Reentrant Readers-Writers
– a Case Study Combining Model Checking with Theorem Proving –

Bernard van Gastel, Leonard Lensink, Sjaak Smetsers and Marko van Eekelen

Institute for Computing and Information Sciences, Radboud University Nijmegen
Heyendaalseweg 135, 6525 AJ, Nijmegen, The Netherlands
email: b.vangastel@student.science.ru.nl,{l.lensink,s.smetsers,m.vaneekelen}@cs.ru.nl

Abstract. The classic readers-writers problem has been extensively studied. This holds to a lesser degree for the reentrant version, where it is allowed to nest locking actions. Such nesting is useful when a library is created with various procedures that each start and end with a lock. Allowing nesting makes it possible for these procedures to call each other. We considered an existing widely used industrial implementation of the reentrant readers-writers problem. We modeled it using a model checker revealing a serious error: a possible deadlock situation. The model was improved and checked satisfactorily for a fixed number of processes. To achieve a correctness result for an arbitrary number of processes the model was converted to a theorem prover with which it was proven.

1 Introduction

It is generally acknowledged that the growth in processor speed is reaching a hard physical limitation. This has led to a revival of interest in concurrent processing. Also in industrial software, concurrency is increasingly used to improve efficiency [26]. It is notoriously hard to write correct concurrent software. Finding bugs in concurrent software and proving the correctness of (parts of) this software is therefore attracting more and more attention, in particular where the software is in the core of safety critical or industrial critical applications.

However, it can be incredibly difficult to track concurrent software bugs down. In concurrent software bugs typically are caused by infrequent 'race conditions' that are hard to reproduce. In such cases, it is necessary to thoroughly investigate 'suspicious' parts of the system in order to improve these components in such a way that correctness is guaranteed.

Two commonly used techniques for checking correctness of such system are *formal verification* and *testing*. In practice, testing is widely and successfully used to discover faulty behavior, but it cannot assure the absence of bugs. In particular, for concurrent software testing is less suited due to the typical characteristics of the bugs (infrequent and hard to reproduce). There are roughly two approaches to formal verification: *model checking* and *theorem proving*. Model checking [6, 23] has the advantage that it can be performed automatically, provided that a suitable model of the software (or hardware) component has been created. Furthermore, in the case a bug is found model checking yields a counterexample scenario. A drawback of model checking is that it suffers from the

state-space explosion and typically requires a closed system. In principle, theorem proving can handle any system. However, creating a proof may be hard and it generally requires a large investment of time. It is only partially automated and mainly driven by the user's understanding of the system. Besides, when theorem proving fails this does not necessarily imply that a bug is present. It may also be that the proof could not be found by the user.

In this paper we consider the *reentrant readers-writers* problem as a formal verification case study. The classic readers-writers problem [8] considers multiple processes that want to have read and/or write access to a common resource (a global variable or a shared object). The problem is to set up an access protocol such that no two writers are writing at the same time and no reader is accessing the common resource while a writer is accessing it. The classic problem is studied extensively[22]; the reentrant variant (in which locking can be nested) has received less attention so far although it is used in Java, C# and C++ libraries.

We have chosen a widely used industrial library (Trolltech's Qt) that provides methods for reentrant readers-writers. For this library a serious bug is revealed and removed. This case study is performed in a structured manner combining the use of a model checker with the use of a theorem prover exploiting the advantages of these methods and avoiding their weaknesses.

In Section 2 we will introduce the case study. Its model will be defined, improved and checked for a fixed number of processes in Section 3. Using a theorem prover the model will be fully verified in Section 4. Finally, related work, future work and concluding remarks are found in Sections 5 and 6.

2 The readers-writers problem

If in a concurrent setting two threads are working on the same resource, synchronization of operations is often necessary to avoid errors. A *test-and-set* operation is an important primitive for protecting common resources. This atomic (i.e. non-interruptible) instruction is used to both test and (conditionally) write to a memory location. To ensure that only one thread is able to access a resource at a given time, these processes usually share a global boolean variable that is controlled via *test-and-set* operations, and if a process is currently performing a test-and-set, it is guaranteed that no other process may begin another test-and-set until the first process is done. This primitive operation can be used to implement *locks*. A lock has two operations: lock and unlock. The lock operation is done before the critical section is entered, and the unlock operation is performed after the critical section is left. The most basic lock can only be locked one time by a given thread. However, for more sophisticated solutions, just an atomic test-and-set operation is insufficient. This will require support of the underlying OS: threads acquiring a lock already occupied by some thread should be de-scheduled until the lock is released. A variant of this way of locking is called *condition locking*: a thread can wait until a certain condition is satisfied, and will automatically continue when notified (*signalled*) that the condition has been changed. An extension for both basic and condition locking is *reentrancy*, i.e. allowing nested lock operations by the same thread.

A so-called *read-write* lock functions differently from a normal lock: it either allows multiple threads to access the resource in a read-only way, or it allows one, and only one, thread at any given time to have full access (both read and write) to the resource ([10]). These locks are the standard solution to the producer/consumer problem in which a buffer has to be shared.

