Noname manuscript No.
(will be inserted by the editor)

Analysing Futex-based Synchronisation Primitives Using Model Checking

Hugues Evrard · Alastair F. Donaldson

the date of receipt and acceptance should be inserted later

Abstract The *futex* Linux system call enables implementing performant inter-thread and inter-process synchronisation primitives, such as mutexes and condition variables. However, the futex system call is notoriously difficult to use correctly. An early implementation of futex-based mutexes in the Linux Native POSIX Thread Library suffered from a subtle defect. When teaching about clever futex-based mutex designs that avoid these early shortcomings, we have found that their intricacies are hard to understand and difficult to convey to students. In this case study, we use the Promela modelling language to model a number of futex-based mutex and condition variable implementations, and the Spin model checker to verify safety properties over these models. We show that model checking is effective at confirming known bugs that affected real-world implementations, and at confirming that current implementations do indeed behave correctly in multi-threaded environments. We also investigate the effectiveness of symmetry reduction and two memory usage reduction techniques for increasing the scalability of model checking in this domain. The Promela models we have developed are available as open source. They may be useful as teaching material for classes that cover futex-based synchronisation primitives, and as a template for performing formal verification on new synchronisation primitive designs.

Keywords futex, mutual exclusion, condition variables, model checking, Promela/Spin

Hugues Evrard
Google
E-mail: hevrard@google.com

Alastair F. Donaldson
Department of Computing
Imperial College London
E-mail: alastair.donaldson@imperial.ac.uk

1 Introduction

The futex system call was introduced to the Linux kernel in the early 2000s in order to support efficient synchronisation primitives [20]. The name "futex" is derived from "**f**ast **u**serspace mu**tex**", because one of the most important use cases for the futex system call is the efficient implementation of mutexes, striking a balance between OS semaphores, whose manipulation involves a system call even when contention is low, and spinlocks, which avoid system calls by operating entirely in userspace but may lead to high CPU usage when contention is high.

When used in a careful and clever manner, futexes can enable efficient inter-thread and inter-process synchronisation. However, futexes are notoriously difficult to use correctly. According to Drepper, in his aptly-titled paper "Futexes are Tricky" [15], a package authored by one of the inventors of the futex system call, containing user-level code demonstrating its use, turned out to be incorrect, and for a period of time a futex-based mutex implementation in the Linux Native POSIX Runtime Library also suffered from a performance-related defect. Drepper describes these problems and presents two alternative implementations, arguing their correctness informally.

More recently, Gustedt [26] presented another approach for a futex-based lock primitive akin to a mutex, and contributed a variation of his approach as the low-level lock primitive of the Musl C standard library [39]. While the method by which this primitive avoids deadlock is discussed in detail, the guarantee that no more than one thread can lock the mutex at a time is only mentioned informally.

In an article on his personal website about futex-based condition variables [9], Denis-Courmont describes a number of flawed proposals for implementing condition variables, explaining why they are incorrect, and a proposal that is argued to be correct under reasonable practical assumptions.

A limitation of these expositions of futex-based synchronisation primitives is that they provide only informal descriptions of how code snippets might behave in a concurrent context. The reader may not fully understand the (often subtle) arguments for (in)correctness, and even if they do, it may be hard for them to imagine the consequences of alternative implementation choices.

In this case study, we investigate the use of the Promela language and its associated Spin model checker [30] to formally express and analyse various proposals for futex-based mutexes and condition variables that have been discussed in previous works [15,9,26]. Due to the ability of model checking to produce counterexamples, our Promela models of incorrect implementations lead to step-by-step traces that illustrate bug-triggering thread interleavings. This facility also aids in understanding why certain details of correct implementations are important, because one can change those details and inspect the counterexamples that arise as a result. We show that model checking can detect bugs that affected real-world implementations of mutexes and that it can confirm bugs in both naive and real-world implementations of condition variables. We also show that model checking aids in understanding the importance of certain intricacies of a futex-based mutex design. In addition, we present experimental results assessing the scalability of Spin-based model checking for verifying correctness of futex-based mutex implementations in scenarios with increasing numbers of threads, assessing the effectiveness of symmetry reduction and two different memory usage reduction approaches at increasing scalability.

The Promela models we have developed are available as open source, together with instructions on how to use Spin to analyse them [17]. We envisage that they may be useful as teaching material in classes that cover futex-based synchronisation primitives. In fact, our investigation into the application of model checking to this problem was inspired by the experience of one of the authors teaching about futex-based mutexes on a course at Imperial College London, and being dissatisfied with his informal correctness-related explanations. We also hope that our models will serve as a template for performing formal verification on new synchronisation primitive designs.

