1: pthread_mutex_lock done
    1: want writelock (this thread nested 0 deep)
    1: has writelock (this thread nested 1 deep)
    1: pthread_mutex_unlock done
  0: pthread_mutex_lock done
  0: want writelock (this thread nested 0 deep)
  0: wait for writelock (this thread nested 0 deep)
  0: begin qwait::wait
  0: pthread_mutex_lock done
  0: pthread_mutex_unlock done
    1: pthread_mutex_lock done
    1: going to release lock (this thread nested 1 deep)
    1: waking one writer
<<<<<START OF CYCLE>>>>>
  0: pthread_mutex_unlock done
    1: pthread_mutex_lock done
  0: sleeping
  0: waked, continuing
  0: pthread_mutex_lock waiting
  0: sleeping
    1: waked 0
    1: pthread_mutex_unlock done
    1: pthread_mutex_unlock waking 0
  0: waked, continuing
  0: pthread_mutex_lock continuing
  0: pthread_mutex_lock done
    1: waked 0
    1: released lock (this thread nested 0 deep)
    1: pthread_mutex_unlock done
    1: pthread_mutex_lock done
    1: want writelock (this thread nested 0 deep)
    1: has writelock (this thread nested 1 deep)
    1: pthread_mutex_unlock done
    1: pthread_mutex_lock done
    1: going to release lock (this thread nested 1 deep)
    1: waking one writer
  0: pthread_mutex_unlock done
    1: pthread_mutex_lock done
    1: pthread_mutex_unlock done
    1: released lock (this thread nested 0 deep)
    1: pthread_mutex_unlock done
    1: pthread_mutex_lock done
    1: want writelock (this thread nested 0 deep)
    1: has writelock (this thread nested 1 deep)
    1: pthread_mutex_unlock done
  0: pthread_mutex_lock done
  0: done qwait::wait
  0: continue for writelock (this thread nested 0 deep)
  0: wait for writelock (this thread nested 0 deep)
  0: begin qwait::wait
  0: pthread_mutex_lock done
  0: pthread_mutex_unlock done
    1: pthread_mutex_lock done
    1: going to release lock (this thread nested 1 deep)
    1: waking one writer
spin: trail ends after 318 steps
```

Figure 6.7: Output of SPIN checking the adjusted QReadWriteLock class for absence of starvation.

Chapter 7

Verifying a deadlock-free and starvation-free readers-writers

In the previous chapter we have seen that there is a starvation issue in QT's condition variable. It is still possible there also exists starvation or deadlock issues in the readers-writers algorithm itself. In this chapter we first check the readers-writers algorithm. As it turns out, the algorithm is correct. Therefore it is possible to construct a new condition variable for QT with the same specification as the condition variable primitive used in verifying the readers-writers algorithm. The implementation of such a class is discussed, and verified. The readers-writers using the new condition variable is verified next. This chapter concludes with verifying all kinds of safety properties for readers-writers, indicating if it works as intended.

7.1 Verifying the readers-writers algorithm

The starvation issue can be avoided by ensuring that no non-signalled thread can get the mutex before the signalled thread (t in the counterexample of Figure 6.6) can start executing again. So a signalled thread requesting the mutex should get priority over threads trying to acquire the mutex for the first time. This is precisely the way the adjusted model of pthread_cond functions, as explained in the previous chapter, with a priority queue and a normal queue. QReadWriteLock can use the POSIX primitives more directly, so this particular starvation issue should not occur again. This means that QMutex functions as a wrapper around pthread_mutex and QWaitCondition functions as a wrapper around pthread_cond. Although this solves our problem, it is not a solution as such a pthread_cond with described properties is not available[1]. However successful verification indicates the readers-writers algorithm itself is starvation free. Both verifying for absence of deadlock and starvation succeed. The results are listed below. As the algorithm is starvation and deadlock free, we can continue with creating a correct implementation of QWaitCondition, with the same semantics as our model of pthread_cond.

[1] Tests indicate that POSIX mutexes are on some platforms implemented as a FIFO queue, so no starvation issues originating from the mutex can occur. However, a thread blocked on a pthread_cond variable, when signalled, request the mutex without extra priority. This can be the source of starvation problems, as explained in the previous chapter.

Memory usage in mebibytes if checking for assertion violations and deadlocks, with a maximum of 3072 mebibyte memory:

Max Locks	1	3	5	7
2 Threads	2.40	3.08	5.65	11.89
3 Threads	4.00	114.42	1118.98	> 3072
4 Threads	85.25	n/a	n/a	n/a

Reached depth if checking for assertion violations and deadlocks, with a maximum of 3072 mebibyte memory:

Max Locks	1	3	5	7
2 Threads	450	3461	10836	23418
3 Threads	14316	439339	2667541	n/a
4 Threads	526515	n/a	n/a	n/a

Memory usage in mebibytes if checking for starvation, with a maximum of 3072 mebibyte memory:

Max Locks	1	3	5	7
2 Threads	2.41	2.43	2.83	3.23
3 Threads	4.37	31.47	173.29	552.82
4 Threads	138.01	n/a	n/a	n/a

Reached depth if checking for starvation, with a maximum of 3072 mebibyte memory:

Max Locks	1	3	5	7
2 Threads	536	1150	1570	1986
3 Threads	13617	95151	218847	403487
4 Threads	515703	n/a	n/a	n/a

7.2 Constructing a starvation-free condition variable

For a real solution we have to construct a starvation-free *condition variable* (QWaitCondition), out of Posix starvation-free *mutexes* and starvation-prone condition variables. To construct a starvation-free QWaitCondition with these basics primitives we will also have to re-implement QMutex. This because preservation of the order of threads calling the condition variable as well as giving priority to signalled threads in the mutex is needed to avoid starvation. Both must be implemented manually in QMutex and QWaitCondition, as the Posix primitives do not have these features. As the Posix condition variable is unreliable due to the spurious wake-ups, the correct order is not preserved. As a Posix mutex has no sense of priority[2], giving priority to specific threads is not possible inside a Posix mutex. As Posix does not define an interface to actively letting specific threads sleep and wake them up, the standard construct pthread_cond has to be used for this purpose. We will first describe the implementation of QMutex and continue with the implementation of QWaitCondition.

[2] Only superusers can set a different scheduling policy, and select a policy with different priorities per thread. This is not viable for typical Qt applications.

```
typedef QMutex {
  pthread_mutex_t mutex;
  pthread_cond_t normalWait;
  pthread_cond_t transferWait;
  byte transferWaiting = 0;
  byte currentLevel = 0;
  byte nextLevel = 0;
};

inline QMutex_lock(this) {
  pthread_mutex_lock(this.mutex);
  level = this.nextLevel;
  this.nextLevel = (this.nextLevel + 1) % THREADS;
  do
  :: this.transferWaiting > 0 || level != this.currentLevel ->
     pthread_cond_wait(this.normalWait, this.mutex);
  :: else ->
     break;
  od;
  level = 0;
  this.currentLevel = (this.currentLevel + 1) % THREADS;
}

inline QMutex_wakeNext(this) {
  if
  :: this.transferWaiting > 0 ->
     pthread_cond_broadcast(this.transferWait, this.mutex);
  :: else ->
     pthread_cond_broadcast(this.normalWait, this.mutex);
  fi;
}

inline QMutex_unlock(this) {
  QMutex_wakeNext(this);
  pthread_mutex_unlock(this.mutex);
}
```

Figure 7.1: Starvation-free PROMELA model of QMutex.

QMutex naturally includes a pthread_mutex variable, so mutual exclusion is available. On the beginning of the `lock` method of QMutex, this mutex is acquired. In the `unlock` method, the mutex is released. Two pthread_cond variables are also included in QMutex, so threads can sleep while waiting on the QMutex or on the QWaitCondition (which does not include any POSIX primitives). One condition variable, called `transferWait`, is for waiting and signalled threads. The other one, called `normalWait`, is for new threads obtaining the mutex but which turn has yet to come (both explained later on). An integer named `transferWaiting` indicates the number of threads currently signalled and waiting to acquire the mutex. If this number is not equal to zero, only signalled threads may obtain the mutex. If a thread waiting on a condition variable gets signalled, it is appended to the queue of threads waiting on the pthread_mutex. Due to this reason it is possible that non-signalled threads obtains a lock (on a QMutex object) before a signalled thread can acquire this mutex. This should not occur. So if a thread is not allowed to acquire the mutex due to this, it remembers its position in the queue and sleeps on the condition variable. The order is remembered by the thread in a local `level` variable. When entering the `lock` method of QMutex the `nextLevel` counter is copied to this `level` variable, and subsequently `nextLevel` is increased. In this way each waiting thread has therefore a unique number. The thread with the lowest number can only pass and obtain the mutex. This lowest number is remembered in the `currentLevel` variable. When the signalled thread is done with the lock, the `unlock` method of QMutex is called. If threads with priority (previously signalled) are waiting (as indicated by the `transferWaiting` variable), all threads waiting on the `transferWait` condition variable are signalled. This is needed because of the