Several kinds of solutions to the classical readers-writers problem exist. Here, we will consider a *read-write* locking mechanism with the following properties.

writers preference Read-write locks suffer from two kinds of starvation, one with each kind of lock operation. Write lock priority results in the possibility of reader starvation: when constantly there is a thread waiting to acquire a write lock, threads waiting for a read lock will never be able to proceed. Most solutions give priority to write locks over read locks because write locks are assumed to be more important, smaller, exclusive, and to occur less.

reentrant A thread can acquire the lock multiple times, even when the thread has not fully released the lock. Note that this property is important for modular programming: a function holding a lock can use other functions which possibly acquire the same lock. We distinguish two variants of reentrancy:

1. *Weakly reentrant*: only permit sequences of either read or write locks;
2. *Strongly reentrant*: permit a thread holding a write lock to acquire a read lock. This will allow the following sequence of lock operations: write_lock, read_lock, unlock, unlock. Note that the same function is called to unlock both a write lock and a read lock. The sequence of a read lock followed by a write lock is not admitted because of the evident risk of a deadlock (e.g. when two threads both want to perform the locking sequence read_lock, write_lock they can both read but none of them can write).

2.1 Implementation of Read-Write locks

In this section we show the C++ implementation of weakly reentrant read/write locks being part of the multi-threading library of the Qt development framework, version 4.3. The code is not complete; parts that are not relevant to this presentation are omitted. This implementation uses other parts of the library: threads, mutexes and conditions. Like e.g. in Java, a `condition` object allows a thread that owns the lock but that cannot proceed, to wait until some condition is satisfied. When a running thread completes a task and determines that a waiting thread can now continue, it can call a signal on the corresponding condition. This mechanism is used in the C++ code listed in Figure 1.

The structure `QReadWriteLockPrivate` contains the attributes of the class. These attributes are accessible via an indirection named `d`. The attributes `mutex`, `readerWait` and `writerWait` are used to synchronize access to the other administrative attributes, of which `accessCount` keeps track of the number of locks (including reentrant locks) acquired for this lock. A negative value is used for write access and a positive value for read access. The attributes `waitingReaders` and `waitingWriters` indicate the number of threads requesting a read respectively write permission, that are currently pending. If some thread owns the write lock,

```
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();
    }
   }
}
```

Fig. 1. `QReadWriteLock` class of Qt

`currentWriter` contains a `HANDLE` to this thread; otherwise `currentWriter` is a null pointer.

The code itself is fairly straightforward. The locking of the `mutex` is done via the constructor of the wrapper class `QMutexLocker`. Unlocking this mutex happens implicitly in the destructor of this wrapper. Observe that 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`).

The code could be polished a bit. E.g. one of the administrative attributes can be expressed in terms of the others. However, we have chosen not to deviate from the original code, except for the messages in the assertions which were, of course, more informative.

3 Model checking readers/writers with Uppaal

Uppaal [17] is a tool for modeling and verification of real-time systems. The idea is to model a system using timed automata. Timed automata are finite state machines with time. A system consists of a collection of such automata. An automaton is composed of locations and transitions between these locations defining how the system behaves. To control when to fire a transition one can use guarded transitions and synchronized transitions. Guards are just boolean expressions whereas the synchronization mechanism is based on hand-shakes: two processes (automata) can take a simultaneous transition, if one does a send, `ch!`, and the other a receive, `ch?`, on the same channel `ch`. For administration purposes, but also for communication between processes, one can use global variables. Moreover, each process can have its own local variables. Assignments to local or global variables can be attached to transitions as so-called *updates*.

In this paper we will not make use of time. In Uppaal terminology: we don't have `clock` variables. Despite the absence of this most distinctive feature of Uppaal, we have still chosen to use Uppaal here because of our local expertise and the intuitive and easy to use graphical interface which supports understanding and improving the model in a elegant way. The choice of model checker is however not essential for the case study. It could also have been performed with any other model checker such as e.g. SMV [19], mCRL2 [11] or SPIN [14].

Constructing the Uppaal model

Our intention is to model the code from Figure 1 as an abstract Uppaal model, preferably in a way that the distance between code and model is kept as small as possible. However, instead of trying to model Qt-threads in Uppaal we will directly use the built-in Uppaal processes to represent these threads. Thread handles are made explicit by numbering the processes, and using these numbers as identifications. `NT` is the total number of processes. The identification numbers are denoted by `tid` in the model, ranging 0 to `NT` - 1. The `NT` value is also used to represent the null pointer for the variable `currentWriter` in the C++ code. Mutexes and conditions directly depend on the thread implementation, so we cannot model these objects by means of code abstraction. Instead we created an abstract model in Uppaal that essentially simulates the behavior of these objects. The result is shown in Figure 2. In this basic locking model, method calls are simulated via synchronization messages. The conditions are represented by two integer variables, `sleepingReaders` and `sleepingWriters`, that maintain the number of waiting readers and waiting writers, respectively. A running process can signal such a process which will result in a wake up message. A process receiving such a message should always immediately try to acquire the lock, otherwise mutual exclusion is not guaranteed anymore.