The rest of the paper is organised as follows. In Section 2 we provide necessary background on the futex system call. We explain how we have modelled this system call in Promela in Section 3. Our Promela models of mutexes and condition variables rely on the modelling of various integer atomic operations, including operations that may overflow; we discuss these in Section 4. In Section 5 we work through examples of futex-based mutex implementations from Drepper's paper [15], explaining how we have modelled each mutex variant using Promela and presenting insights into our analysis of these models using Spin. In Section 6 we do the same for the two futex-based mutex variations from Gustedt's paper [26] and his contribution to the Musl C standard library [39]. Then in Section 7 we turn to condition variables, working through various implementation proposals from Denis-Courmont's article [9] and discussing our use of Promela and Spin for modelling and analysis. In Section 8 we present experimental results assessing the scalability of model checking for verifying correctness of futex-based mutex implementations, with and without symmetry reduction and two memory usage reduction techniques. We discuss related work in Section 9 and conclude with a discussion of future directions in Section 10.

Throughout the paper we assume the reader is familiar with the syntax of Promela and with basic operation of the Spin model checker, referring the reader to the definitive reference for more details [30].

Contribution over our prior work This work extends a paper published at SPIN 2023, the International Symposium on Model Checking Software [18]. Our main additional contributions are: discussion of modelling and model checking results for an additional futex-based mutex implementation from Drepper's paper [15, Section 6] and a further variation of this mutex implementation featuring an additional optimisation (Section 5.4); the application of Promela and Spin to two futex-based mutex implementations from Gustedt [26] on which a mutex implementation in the Musl C standard library [39] is based (Section 6); the application of Promela and Spin to additional futex-based condition variable implementations discussed by Denis-Courmont [9] (Section 7.3 and Section 7.5); and experimental results assessing the effectiveness of symmetry reduction and memory usage reduction techniques (Section 8).

2 The Futex System Call

The word *futex* is often used to designate three things: (1) a 32-bit addressable value also called a *futex word*, (2) the futex system call, and (3) mutex implementations based on the futex system call. In this section, we are concerned with (1) and (2), while (3) is discussed in Section 5 and Section 6.

The futex system call enables threads to block depending on the value of a given memory word—a *futex word*—or to wake up threads that are waiting in a queue associated with a futex word. In practice, a futex system call has the form shown in Listing 1, where `SYS_futex` is the futex system call ID.[1]

The system call is multiplexed via its `futex_op` argument, which refers to one of various operations. In this case study, we focus on the two main operations, `FUTEX_WAIT`

[1] https://man7.org/linux/man-pages/man2/futex.2.html

```
long syscall(SYS_futex,
  // pointer to futex word
  uint32_t *addr,
  // operation (e.g. FUTEX_WAIT, FUTEX_WAKE)
  int futex_op,
  // plain value argument
  uint32_t val,
  // extra arguments for other operations
  // (not used in this paper)
  ...);
```

Listing 1 Signature of the futex system call

```
typedef Futex {
  // Futex word
  byte word;
  // Wait queue: array of bool indexed by thread IDs;
  // thread T is waiting iff wait[T] is true
  bool wait[NUM_THREADS];
  // Number of threads currently waiting
  byte num_waiting;
}
```

Listing 2 The `Futex` type in Promela

and `FUTEX_WAKE`, where only the `addr` and `val` arguments are relevant.

`FUTEX_WAIT`: the calling thread blocks and goes to sleep only if the value of the futex word addressed by `addr` is equal to the plain value argument `val`. In this case, the thread joins a queue of *waiters* associated with the address of the given futex word. The `FUTEX_WAIT` operation is atomic with respect to the futex word, which is typically in memory shared between threads. This call has *compare-and-block* semantics: loading the futex word's value, comparing it to `val` and adding the thread to the waiters queue (if appropriate) happen atomically and are totally ordered with respect to other concurrent operations on the futex word.

`FUTEX_WAKE`: the calling thread wakes up threads in a queue of waiters associated with `addr`, which again is the address of a futex word. It wakes `val` threads, or the number of threads waiting on `addr`, whichever is smaller. Beyond this, there is no guarantee on which threads are woken up, or in which order threads are woken up. In practice, `val` is typically either 1 (to ask for a single thread to be woken) or `INT_MAX` (to wake up all waiters).

When presenting code examples, we use `futex_wait (addr, val)` to denote a futex system call performing `FUTEX_WAIT` with the given `addr` and `val` parameters, and similarly for `futex_wake(addr, val)`.

The name "futex" is derived from **f**ast and **u**serspace because futex-based synchronisation primitive implementations (such as implementations of mu**tex**es) typically first try synchronising in userspace via C atomic operations on a shared futex word, and only resort to slower futex system calls in case of contention. We see this pattern in the mutex implementations of Section 5 and Section 6.