```
#define waiters ((this.nextLevel - this.currentLevel + THREADS) % THREADS)

typedef QWaitCondition {
  byte currentLevel = 0;
  byte nextLevel = 0;
  byte wakeups = 0;
};

inline QWaitCondition_wait(this, m) {
  level = this.nextLevel;
  this.nextLevel = (this.nextLevel + 1) % THREADS;
  do
  :: this.wakeups == 0 || level != this.currentLevel ->
     QMutex_wakeNext(m);
     pthread_cond_wait(m.transferWait, m.mutex);
  :: else ->
     break;
  od;
  level = 0;
  mutex.transferWaiting--;
  assert(m.transferWaiting >= 0);
  this.wakeups--;
  this.currentLevel = (this.currentLevel + 1) % THREADS;
}

inline QWaitCondition_wakeOne(this, m) {
  if
  :: this.wakeups < waiters ->
     mutex.transferWaiting++;
     this.wakeups = this.wakeups+1;
  :: else
  fi;
  pthread_cond_broadcast(m.transferWait, m.mutex);
}

inline QWaitCondition_wakeAll(this, m) {
  m.transferWaiting = m.transferWaiting + (waiters - this.wakeups);
  this.wakeups = waiters;
  pthread_cond_broadcast(m.transferWait, m.mutex);
}
```

Figure 7.2: Starvation-free PROMELA model of QWaitCondition.

spurious wake-ups. If no such threads are waiting, all threads waiting on the `normalWait` condition variable are signalled. The signalled threads will obtain the mutex in correct order, as the order is explicitly remembered.

Next a new model of the QWaitCondition class has to be constructed. The class has three attributes, all integers. Using a similar construction as in QMutex, each threads waiting on the condition generates an unique number for itself. The number is generated from the attribute `nextLevel`. The thread with the unique number `currentLevel` is the next one to continue execution. The third attribute is called `wakeups`. The number of threads allowed to progress is contained therein. This variable is absent from QMutex because if there are no signalled thread waiting to reacquire the mutex, all threads are allowed to progress.

To construct a real implementation, the `wakeOne` and `wakeAll` methods of QWaitCondition must have access to the correct QMutex. One can allow this access by setting a pointer inside a QWaitCondition object to the QMutex object. This pointer must be set in the `wait` method, and is cleared if no threads are waiting anymore on the condition variable. This is according to the specification of QWaitCondition, which states that all threads waiting on a QWaitCondition must call the `wait` method with the same QMutex object. As pointers are absent from PROMELA, an extra argument containing the mutex is added to the `wakeOne` and `wakeAll` methods of QWaitCondition.

The QMutex is listed in Figure 7.1, QWaitCondition is listed in Figure 7.2. A couple of small remarks can be made about the model. The variable `level` must be added to the top of `user` process as listed in Figure 6.2. Strictly, the statement `level = 0` in both classes is not needed. But for our model it is useful, as it reduces the number of values the variable can contain, so the state space explored is also reduced. Adding these statements results in a significant memory reduction, even approaching 80% in some test runs. All the integer attributes are bound to the maximum number of threads, for the same reason of state space reduction. Using modulo operations, one can still obtain a correct result, but one has to perform some extra calculations. For reason of clarity an extra define `waiters` is used in QWaitCondition. This indicates the number of waiting threads. If program memory is at a premium one can adjust QMutex to use only one condition variable. Performance will probably suffer slightly, as too many threads are woken. But the resulting code is still starvation free, as threads can suffer spurious wake-ups and the code checks for this case.

The algorithm uses two starvation-prone condition variables (Posix), even if no starvation-free condition variables are used. But if there are many starvation-free condition variables needed, this algorithm saves all but two condition variables. The performance of this new algorithm, compared with the old one, is not measured due to time constrains.

7.3 Verifying the new condition variable

We can verify the new condition variables by constructing a special model in Promela. We have to verify that a thread waiting on a condition is always allowed to proceed if the condition is signalled. We also have to verify that the situation detected in the previous chapter (another thread acquires the mutex for the first time while a signalled thread is waiting) can not occur. For this we need threads which locks the mutex and we need threads which wait on the condition. We can merge these requirements into one `inline` definition called `usage`. This definition needs a mutex and condition variable. These are declared as `m` and `c` respectively. For verifying absence of starvation we create a process called one which contain a progress state and repeatedly executes the `usage` definition. The other processes of type usageProc repeatedly executes the same definition, but without a progress label. In the model, one process has to signal the waiting processes, otherwise the model would deadlock. To facilitate this

```
QMutex m;
QWaitCondition c;
byte waiting = 0;

inline usage() {
   QMutex_lock(m);
   waiting++;
   QWaitCondition_wait(c, m);
   QMutex_unlock(m);
}

active[1] proctype one() {
   progress:
   do
   :: usage();
   od;
}

active[1] proctype waker() {
   do
   :: atomic {
         waiting > 0 ->
         waiting--;
      }
      QMutex_lock(m);
      QWaitCondition_wakeOne(c, m);
      QMutex_unlock(m);
   od;
}