The `RWLock` implementation is model checked using the combination of this basic locking process with a collection of concurrent processes, each continuously performing either a `lockForRead`, `lockForWrite`, or `unlock` step. The abstract model (see Figure 3) is obtained basically by translating C++ statements into transitions.

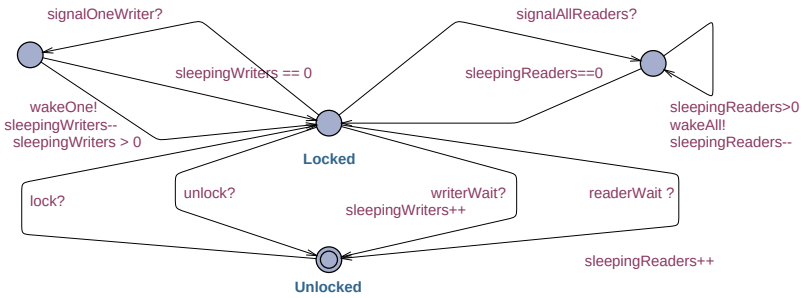

Fig. 2. Mutex and condition model

For convenience of comparison, we have split the model into three parts, corresponding to `lockForRead`, `writeLock` and `unlock` respectively. These parts can be easily combined into a single model by collapsing the `Start` states, and, but not necessarily, the `Abort` states. The auxiliary functions `testRLock`, `testWLock`, and `testReentrantWLock` are defined as:

```
bool testRLock(ThreadId tid)
{ return waitingWriters>0 ||(currentWriter!=NT && currentWriter!=tid);}
```

```
bool testWLock (ThreadId tid)
{  return accessCount != 0 &&
          currentWriter != tid;
}
```

```
bool testReentrantWLock (ThreadId tid)
{  return accessCount != 0 &&
          tid == currentWriter;
}
```

If a process performs a lock operation it will enter a location that is labeled with `EnterXX`. Here, `XX` corresponds to the called operation. The call is left via a `LeaveXX` location. For example, if a thread invokes `lockForRead` it will enter the location `EnterRL`. Hereafter, the possible state transitions directly reflect the corresponding flow of control in the original code for this method. The call ends at `LeaveRL`. These special locations are introduced to have a kind of separation between definition and usage of methods. If the thread was suspended (due to a call to the `wait` method on the `readerWait` condition) the process in the abstract model will be waiting in the location `RWait`. The wrapper `QMutexLocker` has been replaced by a call to `lock`. To take the effect of the destructor into account, we added a call to unlock at the end of the scope of the wrapper object. Furthermore, observe that assertions are modeled as a 'black hole': a state, labeled `Abort`, from which there is no escape possible.

Checking the model

The main purpose of a model checker is to verify the model w.r.t. a requirement specification. In Uppaal, requirements are specified as *queries* consisting of path and state formulae. The latter describe individual states whereas the former range over execution paths or traces of the model. In Uppaal, the (state) formula `A[]` φ expresses that φ should be true in all reachable states. `deadlock` is a built-in formula which is true if the state has no outgoing edges.

In our example we want to verify that the model is deadlock-free, which is a state property. This can easily be expressed by means of the following query:

```
A[] not deadlock
```