Note that there is no need to register `addr`, the address of a futex word, before using it with futex system calls. When a thread calls `futex_wait(addr, val)`, if the thread needs to go sleep and the kernel does not yet know about `addr` then a queue of waiters specific to `addr` is created. Similarly, if `futex_wake(addr, val)` is called when no queue of waiters exists for `addr` the call immediately returns. Also, when the final waiter in the wait queue of `addr` is woken, the queue is deallocated. This means that the kernel overhead associated with the use of futex-based synchronisation primitives is minimal: wait queues and book-keeping state only exist when threads are actually waiting. Millions of futex-based objects can be in use by an application, with only a very small number of them actually known by the by kernel at any moment.

3 Modelling the Futex System Call Variants

We model futexes in Promela as a `Futex` type, and two inline macros `futex_wait` and `futex_wake` to represent these variants of the general system call. Before covering these in detail, we make some general remarks about our modelling approach.

To keep the state vector size under control, we use `byte` values virtually everywhere we would use `int` values in C. This is without loss of generality since, in our examples with up to a dozen threads, all interesting values are within the range [0, 255].

Threads are mapped to Promela's `proctype` and are identified by their `_pid` builtin variable. The total number of threads is a global constant that we use to dimension arrays, defined by a preprocessor macro, `NUM_THREADS`, so that it can be easily changed when invoking Spin (e.g. via a command such as `spin -DNUM_THREADS=5 ...`).

Now, on to futexes. The `Futex` type, shown in Listing 2, contains a futex word, the queue of threads that are waiting on this futex word, and a counter recording the number of threads that are currently waiting.

An array indexed by thread IDs is used to model the wait queue. This will prove convenient to allow waking up sleeping threads in a non-deterministic order. In a C program, each futex is identified by the address of its futex word; here each futex is identified by a variable of type `Futex` which is in global scope so that all threads can refer to it.

The `futex_wait` call is modelled by the inline macro of Listing 3. As arguments it takes a variable of type `Futex` —a futex word—and a plain value to compare to the futex word. If they are equal, the thread goes to sleep: we set its entry in the wait queue, and increment the counter of waiting threads. An assertion checks that only non-sleeping threads may go to sleep. Then, the thread blocks until its wait queue entry is set to false. If the value argument differs from the

```
 1 inline futex_wait(futex, val) {
 2   if
 3   :: d_step {
 4        futex.word == val ->
 5        printf(
 6          "T%d futex_wait, value match: %d; sleep\n",
 7          _pid, futex.word);
 8        // The thread must not be sleeping already
 9        assert(!futex.wait[_pid]);
10        futex.wait[_pid] = true;
11        futex.num_waiting++;
12      }
13      d_step {
14        !futex.wait[_pid] ->
15        printf("T%d has woken\n", _pid);
16      }
17   :: d_step {
18        else ->
19        printf(
20          "T%d futex_wait, value mismatch: %d vs. %d; do
      not sleep\n",
21          _pid, futex.word, val);
22      }
23   fi
24 }
```

Listing 3 Modelling `futex_wait` in Promela

```
inline futex_wake(futex, num_to_wake) {
  atomic {
    // The waker must not be asleep
    assert(!futex.wait[_pid]);
    num_woken = 0;
    do
    :: num_woken == num_to_wake ||
         futex.num_waiting == 0 ->
       break
    :: else ->
       if
       :: futex.wait[0] -> futex.wait[0] = false;
          printf("T%d wakes T0\n", _pid)
       :: futex.wait[1] -> futex.wait[1] = false;
          printf("T%d wakes T1\n", _pid)
#if NUM_THREADS > 2
       :: futex.wait[2] -> futex.wait[2] = false;
          printf("T%d wakes T2\n", _pid)
#endif
#if NUM_THREADS > 3
       :: futex.wait[3] -> futex.wait[3] = false;
          printf("T%d wakes T3\n", _pid)
#endif

...

#if NUM_THREADS > 12
#error "NUM_THREADS > 12, add more if branches in
     futex_wake"
#endif
       fi;
       futex.num_waiting--;
       num_woken++;
    od;
    printf("T%d woke up %d thread(s)\n",
           _pid, num_woken);
    // Reset to avoid state space explosion
    num_woken = 0;
  }
}
```

Listing 4 Modelling `futex_wake` in Promela

futex word, the thread continues without blocking. Log messages prefixed by the ID of the executing thread are printed to ease the understanding of counterexamples.

The atomic compare-and-block semantics is modelled via a `d_step` (deterministic step) scope (line 3), which *"introduces a deterministic code fragment that is executed indivisibly"* [30, p. 401]. This is a better choice compared with the alternative `atomic` scope that Promela offers. This is because all statements in a `d_step` are treated as leading to a single state change by Spin, thus reducing the search depth. In contrast, the statements in an `atomic` scope are executed atomically if no statement blocks, but an intermediate state for each statement is recorded (a) in case nondeterminism is encountered mid-way through the scope, requiring an intermediate state to have multiple successors, (b) to allow control to transfer into or out of the atomic scope at positions other than the scope start and end, and (c) in case a statement blocks (in which case state-space exploration will switch to consider an action by a different process in the Promela model). It is safe to use `d_step` over `atomic` here since all contained statements are deterministic, there is no jump in or out the `d_step` scope, and there is no blocking statement in the middle of the scope (here even the statement in line 4 cannot block since the other `if` branch starts with an `else`). The `d_step` scopes (lines 13 and 17) guarantee that print statements associated with value match/mismatch checks display their messages as soon as the associated check occurs. This is important for ensuring readability of counterexamples: it avoids logging messages from other threads being interleaved between a value check occurring and its associated print statement being executed.