active[THREADS-2] proctype usageProc() {
   do
   :: usage();
   od;
}
```

Figure 7.3: Promela model simulating usage of a condition variable.

another variable is introduced called `waiting`, depicting the number of waiting processes. If this value is greater than zero, a special process called waker locks the mutex and wakes signals on the condition variable. The model is listed in Figure 7.3.

Absence of deadlock and starvation has been verified for this model for a maximum of 5 threads. Therefore our new condition variables satisfies our requirements for condition variables. We can now proceed with using them in the readers-writers algorithm.

7.4 Verifying absence of starvation

Next we verify that the readers-writers implementation has no starvation issues, if the new algorithm for the QWaitCondition class is used. As the implementation of the QWaitCondition class is based on a new algorithm, we also have to reverify the absence of deadlock result for the readers-writers algorithm. Results from this verification concludes this section.

Memory usage in mebibytes if checking for starvation:

Max Locks	1	2	3	4	5	6
2 Threads	5.13	6.04	7.08	8.32	9.95	11.77
3 Threads	393.37	2997.60	6243.38	16424.23	42830.06	59323.46
4 Threads	54775.47	n/a	n/a	n/a	n/a	n/a

Reached depth if checking for starvation:

Max Locks	1	2	3	4	5	6
2 Threads	3387	6075	7659	9091	10523	11955
3 Threads	865975	2794713	5586802	8771577	12525376	16647916
4 Threads	42752566	n/a	n/a	n/a	n/a	n/a

Runtime in hours, minutes and seconds if checking for starvation[3]:

Max Locks	1	2	3	4	5	6
2 Threads	0:00:37	0:00:35	0:00:35	0:00:35	0:00:36	0:00:35
3 Threads	0:01:27	0:05:39	0:15:32	0:35:27	1:10:50	2:03:10
4 Threads	2:39:50	n/a	n/a	n/a	n/a	n/a

We also verified the adjust algorithm for absence for deadlock and assertion violations. The result of these runs is listed below.

Memory usage in mebibytes if checking for deadlock and assertion violations:

Max Locks	1	2	3	4
2 Threads	4.93	7.10	12.94	24.34
3 Threads	380.15	5166.47	25768.86	94744.03
4 Threads	33193.62	n/a	n/a	n/a

[3] As other cores on the machine were used by other programs, the runtime measurement is not entirely accurate across the different runs.

Reached depth if checking for deadlock and assertion violations:

Max Locks	1	2	3	4
2 Threads	3259	12275	28827	51774
3 Threads	999667	9137812	37654838	110571350
4 Threads	52711112	n/a	n/a	n/a

Runtime in hours, minutes and seconds if checking for deadlock and assertion violations[4]:

Max Locks	1	2	3	4
2 Threads	0:00:26	0:00:26	0:00:26	0:00:26
3 Threads	0:00:43	0:04:50	0:24:50	1:34:28
4 Threads	0:40:14	n/a	n/a	n/a

7.5 Verifying safety properties of readers-writers

To check the safety properties a couple of variables are introduced in the readers-writers model to track the number of threads having write locks (called `writers`) and having read locks (called `readers`). Matching code is added to the model to keep track of these model variables. The code needed to keep track of these variables is inserted at appropriates place in the methods of QReadWriteLock. The `readers` and `writers` variables are only incremented on a non-reentrant call of a thread, and therefore decremented only by the final unlock. The other variables stated in the properties are attributes of QReadWriteLock.

We now continue with checking Linear Temporal Logic (LTL) safety properties of the algorithm. These properties are checked by querying Spin with a LTL expression. At this stage the algorithm contains no deadlocks and no writer starvation issues, but can still contain conceptually flawed behaviour, for example allowing both a reader and a writer enter the critical section at the same time. A test called `outsideCS` is introduced, true indicating *no* change can occur inside the lock structure. In other words no thread has obtained the mutex, as indicated by the boolean attribute `mutex.mutex.locked` of the QReadWriteLock class, and no signalled thread is waiting to acquire the mutex, as indicated by the attribute `mutex.transferWaiting`. The relevant properties mentioned in Section 2.3 can now be easily formalised. These are listed below. The `readersWaiting` and `writersWaiting` variables used are attributes from the QReadWriteLock object.

- $[\,]\ (\texttt{readers} = 0\ \vee\ \texttt{writers} = 0)$

  There are not simultaneously writers *and* readers allowed.

- $[\,]\ (\texttt{writers} \leq 1)$

  No more than one writer is allowed.

[4] As other cores on the machine were used by other programs, the runtime measurement is not entirely accurate across the different runs.

- [ ] (`outsideCS` → (`writersWaiting` > 0 → (`readers` $> 0 \ \vee$ `writers` > 0)))
  States that the only possibility of waiting writers is when there are readers or writers busy, but only when there is no change to the lock.
- [ ] (`outsideCS` → (`readersWaiting` > 0 → (`writers` $> 0 \ \vee$ `writersWaiting` > 0)))
  States that the only possibility of waiting readers is when there are writers waiting or writers busy, but only when there is no change to the lock.

We have verified each of the stated invariants for a different number of threads and maximum lock operations. The results are listed below.

Memory usage in mebibytes if checking for safety properties:

Max Locks	1	2	3	4
2 Threads	5.28	8.60	16.46	31.23
3 Threads	512.95	6966.52	35119.27	127673.20
4 Threads	48244.48	n/a	n/a	n/a

Reached depth if checking for safety properties:

Max Locks	1	2	3	4
2 Threads	6679	23625	50368	87238
3 Threads	1861295	15973954	64586554	187270543
4 Threads	94333930	n/a	n/a	n/a

Runtime in hours, minutes and seconds if checking for safety properties[5]:

Max Locks	1	2	3	4
2 Threads	0:00:28	0:00:29	0:00:29	0:00:30
3 Threads	0:00:52	0:06:27	0:33:12	2:08:09
4 Threads	1:01:50	n/a	n/a	n/a

[5] As other cores on the machine were used by other programs, the runtime measurement is not entirely accurate across the different runs.

Chapter 8

Related and future work

8.1 Related work

In [29] model checking and theorem proving are combined to verify a non-reentrant readers-writers algorithm. A tabular specification of the systems is used and can be automatically translated into SPIN and PVS models. Safety and clean completion properties are derived semi-automatically. If an error is found, it is less easily fixed and reverified as with the method described in this thesis. Also their proposed approach is exclusively for verifying concurrent programs, yet the results of this thesis are part of a general approach for verifying any program [17].

An interesting approaches to model checking concurrent programs is proposed in [28]. This approach checks real assembly code produced by the GCC compiler and model checks it with help of the GNU Debugger. This approach is not especially useful for the algorithm in this thesis: as the algorithm locks a global mutex while executing the three functions, no interleaving on instruction level is possible. The use of the proposed method will likely increase the runtime of model checking as the model contains an unneeded level of detail. Also for each platform the model should be reverified. As QT is supported on at least four hardware architectures, the implementation should be reverified for each, further increasing the verification time.

The method described in [33] is similar to the previous approach. This approach checks the platform-independent LLVM intermediate language produced by the LLVM compiler. It uses the SPIN model checker and includes support for the most used POSIX primitives. This method can be applied to our problem, but is not suitable because do not support liveness properties.

8.2 Future work

As the process of constructing the models consisted of a considerable time investment, automatic conversion methods are worth considering. Java PathFinder can automatically convert Java to PROMELA models, although introducing abstractions is more difficult as JPF works on a whole input program [22, 3, 32]. The previous mentioned approach of model checking LLVM Intermediate Language is an interesting new approach as the model checker does not need to have knowledge about the source code language [33]. Another approach is used by the tool Zing, as it uses program abstractions for modular model checking [4]. An overview of model checkers operating directly on C++ source code if found [30]. A comparison between these automatic method and the approach of this thesis could yield interesting results.

There are several options to explore more extensive models in Spin, including a graph representation of the visited spaces and bitstate hashing [10, 25, 24]. Another approach is to use distributed model checking [9, 8]. Mainly because of time constrains these methods were not applied.

Qt's readers-writers including the corrected implementation of condition variables is not yet proven, as [18] proves Qt's readers-writers with an abstract version of the fixed condition variables.

The last couple of years concurrent mutual exclusion algorithms receive more attention. These algorithms, such as [27, 11], provide readers-writers lock for kernels. Key to their design is they minimise (cache) contention between cores and processors, so higher efficiency is obtained. Concepts of the algorithm can be transferred to the user space readers-writers lock, to increase efficiency. The algorithms in the papers are not verified, which create an opportunity for future work.

The number of concurrent threads which can be verified in our models can be increased to gain more confidence in our models. A smaller model of a specific component can be introduced with precisely the same behaviour as the original component. As the component is more simple, the amount of memory needed decreases. This process should be automatic, so no human errors can be introduced. Also the simple version must contain enough detail so the verification remains correct.

Chapter 9

Conclusion

In this thesis we started with a reentrant readers-writers algorithm used in Qt 4.3. The model depends on condition variables. Qt's implementation of condition variables were modelled. A deadlock issue was found and corrected. The deadlock was an error in the readers-writers algorithm, as it occurred even with an abstract model of condition variables. The needed change to the algorithm is discussed, and the models are subsequently adjusted. The corrected algorithm was verified free of deadlocks, using Qt's implementation *and* an abstract model of condition variables.

Using the same abstract starvation-free model of condition variables, the adjusted readers-writers algorithm was also verified free of starvation. However, if using Qt's implementation of condition variables the algorithm suffers from starvation issues. The source of the problem were the condition variables used. Not only the implementation, but also the algorithm of Qt's condition variables was found to be prone to starvation. This affects the readers-writers implementation as it also becomes prone to starvation. A new algorithm for condition variables had to be constructed because replacing Qt's condition variables by Posix variants was not valid, as on most Posix platforms, including Linux and FreeBSD, Posix condition variables are prone to starvation. An algorithm for constructing a starvation-free condition variable out of a starvation-prone condition variable was presented and verified free of deadlock and starvation. By using this algorithm instead of Qt's condition variables, we were also able to verify that the adjusted readers-writers is free of deadlock and starvation. Furthermore to increase the confidence in the readers-writers algorithm, a number of safety properties was successfully verified, even if using the new algorithm for condition variables.

We improved the previous work in several ways. The Qt classes on which the readers-writers implementation depends were modelled in a more accurate way. The two classes implement mutexes and conditional variables. These are implemented in distinct classes, QMutex and QWaitCondition, which depend on operating system primitives. The previous work combined the two classes in one model. The model used a separate process to serialise access and was based on the specifications of the classes rather than their implementations. The new model more closely resembles the execution of threaded programs and eliminates the extra process. Also, instead of the specification, the actual implementation of the condition variable was modelled in this thesis. Using a model of this implementation does not influence the observation of a deadlock. The previous work did not consider starvation, which is, as mentioned above, extensively explored in this thesis. Modelling QT's implementation of condition variables made it possible to detect the starvation issue in the condition variable, which also affecting the readers-writers algorithm. All these improvements increase the confidence in the verification of the readers-writers algorithm.

The model contains many details, therefore verifying the model costs enormous amounts of memory. A special machine must be used to verify these kind of algorithms in detail. In these experiments to verify the algorithm we have verified properties for a maximum of 3 threads, and for a maximum number of lock operations of 4. The experiment runs in about 6 hours (two for deadlock checking, two for the safety properties, and two for starvation checking). If we increase these values slightly, the execution time worsens drastically and/or the memory usage increases above 128 gibibyte, the memory limit for our machines. It was not possible to use more abstractions, as the algorithm examined has a starvation issue in a supporting class. If the source code of the supporting class was not modelled, but an abstract version used instead, the starvation issue was not found. The proper use of abstractions is therefore important to find all problems in the source code.

The new algorithm for starvation-free condition variables presented in this thesis depends only on starvation-prone condition variables and starvation-free mutexes. Although the exact performance of this new algorithm is unknown but probably (a little) slower due increased complexity, in situations were liveness is more important than (extreme) performance this algorithm is an easy and effective way to create starvation-free condition variables. This algorithm is useful on most POSIX platforms, like LINUX and FREEBSD, as the condition variable available on those platforms are prone to starvation.

Appendix A

Promela models

A.1 Some standard functions