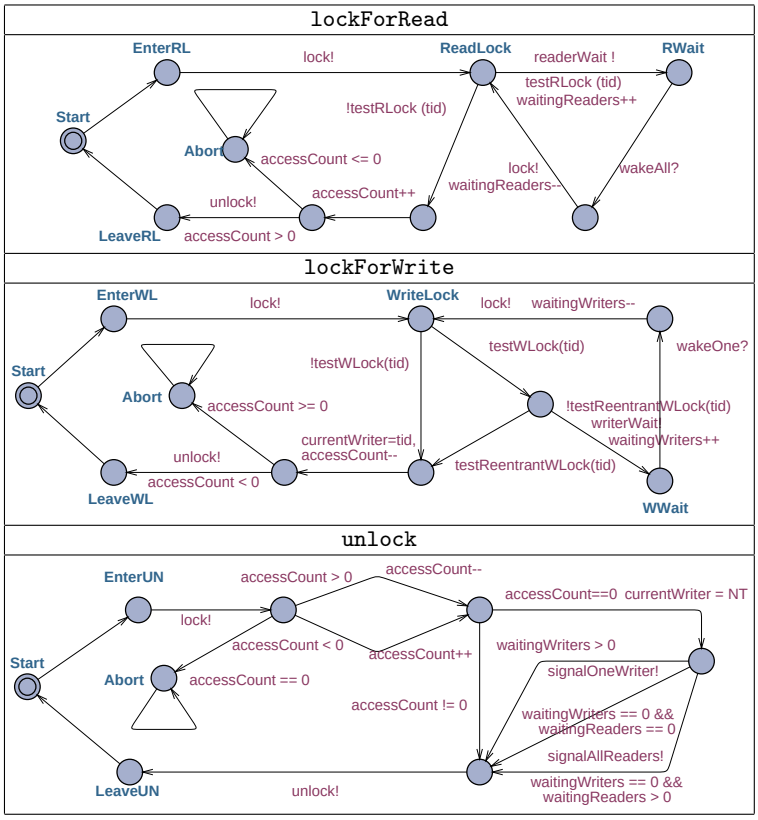

Fig. 3. Uppaal models of the locking primitives

When running Uppaal on this model consisting of 2 threads, the verifier will almost instantly respond with: `Property is not satisfied`. The trace generated by Uppaal shows a counter example of the property, in this case a scenario leading to a deadlock. The problem is that if a thread, which is already holding a read lock, does a (reentrant) request for another read lock, it will be suspended if another thread is pending for a write lock (which is the case if the write lock was requested after the first thread obtained the lock for the first time). Now both threads are waiting for each other.

3.1 Correcting the implementation/model

The solution is to let a reentrant lock attempt always succeed. To avoid writers starvation, new read lock requests should be accepted only if there are no writers waiting for the lock. To distinguish non-reentrant and reentrant uses, we maintain, per thread, the current number of nested locks making no distinction between read and write locks. Additionally, this solution allows strongly reentrant use. In the implementation this is achieved by adding a *hash map* (named `current` of type `QHash`) to the attributes of the class that maps each thread handle to a counter. To illustrate our adjustments, we show the implementation of `lockForRead` [1].

[1] For the complete code, see www.cs.ru.nl/~sjakie/papers/readerswriters/.

```
void QReadWriteLock::lockForRead() {
    QMutexLocker lock(&d->mutex);

    Qt::HANDLE self = QThread::currentThreadId();

    QHash<Qt::HANDLE, int>::iterator it = d->current.find(self);
    if (it != d->current.end()) {
         ++it.value();
         Q_ASSERT_X(d->numberOfThreads > 0, "...", "...");
        return;
    }
    while (d->currentWriter != 0 || d->waitingWriters > 0) {
        ++d->waitingReaders;
        d->readerWait.wait(&d->mutex);
        --d->waitingReaders;
    }
    d->current.insert(self, 1);
    ++d->numberOfThreads;
    Q_ASSERT_X(d->numberOfThreads> 0, "...", "...");
}
```

To verify this implementation we again converted the code to Uppaal. Since handles where represented by integers ranging from 0 to `NT - 1` (where `NT` denotes the number of threads), we can use a simple integer array to maintain the number of nested locks per thread, instead of a hash map. In this array, the process id is used as an index. Figure 4 shows the part of the Uppaal model that corresponds to the improved `lockForRead`. For the full Uppaal model, see www.cs.ru.nl/∼sjakie/papers/readerswriters/.

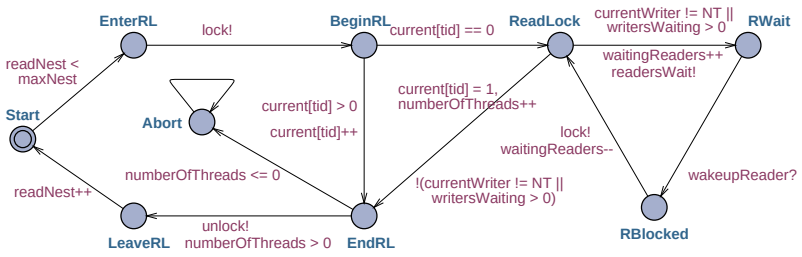

Fig. 4. Uppaal model of the correct version of `lockForRead`

To limit the state space we have added an upper bound `maxNest` to the nesting level and a counter `readNest` indicating the current nesting level. This variable is decremented in the unlock part of the full model. Running Uppaal on the improved model will, not surprisingly, result in the message: `Property is satisfied`. In this experiment we have limited the number of processes to 4, and the maximum number of reentrant calls to 5. If we increase these values slightly, the execution time worsens drastically. So, for a complete correctness result, we have to proceed differently.

4 General reentrant readers-writers model

In this section we will formalize the Uppaal model in PVS [21].

We prove that the reentrant algorithm is free from deadlock when we generalize to *any* number of processes. While explaining the formalization we will briefly introduce PVS. For the complete PVS specification, see www.cs.ru.nl/~sjakie/papers/readerswriters/.

4.1 Readers-Writers model in PVS

PVS offers an interactive environment for the development and analysis of formal specifications. The system consists of a specification language and a theorem prover. The specification language of PVS is based on classic, typed higher-order logic. It resembles common functional programming languages, such as Haskell, LISP or ML. The choice of PVS as the theorem prover to model the readers writers locking algorithm is purely based upon the presence of local expertise. The proof can be reconstructed in any reasonably modern theorem prover, for instance Isabelle [20] or Coq[5]. There is no implicit notion of state in PVS specifications. So, we explicitly keep track of a system state that basically consists of the system variables used in the Uppaal model.