The `futex_wake` call is modelled by the inline macro of Listing 4; again, log messages are used to improve readability of counterexamples. The macro requires that a local variable `num_woken` is in scope.[2]

The `num_to_wake` argument indicates the number of threads to wake up, and `num_woken` counts the number of threads that have been woken so far. We cannot eliminate `num_woken` and instead decrement `num_to_wake` until it reaches zero since the `num_to_wake` macro argument may be a literal value. This will be the case with a call such as `futex_wake(futex, 1)`. A loop is used to wake one thread per iteration, until the desired number of threads have been woken or there are no more threads to wake. When waking a thread, a nondeterministic `if` is used to pick one of the sleeping threads, which is then woken up by setting its entry in the futex wait queue array to `false`.

The whole macro body is contained in an `atomic` scope to prevent concurrent accesses to the futex internals. This time, `d_step` cannot be used due to the nondeterministic order in which threads are woken; recall from above that

[2] Although Spin does allow local variables to be defined in inline macros, we have found that using this feature leads to unexpected increases in state space size. To avoid this, throughout our modelling, we take the approach of declaring all local variables that a process will use at the start of its `proctype` declaration.

```
inline cmpxchg(location, expected, desired, result) {
  d_step {
    result = location;
    location = (location == expected
                -> desired
                : location)
  }
}
```

Listing 5 Modelling compare-and-exchange in Promela

```
#define inc(a)   (a == MAX_BYTE_VALUE -> 0 : a + 1)
inline fetch_inc(location, result) {
  d_step {
    result = location;
    location = inc(location)
  }
}
```

Listing 6 Modelling fetch_inc in Promela

a d_step scope requires the code fragment that it contains to behave deterministically, and documentation about the construct states that *"If non-determinism is present, it is resolved in a fixed and deterministic way, for instance, by always selecting the first true guard in every selection and repetition structure"* [30, p. 401]. This would not be appropriate in the context of futex_wake: the futex system call does not guarantee the order in which waiting threads will be woken, so that faithful modelling demands that the threads that are woken should be selected nondeterministically from the pool of waiting threads.

At the end of the atomic scope, num_woken is reset to zero. This is vital to reduce state-space explosion: it prevents Spin from regarding otherwise identical states that differ only in the final value of num_woken as distinct, which would lead to Spin continuing its exhaustive search from each such state [40].

Relying on the non-deterministic selection of enabled if branches requires exactly NUM_THREADS branches: we use the C preprocessor to achieve this, supporting up to 12 threads, with it being easy to support more threads by adding further if branches. To support an arbitrary thread count, one could easily script the generation of these branches. We opt for the C preprocessor approach to keep the Promela code self-contained.

4 Modelling Atomic Operations and Overflow

The mutex and condition variable implementations rely on C/C++ atomic operations that we model in Promela. The atomic compare-and-exchange operation, cmpxchg, compares the value at a given location with an expected value. If the values match, the location is updated to a given value, desired. Otherwise the location is left unchanged. Either way, the original value of the location is returned. This operation is modelled by the inline macro of Listing 5, which uses a result parameter in lieu of returning a value.

The atomic exchange operation xchg, which unconditionally exchanges the value at a given location with a new value, is modelled in a similar fashion.

The atomic fetch-and-increment operation fetch_inc returns the current value of a location before incrementing it. We make sure to model overflow and wrapping on byte values, but in order to limit the state space and the size of counterexamples, we introduce a tighter upper bound MAX_BYTE_VALUE, which is set to the total number of threads plus one, since byte variables tend to count some number of threads. This is without loss of generality, since C/C++ atomic integers also wrap upon overflow.

We define the inc macro to handle overflow, and use d_step to make fetch_inc atomic as shown in Listing 6. In a similar fashion, we define a dec macro that handles underflow, and a fetch_dec macro for atomic fetch-and-decrement. Some of the Promela models discussed later also make direct use of the inc macro when performing a non-atomic increment in a local expression, rather than operating on a futex word.

5 Model Checking Futex-based Mutexes from Drepper's Paper

We describe the usage scenario and properties for mutexes to which model checking is applied (Section 5.1). We then describe the modelling and verification of various mutex implementations from Drepper's paper [15] (Section 5.2—Section 5.4).