```
/**
* Handy stuff
* Written by Bernard van Gastel, <bvgastel@bitpowder.com>
* Including:
* - picking arbitrary numbers, 'pick'
* - min/max
* - setting and getting bits
*/
#ifndef STD
#define STD

/*
inline arbitrary(n) {
   atomic {
      assert(n > 0);
      n--;
      do
      :: n > 0 -> n--;
      :: break;
      od;
   }
}
*/
/*
inline arbitrary(n) {
   atomic {
      assert(n > 0);
      int retval = n / 2;
      do
      :: retval > 0 -> retval--;
      :: retval < n-1 -> retval++;
      :: break;
      od;
      assert(0 <= retval && retval < n);
      n = retval;
   }
}
*/

// var = {x | lower >= x >= upper};
inline pick (var, lower, upper) {
        atomic {
                var = lower;
                do
                :: (var < (upper)) -> var++;
                :: break
                od
        }
}
```

```
/*
// c = min(a,b);
inline min(c, a, b) {
   if
   :: a < b -> c = a;
   :: a >= b -> c = b;
   fi;
}
*/
//#define min(a,b) (((a) > (b))*(b) + ((b) >= (a))*(a))
//#define max(a,b) (((a) > (b))*(a) + ((b) >= (a))*(b))
#define min(a,b) ((a) > (b) -> (b) : (a))
#define max(a,b) ((a) > (b) -> (a) : (b))

#define getBit(value, bit) (((value) >> (bit)) & 1)
#define setBitTo(value, bit, to) value = (((value) & ~(1 << (bit))) | ((to) << (bit)))

#endif
```

A.2 Queue Abstraction

```
/**
* To nativly use a queue in Promela.
* Written by Bernard van Gastel, <bvgastel@bitpowder.com>
* Usage:
* - decleration of a queue:
*   queue([queuename], [max size], [type of elements]);
*   e.g. queue(intqueue, 10, int);
* - enqueue
*   enqueue([queuename], [item]);
*   e.g. enqueue(intqueue, 10);
* - dequeue
*   dequeue([queuename], [variable name]);
*   e.g. dequeue(intqueue, x); (with the result in x)
*/
#ifndef QUEUE_H
#define QUEUE_H

#define queue(name, size, elements) chan name = [size] of {elements}

#define enqueue(que, elem) que!elem
//atomic { assert(nfull(que)); que!elem }

//#define enqueueFirst(que, elem) que!!elem

#define dequeue(que, elem) que?elem

#endif
```

A.3 Schedular Abstraction

```
#ifndef SCHEDULAR
#define SCHEDULAR
// scheduler
chan cont = [0] of {pid};

#define processSleep() cont?eval(_pid)
#define processWake(processid) cont!processid

//#define processSleep() cont!_pid
//#define processWake(processid) cont?eval(processid)

/*
inline processSleep() {
   atomic {
      printf("%d: sleeping\n", _pid);
      cont?eval(_pid);
      printf("%d: waked, continuing\n", _pid);
   }
}
*/

/*
inline processWake(processid) {
   atomic {
      printf("%d: waking %d\n", _pid, processid);
      cont!processid;
      printf("%d: waked %d\n", _pid, processid);
   }
   //assert(1);
}
*/

#endif
```

A.4 pthread_mutex - opportunistic

```
#ifndef PTHREAD_MUTEX
#define PTHREAD_MUTEX

#ifndef NT
#define NT 255
#endif

#define pthread_mutex_busy(this) (this.locked)

typedef pthread_mutex_t {
   byte locked = false; // if this is a bool then it uses far more space for state space (I guess 32 or 64 bits
};

inline pthread_mutex_lock(d) {
   atomic {
      //d.locked == false;
      !d.locked;
      d.locked = true;
   }
}

inline pthread_mutex_unlock(d) {
   atomic {
      assert(d.locked);
      d.locked = false;
   }
}

#endif
```

A.5 pthread_mutex - starvation free

```
#ifndef PTHREAD_MUTEX_BASIC_FAIR
#define PTHREAD_MUTEX_BASIC_FAIR

#ifndef NT
#define NT 255
#endif

#include SCHEDULAR_MODEL
#include "queue.pml"

#define pthread_mutex_busy(this) (this.locked)

typedef pthread_mutex_t {
   byte locked = false; // if this is a bool then it uses far more space for state space (I guess 32 or 64 bits
   queue(queue, THREADS, pid);
};

inline pthread_mutex_lock(this) {
   atomic {
      if
      // locked, but not by itself
      :: this.locked ->
         // wait;
         printf("%d: pthread_mutex_lock waiting\n", _pid);
         enqueue(this.queue, _pid);
         processSleep();
         printf("%d: pthread_mutex_lock continuing\n", _pid);
      :: else //-> break;
      fi;
      assert(!this.locked);
      this.locked = true;
      //printf("locked: pid %d to nestlevel %d\n", _pid, this.count);
      printf("%d: pthread_mutex_lock done\n", _pid);
   }
}

inline pthread_mutex_unlock(this) {
   atomic {
      assert(this.locked);
      this.locked = false;
      printf("%d: pthread_mutex_unlock done\n", _pid);
      if
      :: nempty(this.queue) ->
         dequeue(this.queue, p1);
         printf("%d: pthread_mutex_unlock waking %d\n", _pid, p1);
         processWake(p1);
         p1 = NT;
      :: empty(this.queue)
      fi;
      //printf("unlocked: pid %d to nestlevel %d\n", _pid, d.count);
   }
}

/*
queue(pthread_mutex_reverse_queue, THREADS, pid); // for relock

inline pthread_mutex_relock(this) {
   atomic {
      do
      // locked, but not by itself
      :: this.locked ->
         // wait;
         printf("%d: pthread_mutex_lock re waiting\n", _pid);
         //enqueueFirst(this.queue, _pid);
         assert(empty(pthread_mutex_reverse_queue));
         do
         :: nempty(this.queue) ->
            dequeue(this.queue, p1);
            enqueue(pthread_mutex_reverse_queue, p1);
         :: empty(this.queue) -> break;
         od;
         assert(empty(this.queue));
         enqueue(this.queue, _pid);
         do
         :: nempty(pthread_mutex_reverse_queue) ->
            dequeue(pthread_mutex_reverse_queue, p1);
            enqueue(this.queue, p1);
         :: empty(pthread_mutex_reverse_queue) -> break;
         od;
         assert(empty(pthread_mutex_reverse_queue));
         processSleep();
         printf("%d: pthread_mutex_lock continuing\n", _pid);
      :: !this.locked -> break;
      od;
      assert(!this.locked);
      this.locked = true;
      //printf("locked: pid %d to nestlevel %d\n", _pid, this.count);
      printf("%d: pthread_mutex_lock done\n", _pid);
   }
}
*/
#endif
```

A.6 pthread_mutex - starvation free with support for starvation-free pthread_cond

```
#ifndef PTHREAD_MUTEX_BASIC_FAIR
#define PTHREAD_MUTEX_BASIC_FAIR

#ifndef NT
#define NT 255
#endif

#include SCHEDULAR_MODEL
#include "queue.pml"

#define pthread_mutex_busy(this) (this.locked)

typedef pthread_mutex_t {
   byte locked = false; // if this is a bool then it uses far more space for state space (I guess 32 or 64 bits
   queue(queue, THREADS, pid);
   queue(requeue, THREADS, pid);
};

inline pthread_mutex_lock(this) {
   atomic {
      if
      // locked, but not by itself
      :: this.locked ->
         // wait;
         printf("%d: pthread_mutex_lock waiting\n", _pid);
         enqueue(this.queue, _pid);
         processSleep();
         printf("%d: pthread_mutex_lock continuing\n", _pid);
      :: else ->
         this.locked = true;
      fi;
      assert(this.locked);
      //printf("locked: pid %d to nestlevel %d\n", _pid, this.count);
      printf("%d: pthread_mutex_lock done\n", _pid);
   }
}

inline pthread_mutex_unlock(this) {
   atomic {
      assert(this.locked);
      //this.locked = false;
      printf("%d: pthread_mutex_unlock done\n", _pid);
      if
      :: nempty(this.requeue) ->
         dequeue(this.requeue, p1);
         printf("%d: pthread_mutex_unlock waking priority %d\n", _pid, p1);
         processWake(p1);
         p1 = NT;
      :: empty(this.requeue) ->
         if
         :: nempty(this.queue) ->
            dequeue(this.queue, p1);
            printf("%d: pthread_mutex_unlock waking %d\n", _pid, p1);
            processWake(p1);
            p1 = NT;
         :: empty(this.queue) ->
            this.locked = false;
         fi;
      fi;
      //printf("unlocked: pid %d to nestlevel %d\n", _pid, d.count);
   }
}

inline pthread_mutex_relock(this) {
   atomic {
      assert(this.locked);
      //this.locked = true;
      //printf("locked: pid %d to nestlevel %d\n", _pid, this.count);
      printf("%d: pthread_mutex_relock done\n", _pid);
   }
}