In the Uppaal model a critical section starts with a `lock!` and ends with either a `unlock!`, `readersWait!` or `writersWait!` synchronization. Not all the state transitions are modelled individually in the PVS model. All actions occuring inside a critical section are modeled as a single transition. This makes the locking mechanism protecting the critical sections superfluous in the PVS model and enables us to reduce the number of different locations. Only these locations in the Uppaal model that are outside a critical section are needed and are tracked by the `ThreadLocation` variable. Furthermore, the `EnterXX` and `LeaveXX` locations are ignored, because they are only used as a label for a function call and have no influence on the behavior of the modeled processes.

With `NT` denoting the total number of processes, we get the following representation:

```
ThreadID       : TYPE = below(NT)[2]
ThreadLocation : TYPE = { START, RWAIT, RBLOCKED, WWAIT, WBLOCKED }
ThreadInfo     : TYPE = [# status : ThreadLocation, current : nat #][3]

System         : TYPE = [# waitingWriters, waitingReaders,
                           numberOfThreads : nat,
                           currentWriter   : below(NT+1),
                           threads : ARRAY [ThreadID → ThreadInfo] #][4]
```

The auxiliary variables `readNest`, `writeNest` and `maxNest` restrict the Uppaal model to a maximum number of nested reads and writes. They also prevent unwanted sequences of lock/unlock operations, e.g. when a write lock request occurs after a read lock has already been obtained. In the PVS model we allow for any amount of nesting, so the variables `writeNest` and `maxNest` introduced to limit nesting can be discarded. The `readNest` variable is used to check whether there already is a read lock present when a write lock is requested. In the PVS model we have implemented this check by testing whether the lock counter for this particular thread is 0 before it starts waiting for a (non-reentrant) write lock. The logic behind it is that if, previously, a read lock had been obtained by this thread, the counter would have been unequal to 0.

Because none of the variable updates in the Uppaal model occur outside of a critical section, we can model the concurrent execution of the different processes obtaining writelocks, readlocks and releasing them by treating them as interleaved functions.

[2] Denotes the set of natural numbers between 0 and `NT`, exclusive of `NT`.
[3] Recordtypes in PVS are surrounded by `[#` and `#]`.
[4] Arrays in PVS are denoted as functions.

We first define a step function that executes one of the possible actions for a single process. The step function is restricted to operate on a subset of the `System` data type, signified by the `validState?` predicate, further explained in Section 4.3. The actions themselves do not deliver just a new state but a *lifted* state. In PVS, the predefined `lift` datatype, consisting of two constructors `up` and `bottom`, adds a bottom element to a given base type, in our case `validState?` incorporating the state of the model. This is useful for defining partial functions, particularly to indicate the cases that certain actions are not permitted.

In essence the step function corresponds to the center of the Uppaal model consisting of the `Start` and the `EnterXX`/`LeaveXX` states.

```
step(tid:ThreadID, s1, s2: (validState?) ):  bool =
    writelock(s1,tid) = up(s2) ∨ readlock(s1,tid) = up(s2) ∨
        unlock(s1,tid) = up(s2)
```

The predicate `interleave` simulates parallel execution of threads.

```
interleave (s1,s2:System): bool =
    ∃ (tid:ThreadID): step(tid,s1,s2) ∧
        ∀ (other_tid: ThreadID): other_tid ≠ tid ⇒
            s1‘threads(other_tid) = s2‘threads(other_tid)
```
[5]

4.2 Translation from Uppaal to PVS

The functions that perform the readlock, writelock and unlock respectively are essentially the same as in the original code. It is very well possible to derive the code automatically from the Uppaal model by identifying all paths that start with a `lock!` action on its edge and lead to the first edge with an `unlock!`, `readersWait!` or `writersWait!` action. The `readlock` function is provided as an example of this translation. For instance, the round trip in Figure 4 from the `Start` location, through `BeginRL` directly going to `EndRL`, has guard `current[tid] > 0`, and action `current[tid]++;` associated with it. It starts and ends in the `START` location of the PVS model. This can be recognized as a part of the code of the `readlock` function below.

```
readlock(s1:(validState?), tid:ThreadID) : lift[(validState?)]  =
LET thread = s1‘threads(tid) IN
    CASES thread‘status OF
    START:
        IF thread‘current > 0
        THEN up(s1 WITH [threads := s1‘threads WITH
                 [tid := thread WITH [current := thread‘current+1]]])
        ELSIF s1‘currentWriter ≠ NT ∨ s1‘waitingWriters > 0
        THEN up(s1 WITH [waitingReaders := s1‘waitingReaders + 1,
                 threads := s1‘threads WITH
                 [tid := thread WITH [status := RWAIT]]])
        ELSE up(s1 WITH [ numberOfThreads := s1‘numberOfThreads + 1,
                 threads := s1‘threads WITH
                 [tid := thread WITH [current := 1]]])
        ENDIF,
    RBLOCKED:
```