5.1 Model Checking Harness and Properties

We use the harness of Listing 7 to enable model checking of various futex-based mutex implementations presented here and in Section 6. This harness uses an active proctype to launch NUM_THREADS threads, each of which uses the lock() and unlock() inline macros to repeatedly lock and unlock a shared mutex. Separate versions of these macros are provided for each mutex implementation discussed below. The macros assume that a global variable of type Futex (see Listing 2) is available. Global variable num_threads_in_cs, initialised to 0 by default, is used to record when threads enter and leave the critical section. The version of the harness in our source code repository also features declarations of various local variables that are relied on by inline macros (e.g. the num_woken variable used in Listing 4). We omit these in Listing 7 for brevity.

We consider model checking of two safety properties: (1) freedom from invalid end states (a built-in feature of

```
// Number of threads in the critical section (CS)
byte num_threads_in_cs;

active [NUM_THREADS] proctype Thread() {
  do
  :: lock();
     num_threads_in_cs++;
     num_threads_in_cs--;
     unlock();
  :: printf("T%d is done\n", _pid) -> break
  od
}

// Never more than one thread in CS
active proctype Monitor() {
end:
  atomic {
    num_threads_in_cs > 1 -> assert(false);
  }
}
```

Listing 7 Harness for model checking futex-based mutexes

```
class Mutex {
public:
  Mutex() : futex_word(0) {}
  void lock() {
    uint32_t old_value;
    while ((old_value = futex_word.fetch_add(1)) != 0)
      futex_wait(&futex_word, old_value + 1);
  }
  void unlock() {
    futex_word.store(0);
    futex_wake(&futex_word, 1);
  }

private:
  atomic<uint32_t> futex_word;
};
```

Listing 8 Incorrect futex-based mutex implementation adapted from "Mutex, Take 1" in Drepper's paper [15]

Spin), which confirms that it is not possible for a thread to become blocked in a call to `futex_wait` when all other threads have terminated, and (2) mutual exclusion, captured by the `Monitor` active proctype shown in Listing 7. The monitor is designed to check that, in every reachable state, the number of threads in the critical section does not exceed one. The `end` label informs Spin that it is acceptable for the system to end up in a state where the monitor is blocked checking its condition despite all other processes having terminated, so that this scenario is not reported as an invalid end state.

5.2 Incorrect Futex-based Mutex

Listing 8 shows C++ code adapted from a presentation of a subtly incorrect futex-based mutex [15, Section 4]. The futex word is the 32-bit atomic integer field `futex_word`. The intention is that the mutex is unlocked if and only if `futex_word` has value 0.

A thread attempts to lock the mutex by atomically incrementing `futex_word` via a `fetch_add`, storing its previous value in local variable `old_value`. If the previous value is 0 then the thread has locked the mutex, by changing `futex_word` from 0 to 1, and can return from `lock`. Otherwise, the thread calls `futex_wait` passing the argument `old_value + 1` in order to go to sleep until the lock becomes free. Recall from Section 2 that `futex_wait` only puts a thread to sleep if the value of the futex word is equal to the integer passed as the second argument. Thus the value `old_value + 1` is passed to `futex_wait`: if no other thread modifies the futex word between the `fetch_add` and `futex_wait` calls, this value will match the futex word and the thread will go to sleep until the lock becomes free. However, if another thread modifies the futex word before the call to `futex_wait`, then this call will not put the first thread to sleep so that the thread will immediately attempt to acquire the mutex again via another `fetch_add`.

Unlocking the mutex is simpler: `futex_word` is set to 0, and `futex_wake` is called so that one of the threads waiting on `futex_word`, if any, will be woken.

Correctness aside, Drepper notes that this mutex implementation is not ideal because it involves a `futex_wake` call *every* time the mutex is unlocked, regardless of whether there are waiters, a deficiency that is fixed in the mutex implementations studied in Section 5.3 and Section 5.4.

Drepper discusses a correctness issue triggered by an overflow of the futex word. Suppose several threads are contending to try to lock an already-locked mutex. It is possible that while a given contending thread T1 is between the calls to `fetch_add` and `futex_wait`, another contending thread T2 calls `fetch_add` and modifies the futex word, such that T1 will not go to sleep and will itself call `fetch_add` again, preventing T2 from going to sleep. This can go on until the futex word wraps back to 0, in which case a contending thread will believe it has successfully locked the mutex. Drepper notes [15, Section 4] that this extreme scenario is made more likely in practice by the fact that `futex_wait` may return prematurely if the waiting thread is interrupted by a signal [22].

This mutex design is modelled in Promela by the inline macros of Listing 9. The macros rely on a local variable, `old_value`, being in scope.