// put first in new queue,
inline pthread_mutex_requeue_one(this, process) {
   atomic {
      assert(this.locked);
      enqueue(this.requeue, process);
      assert(nempty(this.requeue));
   }
}

// put first in new queue,
inline pthread_mutex_requeue_queue(this, toBeQueued) {
   atomic {
      assert(this.locked);
      //assert(nempty(toBeQueued));
      do
      :: nempty(toBeQueued) ->
         dequeue(toBeQueued, p1);
         enqueue(this.requeue, p1);
      :: empty(toBeQueued) ->
         break;
      od;
      p1 = NT;
      assert(empty(toBeQueued));
      //assert(nempty(this.requeue));
   }
}

#endif
```

A.7 pthread_cond - opportunistic

```
#ifndef PTHREAD_COND_ABS
#define PTHREAD_COND_ABS

#include PTHREAD_MUTEX_MODEL

typedef pthread_cond_t {
   byte waiters = 0;
   chan condcont = [0] of {bit};
};

inline pthread_cond_wait(this, _mutex) {
   atomic {
      this.waiters++;
      pthread_mutex_unlock(_mutex);
      this.condcont!1;
   }
   pthread_mutex_lock(_mutex);
}

inline pthread_cond_signal(this, _mutex) {
   // so each receive operation on cont is successful
   atomic {
      if
      :: this.waiters > 0 ->
         this.waiters--;
         this.condcont?1;
      :: else
      fi;
   }
}

inline pthread_cond_broadcast(this, _mutex) {
   // so each receive operation on cont is successful
   atomic {
      do
      :: this.waiters > 0 ->
         this.waiters--;
         this.condcont?1;
      :: else ->
         break;
      od;
   }
}
#endif
```

A.8 pthread_cond - starvation free

```
#ifndef PTHREAD_COND_FAIR
#define PTHREAD_COND_FAIR

#ifndef NT
#define NT 255
#endif

#include PTHREAD_MUTEX_MODEL
#include "queue.pml"
#include SCHEDULAR_MODEL

#define pthread_cond_busy(this) (rwlock.readerWait.cond.queue?[_])
//(nempty(this.queue))

typedef pthread_cond_t {
   queue(queue, THREADS, pid);
};

inline pthread_cond_wait(this, /* pthread_mutex_t */ _mutex) {
   #ifdef SPURIOUS_WAKEUP
   if
   ::
   #endif
      // immediately tries to obtain mutex, so either one is waiting for the condition or one is waiting for the
      atomic {
         enqueue(this.queue, _pid);
         pthread_mutex_unlock(_mutex);
         processSleep();
         pthread_mutex_relock(_mutex);
      }
   #ifdef SPURIOUS_WAKEUP
   :: pthread_mutex_unlock(_mutex);
      pthread_mutex_lock(_mutex);
   fi;
   #endif
}

inline pthread_cond_signal(this, _mutex) {
   atomic {
      if
      :: nempty(this.queue) ->
         dequeue(this.queue, p1);
         pthread_mutex_requeue_one(_mutex, p1);
         p1 = NT;
      :: empty(this.queue)
      fi;
   }
}

inline pthread_cond_broadcast(this, _mutex) {
   atomic {
      pthread_mutex_requeue_queue(_mutex, this.queue);
      assert(empty(this.queue));
   }
}
#endif
```

A.9 QMutex - wrapper around pthread_mutex

```
#ifndef QMUTEX
#define QMUTEX
```

```
#include PTHREAD_MUTEX_MODEL

#define qmutex_busy(this) (pthread_mutex_busy(this.mutex))

typedef QMutex {
   pthread_mutex_t mutex;
};

inline QMutex_lock(this) {
   pthread_mutex_lock(this.mutex);
}

inline QMutex_unlock(this) {
   pthread_mutex_unlock(this.mutex);
}

#endif
```

A.10 QMutex - starvation-free version

```
#ifndef QMUTEX
#define QMUTEX

#include PTHREAD_MUTEX_MODEL
#include PTHREAD_COND_MODEL

#define qmutex_busy(x) (pthread_mutex_busy(x.mutex) || x.transferWaiting > 0)
#define waiters ((this.nextLevel - this.currentLevel + THREADS) % THREADS)

typedef QMutex {
   pthread_mutex_t mutex;

   pthread_cond_t normalWait;
   pthread_cond_t transferWait;

   byte transferWaiting = 0;

   byte currentLevel = 0;
   byte nextLevel = 0;
};

inline QMutex_lock(this) {
   pthread_mutex_lock(this.mutex);
   level = this.nextLevel;
   this.nextLevel = (this.nextLevel + 1) % THREADS;
   do
   :: this.transferWaiting > 0 || level != this.currentLevel ->
      printf("%d: QMutex_lock: wait (level = %d)\n", _pid, level);
      pthread_cond_wait(this.normalWait, this.mutex);
      printf("%d: QMutex_lock: woken (level = %d)\n", _pid, level);
   :: else ->
      break;
   od;
   level = 0;
   this.currentLevel = (this.currentLevel + 1) % THREADS;
}

inline QMutex_wakeNext(this) {
   if
   :: this.transferWaiting > 0 ->
      pthread_cond_broadcast(this.transferWait, this.mutex);
   :: else ->
      pthread_cond_broadcast(this.normalWait, this.mutex);
   fi;
}

inline QMutex_unlock(this) {
   QMutex_wakeNext(this);
   pthread_mutex_unlock(this.mutex);
}
```

```
inline QMutex_transferLock(this) {
   pthread_cond_broadcast(this.transferWait, this.mutex);
}

inline QMutex_takeOverLock(this) {
   QMutex_wakeNext(this);
   pthread_cond_wait(this.transferWait, this.mutex);
}

#endif
```

A.11 QWaitCondition - Qt 4.3 & Qt 4.4

```
#ifndef QWAITCONDITION
#define QWAITCONDITION

#include "std.pml"
#include QMUTEX_MODEL
#include PTHREAD_COND_MODEL
#include PTHREAD_MUTEX_MODEL
#ifndef NT
#define NT 255
#endif
/*
   Klein beetje versimpelt, vooral de wait
*/

#define QWaitCondition_busy(this) (this.busy > 0)

typedef QWaitCondition {
   pthread_mutex_t mutex;
   pthread_cond_t cond;
   int waiters = 0;
   int wakeups = 0;
   byte busy = 0;
};

inline QWaitCondition_wait(this, _mutex) {
   this.busy++;
   printf("%d: begin qwait::wait\n", _pid);
   pthread_mutex_lock(this.mutex);
   this.waiters++;
   QMutex_unlock(_mutex);
   do
   :: this.wakeups == 0 -> pthread_cond_wait(this.cond, this.mutex);
   :: this.wakeups > 0 -> break;
   od;
   this.waiters--;
   this.wakeups--;
   pthread_mutex_unlock(this.mutex);
   QMutex_lock(_mutex);
   printf("%d: done qwait::wait\n", _pid);
   this.busy--;
}

inline QWaitCondition_wakeOne(this, _mutex) {
   pthread_mutex_lock(this.mutex);
   assert(this.wakeups <= this.waiters);
   this.wakeups = min(this.wakeups + 1, this.waiters);
   assert(this.wakeups <= this.waiters);
   pthread_cond_signal(this.cond, this.mutex);
   pthread_mutex_unlock(this.mutex);
}

inline QWaitCondition_wakeAll(this, _mutex) {
   pthread_mutex_lock(this.mutex);
   this.wakeups = this.waiters;
   pthread_cond_broadcast(this.cond, this.mutex);
   pthread_mutex_unlock(this.mutex);
}
#endif
```

A.12 QWaitCondition - starvation-free version

```
#ifndef QWAITCONDITION
#define QWAITCONDITION

#include QMUTEX_MODEL
#include "std.pml"