[5] The ‘ operator denotes record selection.

```
        IF s1`currentWriter ≠ NT ∨ s1`waitingWriters > 0
        THEN up(s1)
        ELSE up(s1 WITH [ numberOfThreads := s1`numberOfThreads + 1,
                 waitingReaders := s1`waitingReaders - 1,
                 threads := s1`threads WITH
                 [tid := thread WITH [current := 1, status := START]]])
                 ENDIF
    ELSE:
        up(s1)
    ENDCASES
```

4.3 System invariants

Not every combination of variables will be reached during normal execution of the program. Auxiliary variables are maintained that keep track of the total amount of processes that are in their critical section and of the number of processes that are waiting for a lock. We express the consistency of the values of those variables by using a `validState?` predicate. This is an invariant on the global state of all the processes and essential in proving that the algorithm is deadlock free. We want to express in this invariant that the global state is sane and safe. Sanity is defined as:

- The value of the `waitingReaders` should be equal to the total number of processes with a status of `RWAIT` or `RBLOCKED`.
- The value of the `waitingWriters` should be equal to the total number of processes with a status of `WWAIT` or `WBLOCKED`.
- The value of the `numberOfThreads` variable should be equal to the number of processes with a lock count of 1 or higher.

Besides the redundant variables having sane values, we also prove that the invariant satisfies that any waiting process has a count of zero current readlocks, stored in the `current` field of `ThreadInfo`. Furthermore, if a process has obtained a write lock, then only that process can be in its critical section:

```
s: VAR System
countInv(s): bool = s`numberOfThreads = count(s`threads)

waitingWritersInv(s): bool = s`waitingWriters  = waitingWriters(s)
waitingReadersInv(s): bool = s`waitingReaders  = waitingReaders(s)

statusInv(s): bool = ∀(tid:ThreadID):
  LET thr = s`threads(tid) IN
     thr`status = WWAIT ∨ thr`status = WBLOCKED ∨
     thr`status = RWAIT ∨ thr`status = RBLOCKED ⇒ thr`current = 0

writeLockedByInv(s) : bool = LET twlb = s`currentWriter IN
  twlb ≠ NT ⇒ s`numberOfThreads = 1 ∧
     s`threads(twlb)`status = START ∧  s`threads(twlb)`current > 0 ∧
     ∀(tid:ThreadID): tid ≠ twlb ⇒ s`threads(tid)`current = 0))

validState?(s) : bool =  countInv(s) ∧ waitingWritersInv(s) ∧
          statusInv(s) ∧ writeLockedByInv(s) ∧ waitingReadersInv(s)
```

Before trying to prove the invariant with PVS, we have first tested the above properties (except for `waitingWritersInv`) and `waitingReadersInv`) in the Uppaal model to see if they hold in the fixed size model (see Figure 5). The properties `waitingWritersInv` and `waitingReadersInv` cannot be expressed in Uppaal because one cannot count the number of processes residing in a specific location. The inspection of the above properties in Uppaal enables us to detect any mistakes in the invariant before spending precious time on trying to prove them in PVS.

- $\texttt{A[]}\,\texttt{countCurrents()} = \texttt{numberOfThreads}$ (Count Inv.) [6]
- $\texttt{A[]}\ \forall \texttt{t} \in \texttt{ThreadId} : \texttt{Thread(t).WWait} \vee \texttt{Thread(t).RWait} \vee \texttt{Thread(t).WBlocked} \vee \texttt{Thread(t).RBlocked} \Rightarrow \texttt{current[t]} = 0$ (Status Inv.)
- $\texttt{A[]}\,\texttt{currentWriter} \neq \texttt{NT} \Rightarrow$ (writeLockedBy Inv.)
  $\texttt{numberOfThreads} = 1 \wedge$
  $\neg\texttt{Thread(currentWriter).writeLockEnd} \Rightarrow \texttt{current[currentWriter]} > 0 \wedge$
  $\forall \texttt{t} \in \texttt{ThreadId} : \texttt{t} \neq \texttt{currentWriter} \Rightarrow \texttt{current[t]} = 0$

Fig. 5. The invariants checked in Uppaal

The definition of the `readlock` function over the dependent type `validState?` implies that automatically type checking conditions are generated. They oblige us to prove that, if we are in a valid state, the transition to another state will yield a state for which the invariant still holds. The proof itself is a straightforward, albeit large (about 400 proof commands), case distinction with the help of some auxiliary lemmas.