We use print statements so that counterexamples produced by Spin are readable. We use `atomic` and `d_step` scopes to (a) ensure that print statements are executed atomically with the actions that they aim to document, and (b) limit state explosion by allowing interleavings only between operations that have inter-thread visibility (known as *visible operations* [19]): statements that manipulate the futex word and calls to the `futex_wait` and `futex_wake` inline macros. For example, there must be an interleaving point between `fetch_inc` (line 4) and `futex_wait` (line 14), but there is no value in considering thread interleavings between the call to `fetch_inc` and the `if..fi` that imme-

```
inline lock() {
  do
  :: atomic {
       fetch_inc(futex.word, old_value);
       if
       :: old_value == 0 ->
          printf("T%d locks mutex\n", _pid);
          break
       :: else ->
          printf("T%d lock fail, old_value: %d\n",
                 _pid, old_value);
       fi
     }
     futex_wait(futex, inc(old_value))
  od
}

inline unlock() {
  d_step {
    futex.word = 0;
    printf("T%d unlocks mutex\n", _pid)
  }
  futex_wake(futex, 1)
}
```

Listing 9 Modelling the incorrect futex-based mutex of Listing 8 in Promela

diately follows. These only involve a thread manipulating its local state. An interleaving point will cause needless state-space explosion which we have found Spin's partial order reduction (which is based on conservative static analysis [31]) does not completely alleviate.

Recall from Section 4 that the `fetch_inc` operation is modelled over a small integer range, so that overflows rapidly occur.

With two threads, Spin quickly verifies the mutual exclusion property and confirms that all end states are valid. This makes sense: the bug described above requires a race between multiple contending threads when the mutex is already held by a further thread.

With three threads, Spin quickly reports a counterexample (minimised using Spin's iterative shortening algorithm) with the following messages, showing that it is possible for two threads to simultaneously acquire the mutex:

Thread	Message
T0	locks mutex
T1	lock fail, `old_value`: 1
T2	lock fail, `old_value`: 2
T1	`futex_wait`, value mismatch: 3 vs. 2; do not sleep
T1	lock fail, `old_value`: 3
T2	`futex_wait`, value mismatch: 4 vs. 3; do not sleep
T2	lock fail, `old_value`: 4
T1	`futex_wait`, value mismatch: 0 vs. 4; do not sleep
T1	locks mutex

This nicely illustrates the problem where threads T1 and T2 repeatedly prevent one another from sleeping, with each thread incrementing the futex word before the other can call `futex_wake`: "value mismatch: 0 vs. 4" shows the futex word wrapping from 4 (the value limit when checking using three threads; see Section 4) to 0.

When the monitor process that checks for mutual exclusion is disabled, Spin also confirms that the "no invalid end states" property fails, though with a longer counterexample. Here is a summary of the problem. Suppose that T0 holds the lock. T1 and T2 then get into a race, incrementing the futex word until T1 observes the word's old value to be 3 and T2 observes the word's old value to be 4, so that the word's *current* value is 0 (T2 having caused it to wrap-around). T1 is poised to call `futex_wait(4)`, and T2 is poised to call `futex_wait(0)`, but neither have done so yet.

T0 unlocks the mutex (so that the value of the futex word remains 0), wakes up no threads (because neither T1 nor T2 has managed to go to sleep) and terminates. T1 calls `futex_wait(4)`, which directly returns due to a value mismatch. T1 tries to lock the mutex again, *succeeds* (incrementing the futex word from 0 to 1), immediately unlocks the mutex (setting the futex word to 0), wakes up no threads (because T2 has not managed to go to sleep) and terminates. Finally, T2 calls `futex_wait(0)`, which it has been poised to do for a while. Because the value of the futex word *is* 0, T2 goes to sleep. All other threads having terminated, T2 is stuck in its sleeping state.

The problem with this mutex design has been reported to affect real code [15]. It is encouraging that model checking can quickly expose it, providing clear counterexamples.

It is important to note, however, that model checking was able to provide a concise counterexample for this issue because of our use of a counter that rapidly overflows (as discussed in Section 4), which we designed based on prior knowledge of the overflow-related problem. If instead we use a counter that only overflows upon reaching the maximum value of 255 for a byte in Promela, Spin reports a counterexample involving 3,721 steps, which would be a lot harder to comprehend.

5.3 Correct Futex-based Mutex

Drepper goes on to present mutex implementation shown in Listing 10, which is more intricate compared with that of Listing 8. The mutex implementation of Listing 10 is claimed to be correct [15, Section 5].