#define waiters ((this.nextLevel - this.currentLevel + THREADS) % THREADS)

typedef QWaitCondition {
   byte currentLevel = 0;
   byte nextLevel = 0;
   byte wakeups = 0;
};

inline QWaitCondition_wait(this, mutex2) {
   //this.waiters++;
   level = this.nextLevel;
   this.nextLevel = (this.nextLevel + 1) % THREADS;
   do
   :: this.wakeups == 0 || level != this.currentLevel ->
      QMutex_takeOverLock(mutex2);
   :: else ->
      break;
   od;
   level = 0; // only to reduce state space, 450 mb of 2200 mob saved (3 threads, 1 maxlocks)
   mutex2.transferWaiting--;
   assert(mutex2.transferWaiting >= 0);
   this.wakeups--;
   this.currentLevel = (this.currentLevel + 1) % THREADS;
   //this.waiters--;
}

inline QWaitCondition_wakeOne(this, mutex2) {
   if
   :: this.wakeups < waiters ->
      mutex2.transferWaiting++;
      this.wakeups = this.wakeups+1;
   :: else
   fi;
   QMutex_transferLock(mutex2);
}

inline QWaitCondition_wakeAll(this, mutex2) {
   mutex2.transferWaiting = mutex2.transferWaiting + (waiters - this.wakeups);
   this.wakeups = waiters;
   QMutex_transferLock(mutex2);
}

#endif
```

A.13 Usage of QWaitCondition

```
#include QMUTEX_MODEL
#include QWAITCONDITION_MODEL

QMutex m;
QWaitCondition c;
byte waiting = 0;

active[1] proctype waker() {
   byte level;
   pid p1;
   do
   :: atomic {
         waiting > 0 ->
         waiting--;
      }
      QMutex_lock(m);
      QWaitCondition_wakeOne(c, m);
      QMutex_unlock(m);
   od;
}

inline usage() {
   byte level;
   pid p1;
   QMutex_lock(m);
   waiting++;
   QWaitCondition_wait(c, m);
   QMutex_unlock(m);
}

#ifdef PROGRESS
active[1] proctype one() {
   progress:
   do
   :: usage();
   od;
}
#endif

proctype usageProc() {
   do
   :: usage();
   od;
}

byte icopy = 0;
init {
#ifdef PROGRESS
   int i = THREADS-2;
#else
   int i = THREADS-1;
#endif
   icopy = i;
   assert(i >= 1);
   do
   :: i > 1 -> i--; run usageProc();
   :: i == 1 -> i--;
      do
      :: usage();
      od;
   :: i <= 0 -> break;
   od;
}
```

A.14 QReadWriteLock - with deadlock / Qt 4.3

```
#ifndef QREADWRITELOCK
#define QREADWRITELOCK

#include QWAITCONDITION_MODEL
#include QMUTEX_MODEL

typedef QReadWriteLock {
   QMutex mutex;
   QWaitCondition readerWait;
   QWaitCondition writerWait;

   // readers writers lock
   int accessCount = 0;
   pid currentWriter = NT;
   int waitingReaders = 0;
   int waitingWriters = 0;
}

inline QReadWriteLock_lockForRead(this) {
   QMutex_lock(this.mutex);
   printf("%d: enter lockForRead (%d)\n", _pid, this.accessCount);
   do
   :: this.accessCount < 0 || this.waitingWriters > 0 ->
```

```
      this.waitingReaders++;
      printf("%d: waiting\n", _pid);
      QWaitCondition_wait(this.readerWait, this.mutex);
      printf("%d: continue\n", _pid);
      this.waitingReaders--;
   :: !(this.accessCount < 0 || this.waitingWriters > 0) ->
      break;
   od;
   this.accessCount = this.accessCount + 1;
   assert(this.accessCount > 0);
   printf("%d: leave lockForRead (%d)\n", _pid, this.accessCount);
   QMutex_unlock(this.mutex);
}

inline QReadWriteLock_lockForWrite(this) {
   QMutex_lock(this.mutex);
   printf("%d: enter lockForWrite (%d)\n", _pid, this.accessCount);
   pid self = _pid;
   do
   :: this.accessCount != 0 ->
      if
      :: this.accessCount < 0 && self == this.currentWriter ->
         break;
      :: else
      fi;
      this.waitingWriters++;
      printf("%d: waiting\n", _pid);
      QWaitCondition_wait(this.writerWait, this.mutex);
      printf("%d: continue\n", _pid);
      this.waitingWriters--;
   :: !(this.accessCount != 0) -> break;
   od;
   this.currentWriter = self;
   this.accessCount = this.accessCount - 1;
   assert(this.accessCount < 0);
   printf("%d: leave lockForWrite (%d)\n", _pid, this.accessCount);
   QMutex_unlock(this.mutex);
}

inline QReadWriteLock_unlock(this) {
   QMutex_lock(this.mutex);
   printf("%d: enter unlock (%d)\n", _pid, this.accessCount);
   assert(this.accessCount != 0);
   if
   :: this.accessCount > 0 -> this.accessCount = this.accessCount - 1;
   :: this.accessCount < 0 -> this.accessCount = this.accessCount + 1;
   fi;
   if
   :: this.accessCount == 0 ->
      this.currentWriter = NT;
      if
      :: this.waitingWriters > 0 ->
         QWaitCondition_wakeOne(this.writerWait, this.mutex);
      :: else ->
         if
         :: this.waitingReaders > 0 ->
            QWaitCondition_wakeAll(this.readerWait, this.mutex);
         :: else
         fi;
      fi;
   :: else
   fi;
   printf("%d: leave lock (%d)\n", _pid, this.accessCount);
   QMutex_unlock(this.mutex);
}

#endif
```

A.15 QReadWriteLock - deadlock free / Qt 4.4

```
#ifndef QREADWRITELOCK
#define QREADWRITELOCK
```

```
#include QWAITCONDITION_MODEL
#include QMUTEX_MODEL

typedef QReadWriteLock {
   QMutex mutex;
   QWaitCondition readerWait;
   QWaitCondition writerWait;

   // readers writers lock
   int threadCount = 0;
   int waitingReaders = 0;
   int waitingWriters = 0;

   pid currentWriter = NT;
   int count[THREADS] = 0;
}

inline QReadWriteLock_lockForRead(this) {
   // atomic lock
   QMutex_lock(this.mutex);
   printf("%d: want readlock (this thread nested %d deep). (%d/%d) (%d/%d)\n", _pid, this.count[_pid], readNest,
   if
   :: this.count[_pid] == 0 ->
      do
      :: (this.currentWriter != NT || this.waitingWriters > 0) ->
         this.waitingReaders++;
         printf("%d: wait for readlock (this thread nested %d deep)\n", _pid, this.count[_pid]);
         QWaitCondition_wait(this.readerWait, this.mutex);
         printf("%d: continue for readlock (this thread nested %d deep)\n", _pid, this.count[_pid]);
         this.waitingReaders--;
      :: !(this.currentWriter != NT || this.waitingWriters > 0) ->
         break;
      od;
      this.threadCount++;
      assert(this.waitingWriters == 0);
   :: else
   fi;
   this.count[_pid]++;
   // atomic unlock
   atomic {
      readNest++;
      if
      :: readNest == 1 && writeNest == 0 -> readers++;
      :: else
      fi;
      printf("%d: has readlock (this thread nested %d deep). (%d/%d) (%d/%d)\n", _pid, this.count[_pid], readNes
   }
   QMutex_unlock(this.mutex);
}

inline QReadWriteLock_lockForWrite(this) {
   // atomic lock
   QMutex_lock(this.mutex);
   printf("%d: want writelock (this thread nested %d deep). (%d/%d) (%d/%d)\n", _pid, this.count[_pid], readNest
   if
   :: this.currentWriter != _pid ->
      do
      :: (this.threadCount != 0) ->
         this.waitingWriters++;
         printf("%d: wait for writelock (this thread nested %d deep)\n", _pid, this.count[_pid]);
         QWaitCondition_wait(this.writerWait, this.mutex);
         printf("%d: continue for writelock (this thread nested %d deep)\n", _pid, this.count[_pid]);
         this.waitingWriters--;
      :: this.threadCount == 0 ->
         break;
      od;
      this.currentWriter = _pid;
      this.threadCount++;
   :: else
   fi;
   assert(this.threadCount == 1);
   assert(this.currentWriter == _pid);
   this.count[_pid]++;
   // atomic unlock
   atomic {
      writeNest++;
      if
      :: writeNest == 1 -> writers++;
      :: else
      fi;
      printf("%d: has writelock (this thread nested %d deep) (%d/%d) (%d/%d)\n", _pid, this.count[_p
   }
   QMutex_unlock(this.mutex);
}

inline QReadWriteLock_unlock(this) {
   // atomic lock
   QMutex_lock(this.mutex);
   printf("%d: going to release lock (this thread nested %d deep) (%d/%d) (%d/%d)\n", _pid, this.cou
   this.count[_pid]--;
   if
   :: this.count[_pid] == 0 ->
      this.threadCount--;
      if
      :: this.threadCount == 0 ->
         this.currentWriter = NT;
         if
         :: this.waitingWriters > 0 ->
            // wake one writer
            printf("%d: wake one writer\n", _pid);
            QWaitCondition_wakeOne(this.writerWait, this.mutex);
         :: else ->
            if
            :: this.waitingReaders > 0 ->
               // wake all readers
               printf("%d: wake all readers\n", _pid);
               QWaitCondition_wakeAll(this.readerWait, this.mutex);
            :: else
            fi;
         fi;
      :: else
      fi;
   :: else
   fi;