4.4 No deadlock

The theorem-prover PVS does not have an innate notion of deadlock. If, however, we consider the state-transition model as a directed graph, in which the edges are determined by the `interleave` function, deadlock can be detected in this state transition graph by identifying a state for which there are no outgoing edges. This interpretation of deadlock can be too limited. If, for example, there is a situation where a process alters one of the state variables in a non terminating loop, the state-transition model will yield an infinite graph and a deadlock will not be detected, because each state has an outgoing edge. Still, all the other processes will not be able to make progress. To obtain a more refined notion of deadlock, we define a well founded ordering on the system state and show that for each state reachable from the starting state (except for the starting state itself), there exists a transition to a smaller state according to that ordering. The smallest element within the order is the starting state. This means that each reachable state has a path back to the starting state and consequently it is impossible for any process to remain in a such a loop indefinitely. Moreover, this also covers the situation in which we would have a *local deadlock* (i.e. several but not all processes are waiting for each other).

```
t : VAR ThreadInfo
starting? : PRED[ThreadInfo] = { t | t`status = START ∧ t`current = 0}

startingState(s: (validState?)): bool =
   ∀(tid:ThreadID): starting?(s`threads(tid))
```

In the starting state all processes are running and there are no locks.

[6] `countCurrents` determines the number of threads having a `current` greater than 0.

We create a well founded ordering by defining a state to become smaller if the number of waiting processes decreases or alternatively, if the number of waiting processes remains the same and the total count of the number of processes that have obtained a lock is decreasing. Well foundedness follows directly from the well foundedness of the lexicographical ordering on pairs of natural numbers.

```
smallerState(s2, s1 : (validState?)) : bool =
    numberWaiting(s2) < numberWaiting(s1) ∨
        numberWaiting(s2) = numberWaiting(s1) ∧
        totalCount(s2) < totalCount(s1)
```

The `numberWaiting` function as well as the `totalCount` function are recursive functions on the array with thread information yielding the number of processes that have either a `RBLOCKED`, `RWAIT`, `WBLOCKED` or `WWAIT` status, and sum of all `current` fields respectively.

Once we have established that each state transition maintains the invariant, all we have to prove is that each transition, except for the starting state will possibly result in a state that is smaller. This is the `noDeadlock` theorem. Proving this theorem is mainly a case distinction with a couple of inductive proofs thrown in for good measure. The induction is needed to establish that the increase and decrease in the variables can only happen if certain preconditions are met. The proof takes about 300 proof commands.

```
noDeadlock: THEOREM
   ∀(s1: (validState?)) : ¬ startingState(s1) ⇒
      ∃(s2: (validState?)) : interleave(s1, s2) ∧ smallerState(s2, s1)
```

5 Related and future Work

Several studies investigated *either* the conversion of code to state transition models, as is done e.g. in [28] with mcrl2 *or* the transformation of a state transition model specified in a model checker to a state transition model specified in a theorem prover, as is done e.g. in [16] using VeriTech. With the tool TAME one can specify a time automaton directly in the theorem prover PVS [3]. For the purpose of developing consistent requirement specifications, the transformation of specifications in Uppaal [17] to specifications in PVS has been studied in [9].

In [22] model checking and theorem proving are combined to analyze the classic (non-reentrant) Readers/Writers problem. The authors do not start with actual source code but with a tabular specification that can be translated straightforwardly into SPIN and PVS. Safety and clean completion properties are derived semi-automatically. Model checking is used to validate potential invariants.

[13] reports on experiments in combing theorem proving with model checking for verifying transition systems. The complexity of systems is reduced abstracting out sources for unboundedness using theorem proving, resulting in an bounded system suited for being model checked. One of the main difficulties is that formal proof techniques are usually not scalable to real sized systems without an extra effort to abstract the system manually to a suitable model.

The verification framework SAL (See [25]) combines different analysis tools and techniques for analyzing transition systems. Besides model checking and theorem proving it provides program slicing, abstraction and invariant generation.

In [12] part of an aircraft control system is analyzed, using a theorem prover. This experiment was previously performed on a single configuration with a model checker. A

technique called *feature-based decomposition* is proposed to determine inductive invariants. It appears that this approach admits incremental extension of an initially simple base model making it better scalable than traditional techniques.

Java Pathfinder (JPF) [29] operates directly on Java making a transformation of source code superfluous. However, this tool works on a complete program, such that it is much more difficult to create abstractions. The extension of JPF with symbolic execution as discussed by [1] might be a solution to this problem.

An alternative for JPF is Bandera [7], which translates Java programs to the input languages of SMV and SPIN. Like in JPF, it is difficult to analyse separate pieces of code in Bandera. There is an interesting connection between Bandera and PVS. To express that properties do not depend on specific values, Bandera provides a dedicated language for specifying abstractions, i.e. concrete values are automatically replaced by abstract values, thus reducing the state space. The introduction of these abstract values may lead to prove obligations which can be expressed and proven in PVS.

In [24] a model checking method is given which uses an extension of JML [18] to check properties of multi-threaded Java programs.

With Zing [2] on the one hand models can be created from source code and on the other hand executable versions of the transition relation of a model can be generated from the model. This has been used successfully by Microsoft to model check parts of their concurrency libraries.