We use *waiters* to refer to threads that are asleep due to having called `futex_wait`. In this implementation, the futex word can take one of three values. A value of 0 means that the mutex is free, while values 1 and 2 mean that some thread, say T, holds the mutex. If the futex word is 1, a state referred as "locked, no waiters", then when T unlocks the mutex, T is not obliged to wake up any waiters. In contrast,

```
 1 class Mutex {
 2 public:
 3   Mutex() : futex_word(0) {}
 4   void lock() {
 5     uint32_t old_value;
 6     if ((old_value = cmpxchg(futex_word, 0, 1)) != 0)
 7       do {
 8         if (old_value == 2
 9               || cmpxchg(futex_word, 1, 2) != 0)
10           futex_wait(&futex_word, 2);
11         old_value = cmpxchg(futex_word, 0, 2);
12       } while (old_value != 0);
13   }
14   void unlock() {
15     if (futex_word.fetch_sub(1) != 1) {
16       futex_word.store(0);
17       futex_wake(&futex_word, 1);
18     }
19   }
20 
21 private:
22   atomic<uint32_t> futex_word;
23 };
```

Listing 10 Correct futex-based mutex implementation adapted from "Mutex, Take 2" in Drepper's paper [15]

if the futex word is 2, a state referred as "locked, waiters", then when T unlocks the mutex, T must call `futex_wake` to request that one waiter be woken.

In `lock`, a thread T first tries to change the value of the futex word from 0 to 1 via a `cmpxchg` (line 6). If T succeeds in doing this then it has locked the mutex and can return. In this case, we say that the thread has locked the mutex on the *fast path*.

Otherwise, T must contend for the mutex on the *slow path*, via a loop (line 7). The thread considers making a call to `futex_wait` to go to sleep and be notified when the mutex becomes free. Before this (line 8) T checks whether the previous value of the futex word was already 2 ("locked, waiters"). If not, the previous value of the futex word must have been 1 ("locked, no waiters"), so T attempts to change the value from 1 to 2 via another `cmpxchg` (line 9). Normally T will then call `futex_wait` (line 10), but if the `cmpxchg` returns a previous value of 0 this indicates that the mutex has suddenly become free, in which case there is no point calling `futex_wait`; instead, T should try again to lock the mutex.

Once T returns from `futex_wait`, or if T decided not to perform this call due to observing the mutex to be free, it performs another `cmpxchg` (line 11) to try to lock the mutex. In contrast to line 6, here T attempts to change the futex word from 0 to 2 ("locked, waiters"): T had to contend for the mutex (and was possibly itself a waiter). By setting the futex word to 2, T records the fact that contention has occurred and that there might exist sleeping threads that will need to be woken. The thread T leaves the loop only when the `cmpxchg` (line 11) returns 0: we say that T has locked the mutex *on the slow path*.

The `unlock()` function is simpler: the futex word is atomically decremented and its old value inspected (line 15). If the old value is 1, "locked, no waiters", this means that (a) the value of the futex word is now 0, so the mutex is unlocked as desired, and (b) the unlocking thread has no obligation to wake up any waiters, so the thread is done and can return from `unlock()`. Otherwise the old value must have been 2, "locked, waiters". In this case the unlocking thread sets the futex word to 0 (line 16) and calls `futex_wake` to wake up one waiter, if any waiters exist (line 17).

Although the code is short, this clever mutex implementation is difficult to understand. It is unlikely that a reader will gain a full understanding of the design from a best-effort prose explanation such as the above, or the explanation given by Drepper [15]. Particularly subtle is the fact that the futex word can have value 1, "locked, no waiters", despite the fact that there *are* waiters, and conversely the mutex word can have value 2, "locked, waiters" even though there are *no* waiters. Reasoning informally that this mutex implementation is correct is difficult, hence why we decided to model it formally.

The Promela `lock()` and `unlock()` inline macros for this mutex implementation are presented in Listing 11. As with the Promela code of Listing 9 we use print statements for counterexample readability and use `atomic` and `d_step` so that threads only interleave after issuing visible operations. The Promela code is a fairly straightforward reflection of the original C++ code, but the differences in the structured control flow constructs offered by the language led to us making use of Promela's `goto`.

Checking correctness The mutual exclusion property and freedom from invalid end states (see Section 5.1) are verified by Spin for our model of this mutex implementation within seconds for up to 5 threads, and Spin can verify a configuration with 6 threads in a matter of minutes. Employing symmetry reduction, verification scales further to 12 threads. These experimental results, and the associated experimental setup, are presented in full in Section 8.

Understanding bugs in incorrect variants Having a formal, checkable model makes it easy to experiment with the intricacies of this futex-based mutex implementation and understand why they are needed. We give two examples of changes to the mutex implementation that compromise its correctness in ways that might not seem immediately obvious. For each, we show that model checking quickly produces short, illuminating counterexample traces.