   // atomic unlock
   atomic {
      if
      :: readNest > 0 ->
         readNest--;
         if
         :: readNest == 0 && writeNest == 0 ->
            readers = readers - 1;
         :: else
         fi;
      :: readNest == 0 && writeNest > 0 ->
         writeNest--;
         if
         :: writeNest == 0 -> writers--;
         :: else
         fi;
      fi;
      printf("%d: released lock (this thread nested %d deep) (%d/%d) (%d/%d)\n", _pid, this.count[_p
   }
   QMutex_unlock(this.mutex);
}

#endif
```

A.16 Usage of QReadWriteLock - Qt 4.3

```
#include QREADWRITELOCK_MODEL

QReadWriteLock rwlock;
int readers = 0;
int writers = 0;

active[THREADS] proctype user() {
   pid p1;
   byte maxLocks;
   byte nest = 0;
   do
   :: maxLocks = MAXLOCKS;
      if
      :: do
         :: maxLocks > 0 ->
            nest++;
            QReadWriteLock_lockForWrite(rwlock);
         :: nest > 0 ->
            nest--;
            QReadWriteLock_unlock(rwlock);
         :: maxLocks != MAXLOCKS && nest == 0 -> break;
         od;
      :: do
         :: maxLocks > 0 ->
            nest++;
            QReadWriteLock_lockForWrite(rwlock);
         :: nest > 0 ->
            nest--;
            QReadWriteLock_unlock(rwlock);
         :: maxLocks != MAXLOCKS && nest == 0 -> break;
         od;
      fi;
   od;
}

proctype compactUser() {
   pid p1;
   int readNest = 0;
   int writeNest = 0;
   do
   :: writeNest + readNest < MAX_NEST ->
      QReadWriteLock_lockForRead(rwlock);
      readNest++;
   :: writeNest + readNest < MAX_NEST && readNest == 0 ->
      QReadWriteLock_lockForWrite(rwlock);
      writeNest++;
   :: writeNest + readNest > 0 ->
      QReadWriteLock_unlock(rwlock);
      if
      :: readNest > 0 -> readNest--;
      :: else ->         writeNest--;
      fi;
   od;
}
```

A.17 Usage of QReadWriteLock - deadlock free / Qt 4.4

```
//#include MODEL

#include QREADWRITELOCK_MODEL

QReadWriteLock rwlock;
int readers = 0;
int writers = 0;

#define areReaders (readers > 0)
#define areWriters (writers > 0)
#define noReaders (readers == 0)
#define noWriters (writers == 0)
#define maxOneWriter (writers <= 1)
//#define active (rwlock.mutex.m.lockedBy != NT)
//#define notActive (rwlock.mutex.m.lockedBy == NT)
//#define isActive (pthread_mutex_locked(rwlock.mutex.mutex))
#define notActiveExtended (!(qmutex_busy(rwlock.mutex) || QWaitCondition_busy(rwlock.readerWait) || QWaitConditi
//#define notActive (!(pthread_mutex_locked(rwlock.mutex.mutex)))
//#define notActive (!(QMutex_busy(rwlock.mutex)))
#define notActive (!(qmutex_busy(rwlock.mutex)))
// || QWaitCondition_busy(rwlock.readerWait) || QWaitCondition_busy(rwlock.writerWait)))
#define readersWaiting (rwlock.waitingReaders > 0)
#define writersWaiting (rwlock.waitingWriters > 0)

#define threadCountCorrect (readers + writers == rwlock.threadCount)

#ifdef PROGRESS
active[1] proctype userWriter()
{
   byte level;
   pid p1;
   byte readNest = 0;
   byte writeNest = 0;
progress:
   QReadWriteLock_lockForWrite(rwlock);
   QReadWriteLock_unlock(rwlock);
   goto progress;
}
#endif

inline user() {
   byte level;
   pid p1;
   byte readNest = 0;
   byte writeNest = 0;
   byte maxLocks;
   do
   :: maxLocks = MAXLOCKS;
      do
      :: maxLocks > 0 ->
         maxLocks--;
         if
         :: readNest == 0 -> QReadWriteLock_lockForWrite(rwlock);
         :: QReadWriteLock_lockForRead(rwlock);
         fi;
      :: writeNest + readNest > 0 ->
         QReadWriteLock_unlock(rwlock);
      :: MAXLOCKS != maxLocks && writeNest + readNest == 0 ->
         break;
      od;
   od;
}

proctype userProc()
{
   user();
}

init {
#ifdef PROGRESS
   int i = THREADS-1;
#else
   int i = THREADS;
#endif
   assert(i >= 1);
   do
   :: i > 1 -> i--; run userProc();
   :: i == 1 -> i--; user();
   :: else -> break;
   od;
}
```

Appendix B

Batch config files

The batch tool used in this thesis is specifically designed to support the work in this thesis. The tool is available alongside the models. Yet the tool has a more general application. The config files are easy understandable, and are listed below.

B.1 Stack Example

```
iterate-int STACK_SIZE 1 12 3
iterate-int CONSUMERS 1 8 3
iterate-int PRODUCERS 1 8 3
#define PRODUCERS 2
#define CONSUMERS 2
#define THREADS 4
#define STACK_SIZE 2

define   SCHEDULAR_MODEL      \"schedular.abs\"
define   PTHREAD_COND_MODEL   \"pthread_cond.abs\"
define   PTHREAD_MUTEX_MODEL  \"pthread_mutex.basic.abs\"
#define  SPURIOUS_WAKEUPS

model stack.spin

#iterative

#depth   200000000
depth 10000000
#depth 15000
#memory  129204
memory 3000
search   depth-first
#multicore 16

automaton 20

basic

logfile batch-stack-log
```

B.2 Checking the Qt 4.3 version of QReadWriteLock

```
#define OTHERTHREADS 1
#int MAXLOCKS 1 10

#iterate-int THREADS 2 4
#iterate-int MAX_NEST 1 10 3
```

```
define MAXLOCKS 2
define MAX_NEST 2
define THREADS 2

define QREADWRITELOCK_MODEL \"qreadwritelock43.spin\"
define QWAITCONDITION_MODEL \"qwaitcondition.pml\"
define PTHREAD_COND_MODEL \"pthread_cond.abs\"

define QMUTEX_MODEL \"qmutex.abs\"
define PTHREAD_MUTEX_MODEL \"pthread_mutex.basic.abs\"

define SCHEDULAR_MODEL \"schedular.abs\"

model qreadwritelock44.usage

iterative

#depth   200000000
#memory  129204
depth 1000
memory 3072
search  depth-first

basic
#progress
#safety ([] (noReaders || noWriters)) && ([] maxOneWriter) && ([] notBusy -> threadCountCorrect)
```

B.3 Checking the Qt 4.4 version of QReadWriteLock

```
#iterate-int THREADS 2 4
#define OTHERTHREADS 1
#define MAXLOCKS 1
#iterate-int MAXLOCKS 1 9 2
#int MAX_NEST 1 10

#define THREADS 3
#define MAXLOCKS 1

iterate-int MAXLOCKS 1 7 2
iterate-int THREADS 2 4

define   QREADWRITELOCK_MODEL \"qreadwritelock.spin\"
define   QWAITCONDITION_MODEL \"qwaitcondition.spin\"
define   QMUTEX_MODEL      \"qmutex.abs\"

define   PTHREAD_COND_MODEL   \"pthread_cond.abs\"
define   PTHREAD_MUTEX_MODEL  \"pthread_mutex.basic.abs\"

#define  PTHREAD_COND_MODEL   \"pthread_cond.fair2.abs\"
#define  PTHREAD_MUTEX_MODEL  \"pthread_mutex.basic.fair3.abs\"
#define  SCHEDULAR_MODEL      \"schedular.abs\"

#define  QSEMAPHORE_MODEL  \"qsemaphore.abs\"

model qreadwritelock44.usage

#iterative

depth 200000000
memory   129204
#memory 3072
#depth 4000000
#depth 130
search   depth-first
#multicore 16

basic
#safety ([] (noReaders || noWriters)) && ([] maxOneWriter) && ([] notActiveExtended -> threadCountCo
safety ([] ((noReaders || noWriters) && (maxOneWriter) && (notActiveExtended -> threadCountCorrect)
#progress

logfile batch44-qwait-log
```

B.4 Checking the starvation-free version of QReadWriteLock

B.4.1 Verifying assertions, safety properties and absence of deadlocks