Future work

The methodology used (creating in a structured way a model close to the code, model checking it first and proving it afterwards) proved to be very valuable. We found a bug, improved the code, extended the capabilities of the code and proved it correct. One can say that the model checker was used to develop the formal model which was proven with the theorem prover. This decreased significantly the time investment of the use of a theorem prover to enhance reliability. However, every model was created manually. We identified several opportunities for tool support and further research.

Model checked related to source code Tool support could be helpful here: not only to 'translate' the code from the source language to the model checker's language. It could also be used to record the abstractions that are made. In this case that were: basic locks → lock process model, hash tables → arrays, threads → processes and some name changes. A tool that recorded these abstractions, could assist in creating trusted source code from the model checked model.

Model checked related to model proven It would be interesting to prove that the model in the theorem prover is equivalent with the model checked. Interesting methods to do this would be using a semantic compiler, as was done in the European Robin project [27], or employing a specially designed formal library for models created with a model checker, like e.g. TAME [3].

Model proven related to source code Another interesting future research option is to investigate generating code from the fully proven model. This could be code generated from code-carrying theories [15] or it could be proof-carrying code through the use of refinement techniques [4].

6 Concluding remarks

We have investigated Trolltech's widely used industrial implementation of the reentrant readers-writers problem. Model checking revealed an error in the implementation.

Trolltech was informed about the bug. Recently, Trolltech released a new version of the thread library (version 4.4) in which the error 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.

The improved Readers-Writers model described in this paper is *deadlock free* and *strongly reentrant*. The model was first developed and checked for a limited number of processes using a model checker. Then, the properties were proven for any number of processes using a theorem prover.

Acknowledgements

We would like to thank both Erik Poll and the anonymous referees of an earlier version of this paper for their useful comments improving the presentation of this work.

References

1. S. Anand, C. S. Pasareanu, and W. Visser. Jpf-se: A symbolic execution extension to java pathfinder. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 134–138. Springer, 2007.
2. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In R. Alur and D. Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 484–487. Springer, 2004.
3. M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *User Interfaces for Theorem Provers*, Eindhoven, The Netherlands, 1998.
4. M. A. Barbosa. A refinement calculus for software components and architectures. *SIGSOFT Softw. Eng. Notes*, 30(5):377–380, 2005.
5. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
6. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *POPL*, pages 117–126, 1983.
7. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448, 2000.
8. P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.
9. A. de Groot. *Practical Automaton Proofs in PVS*. PhD thesis, Radboud University Nijmegen, 2008.
10. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison Wesley Professional, 2006.
11. J. F. Groote, A. H. J. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. J. van Weerdenburg. The formal specification language mCRL2. In *Proc. Methods for Modelling Software Systems*, number 06351 in Dagstuhl Seminar Proceedings, 2007.
12. V. Ha, M. Rangarajan, D. Cofer, H. Rues, and B. Dutertre. Feature-based decomposition of inductive proofs applied to real-time avionics software: An experience report. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 304–313, Washington, DC, USA, 2004. IEEE Computer Society.

13. K. Havelund and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 662–681. Springer-Verlag, 1996.
14. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
15. B. Jacobs, S. Smetsers, and R. Wichers Schreur. Code-carrying theories. *Formal Asp. Comput.*, 19(2):191–203, 2007.
16. S. Katz. Faithful translations among models and specifications. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 419–434, London, UK, 2001. Springer-Verlag.
17. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
18. G. T. Leavens, J. R. Kiniry, and E. Poll. A jml tutorial: Modular specification and verification of functional behavior for java. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, page 37. Springer, 2007.
19. K. L. McMillan. The SMV System. 1998-2001. Carnegie Mellon University, www.cs.cmu.edu/ modelcheck/smv.html.
20. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
21. S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.
22. V. Pantelic, X.-H. Jin, M. Lawford, and D. L. Parnas. Inspection of concurrent systems: Combining tables, theorem proving and model checking. In H. R. Arabnia and H. Reza, editors, *Software Engineering Research and Practice*, pages 629–635. CSREA Press, 2006.
23. J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In M. Dezani-Ciancaglini and U. Montanari, editors, *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.
24. Robby, E. Rodríguez, M. B. Dwyer, and J. Hatcliff. Checking jml specifications using an extensible software model checking framework. *STTT*, 8(3):280–299, 2006.
25. N. Shankar. Combining theorem proving and model checking through symbolic analysis. In C. Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2000.
26. H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.
27. H. Tews, T. Weber, M. Völp, E. Poll, M. v. Eekelen, and P. v. Rossum. Nova Micro–Hypervisor Verification. Technical Report ICIS–R08012, Radboud University Nijmegen, May 2008. Robin deliverable D13.
28. M. van Eekelen, S. ten Hoedt, R. Schreurs, and Y. S. Usenko. Analysis of a session-layer protocol in mcrl2. verification of a real-life industrial implementation. In P. Merino and S. Leue, editors, *Proc. 12th Int'l Workshop on Formal Methods for Industrial Critical Systems (FMICS 2007)*, volume 4916 of *Lecture Notes Computer Science*, pages 182–199. Springer, 2008.
29. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.