```
#define OTHERTHREADS 1
#iterate-int MAXLOCKS 1 7 2
iterate-int MAXLOCKS 1 2
iterate-int THREADS 2 4
#int MAX_NEST 1 10
#define THREADS 3
#define MAXLOCKS 1

define   QREADWRITELOCK_MODEL \"qreadwritelock.spin\"
define   QWAITCONDITION_MODEL \"qwaitcondition.fair.alt4.spin\"
define   QMUTEX_MODEL   \"qmutex.fair.alt4.spin\"

#define  QWAITCONDITION_MODEL \"qwaitcondition.pml\"
#define  QMUTEX_MODEL   \"qmutex.pml\"

#define  PTHREAD_COND_MODEL   \"pthread_cond.fair2.abs\"
define   PTHREAD_COND_MODEL   \"pthread_cond.abs\"
#define  PTHREAD_MUTEX_MODEL  \"pthread_mutex.basic.fair3.abs\"
define   PTHREAD_MUTEX_MODEL  \"pthread_mutex.basic.fair.abs\"

define   SCHEDULAR_MODEL   \"schedular.abs\"

define   QSEMAPHORE_MODEL  \"qsemaphore.abs\"

#model   fairreadwritelock.usage
model qreadwritelock44.usage

#iterative

#depth   200000000
depth 15000000
#depth 500
#memory  129204
memory 3072
search   depth-first
#multicore 16

basic
#define PROGRESS
#progress
#safety ([] (noReaders || noWriters)) && ([] maxOneWriter) && ([] (notActive -> threadCountCorrect)) && ([] (not

##safety ([] ((noReaders || noWriters) && (maxOneWriter) && (notActive -> threadCountCorrect) && (notActive -> (

#safety ([] (noReaders || noWriters))
#safety ([] (maxOneWriter))
#safety ([] (notActive -> threadCountCorrect))
#safety ([] (notActive -> (writersWaiting -> (areReaders || areWriters))))
#safety ([] (notActive -> (readersWaiting -> (areWriters || writersWaiting))))

logfile batch-fair-log
```

B.4.2 Verifying absence of starvation

```
#define OTHERTHREADS 1
#iterate-int MAXLOCKS 1 3
#iterate-int THREADS 2 4
#int MAX_NEST 1 10
define THREADS 3
define MAXLOCKS 1

define   QREADWRITELOCK_MODEL \"qreadwritelock.spin\"
define   QWAITCONDITION_MODEL \"qwaitcondition.fair.alt4.spin\"
define   QMUTEX_MODEL   \"qmutex.fair.alt4.spin\"
```

```
#define  QWAITCONDITION_MODEL \"qwaitcondition.pml\"
#define  QMUTEX_MODEL   \"qmutex.pml\"

#define  PTHREAD_COND_MODEL   \"pthread_cond.fair2.abs\"
define   PTHREAD_COND_MODEL   \"pthread_cond.abs\"
#define  PTHREAD_MUTEX_MODEL  \"pthread_mutex.basic.fair3.abs\"
define   PTHREAD_MUTEX_MODEL  \"pthread_mutex.basic.fair.abs\"

define   SCHEDULAR_MODEL   \"schedular.abs\"

define   QSEMAPHORE_MODEL  \"qsemaphore.abs\"

#model   fairreadwritelock.usage
model qreadwritelock44.usage

#iterative

#depth   200000000
depth 10000000
#depth 1250
#memory  129204
memory 3000
search   depth-first
#multicore 16

#basic

#safety ([] (noReaders || noWriters)) && ([] maxOneWriter) && ([] (notActive -> threadCountCorrect))
#safety ([] ((noReaders || noWriters) && (maxOneWriter) && (notActive -> threadCountCorrect) && (not

define PROGRESS
progress

logfile batch-fair-progress-log
```

Bibliography

Academic References

[1] *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS 2009)*. IEEE, July 2009.

[2] R. Alur and D. Peled, editors. *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[3] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In Grumberg and Huth [20], pages 134–138.

[4] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In Alur and Peled [2], pages 484–487.

[5] H. R. Arabnia and H. Reza, editors. *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 2*. CSREA Press, 2006.

[6] F. M. auf der Heide and M. A. Bender, editors. *SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallel Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009*. ACM, 2009.

[7] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. MIT Press, 2008.

[8] J. Barnat, L. Brim, I. Cerná, P. Moravec, and P. Rockai. DiVinE – a tool for distributed verification. In T. Ball and R. B. Jones, editors, *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 278–281. Springer-Verlag, August 2006.

[9] J. Barnat, V. Forejt, M. Leucker, and M. Weber. Michael. DivSpin- A Spin compatible distributed model checker. In *Proceedings of 4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC05)*, pages 95–100. TU Munchen Further information.

[10] M. Ben-Ari. *Principles of the* Spin *Model Checker*. Springer-Verlag, 2008.

[11] B. Brandenburg and J. Anderson. Reader-writer synchronization for shared-memory multiprocessor real-time systems. [1], pages 184–193.

[12] P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1975.

[13] D. D. Cofer and A. Fantechi, editors. *Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers*, volume 5596 of *Lecture Notes in Computer Science*. Springer, 2009.

[14] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.

[15] E. Dijkstra. Cooperating Sequential Processes, Technical Report EWD-123. *Technological University, Eindhoven*, 1965.

[16] B. van Gastel. Reliability of a read-write lock implementation, Bachelor's thesis, Radboud Universiteit, Nijmegen, the Netherlands, February 2008.

[17] B. van Gastel, L. Lensink, S. Smetsers, and M. C. J. D. van Eekelen. Reentrant readers-writers: A case study combining model checking with theorem proving. In Cofer and Fantechi [13], pages 85–102.

[18] B. van Gastel, L. Lensink, S. Smetsers, and M. C. J. D. van Eekelen. Deadlock and starvation free reentrant readers-writers: a case study combining model checking with theorem proving. *Science of Computer Programming*, 2010. Under Consideration.

[19] P. Godefroid, editor. *Model Checking Software: 12th International* SPIN *Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings*, volume 3639 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2005.

[20] O. Grumberg and M. Huth, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[21] K. Havelund, R. Majumdar, and J. Palsberg, editors. *Model Checking Software, 15th International* SPIN *Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings*, volume 5156 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[22] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *STTT*, 2(4):366–381, 2000.

[23] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[24] G. Holzmann. *The* SPIN *Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, USA, 2004.

[25] G. J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[26] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[27] Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. In auf der Heide and Bender [6], pages 101–110.

[28] E. Mercer and M. Jones. Model checking machine code with the GNU debugger. In Godefroid [19], pages 251–265.

[29] V. Pantelic, X.-H. Jin, M. Lawford, and D. L. Parnas. Inspection of concurrent systems: Combining tables, theorem proving and model checking. In Arabnia and Reza [5], pages 629–635.

[30] B. Schlich and S. Kowalewski. Model checking c source code for embedded systems. *STTT*, 11(3):187–202, 2009.

[31] E. W. Stark. Semaphore primitives and starvation-free mutual exclusion. *Journal of the ACM*, 29(4):1049–1072, 1982.

[32] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *International Journal on Automated Software Engineering*, 10(2):203–232, April 2003.

[33] A. Zaks and R. Joshi. Verifying multi-threaded C programs with Spin. In Havelund et al. [21], pages 325–342.

Non-Academic References

[34] U. Drepper. Futexes are tricky. *Red Hat Inc*, August 2009.

[35] U. Drepper and I. Molnar. The native POSIX thread library for Linux. *White Paper, Red Hat*, 2003.

[36] T. Fletcher and C. Burgess. A condvar is not a semaphore, March 2007. `http://sendreceivereply.wordpress.com/2007/03/13/4/`.

[37] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium*, pages 479–489. Citeseer, 2002.

[38] G. Holzmann. Man page of `atomic`, November 2004. `http://spinroot.com/spin/Man/atomic.html`.

[39] G. Holzmann. Man page of `else`, November 2004. `http://spinroot.com/spin/Man/else.html`.

[40] A. D. Marshall. Programming in C: Unix system calls and subroutines using C, 2005. `http://www.cs.cf.ac.uk/Dave/C/`.

[41] Nokia. Desktop applications powered by Qt, February 2010. `http://qt.nokia.com/qt-in-use/story/qt-in-use/target/desktop`.

[42] C.-K. Shene. Threadmentor: The dining philosophers problem, September 2005. http://www.cs.mtu.edu/~shene/NSF-3/e-Book/MUTEX/TM-example-philos-1.html.

[43] Trolltech. QWaitCondition class reference, 2008. http://doc.trolltech.com/4.3/qwaitcondition.html.

[44] Wikipedia. Dining philosophers problem, March 2010. http://en.wikipedia.org/wiki/Dining_philosophers_problem.