Bug 1: incorrect simplification. On line 9 of Listing 10, the conditions under which a thread calls `futex_wait` are rather complex and, as discussed by Drepper [15], some of this intricacy is for purposes of optimisation. One might wonder whether, from a correctness point of view, it would

```
inline lock() {
  atomic {
    cmpxchg(futex.word, 0, 1, old_value);
    if
    :: old_value == 0 ->
       printf("T%d locks mutex on fast path\n", _pid);
       goto acquired_mutex
    :: else ->
       printf("T%d fails to lock mutex on fast path\n",
              _pid)
    fi
  }
  do
  :: atomic {
       if
       :: old_value == 2
       :: else ->
          assert(old_value == 1);
          cmpxchg(futex.word, 1, 2, old_value);
          if
          :: old_value == 0 -> goto retry
          :: else
          fi
       fi
     }
     futex_wait(futex, 2);
     retry:
     atomic {
       cmpxchg(futex.word, 0, 2, old_value);
       if
       :: old_value == 0 ->
          printf("T%d locks mutex on slow path\n",
                 _pid);
          goto acquired_mutex
       :: else ->
          printf(
            "T%d fails to lock mutex on slow path\n",
               _pid)
       fi
     }
  od;
  acquired_mutex: skip;
}

inline unlock() {
  d_step {
    fetch_dec(futex.word, old_value);
    printf("T%d decrements futex word from %d to %d\n",
           _pid, old_value, futex.word)
  }
  if
  :: d_step {
       old_value == 2 ->
       futex.word = 0;
       old_value = 0
     }
     futex_wake(futex, 1)
  :: d_step {
       old_value == 1 ->
       old_value = 0
     }
  fi
}
```

Listing 11 Modelling the correct futex-based mutex of Listing 10 in Promela

suffice for a thread that just failed to lock the mutex to set the futex word to 2 (“locked, waiters”), and call `futex_wait` in an attempt to go to sleep. This would amount to replacing lines 8–10 of the C++ code with:

```
futex_word.store(2);
futex_wait(&futex_word, 2);
```

This change does not lead to violations of the mutual exclusion property, but does lead to the possibility of “lost waiters”, i.e. threads stuck waiting while all other threads have terminated.

Making corresponding adjustments to `lock()` in our Promela model (including adding a print statement to log the storing of 2 to `futex_word` by a thread), Spin quickly produces the following counterexample when invoked on a 2-threaded configuration:

Thread	**Message**
T0	locks mutex on fast path
T1	fails to lock mutex on fast path
T0	decrements futex word from 1 to 0
T0	is done
T1	sets `futex.val` to 2
T1	`futex_wait`, value match: 2; sleep

The problem is that between T1 observing the mutex to be unavailable and setting the futex word to 2, T0 unlocks the mutex, waking up no waiters, because there are none yet, and terminates. T1 then sets the futex word to 2, goes to sleep and is never woken.

Bug 2: incorrect `cmpxchg`. On line 11 of Listing 10, when a thread attempts to lock the mutex on the slow path it tries to change the value of the futex word from 0 to 2, in contrast to the fast path, where a value change from 0 to 1 is attempted (line 6). A reasonable question is: is it essential that the slow path changes the futex word to 2? Adapting the `lock()` implementation in Promela so that the slow path changes the futex word to 1 instead of 2, and applying Spin to a two-threaded configuration leads to successful verification. But with three threads, Spin quickly reports a counterexample demonstrating an invalid end state:

Thread	**Message**
T0	locks mutex on fast path
T1	fails to lock mutex on fast path
T1	`futex_wait`, value match: 2; sleep
T2	fails to lock mutex on fast path
T2	`futex_wait`, value match: 2; sleep
T0	decrements futex word from 2 to 1
T0	wakes T2
T0	woke up 1 thread(s)
T0	is done
T2	has woken
T2	locks mutex on slow path
T2	decrements futex word from 1 to 0
T2	is done

The counterexample illustrates a situation where threads T1 and T2 go to sleep due to T0 holding the mutex. When the mutex becomes free, T0 wakes up T2, and T0 terminates. T2 then succeeds in locking the mutex on the slow path, but (due to the change we have introduced) does *not* set the futex

```
 1  void lock() {
 2    uint32_t old_value;
 3    if ((old_value = cmpxchg(futex_word, 0, 1)) != 0) {
 4      if (old_value != 2)
 5        old_value = xchg(futex_word, 2);
 6      while (old_value != 0) {
 7        futex_wait(&futex_word, 2);
 8        old_value = xchg(futex_word, 2);
 9      }
10    }
11  }
```

Listing 12 Optimised `lock` function adapted from “Mutex, Take 3” in Drepper’s paper [15]

word to 2. As a result, when T2 unlocks the mutex it is not obliged to wake up any waiters, so T1 remains asleep. T2 then terminates, so that T1 becomes a lost waiter.

This concrete example sheds light on why it is *essential* that the `cmpxchg` used to lock the mutex on the slow path changes the futex word to the “locked, waiters” state: this ensures that if there are additional waiters, the thread that succeeds in locking the mutex on the slow path is guaranteed to wake up one of them. Model checking facilitates experimenting with design variations, and quickly produces counterexamples that clearly illustrate defects.

5.4 Optimised Futex-based Mutex

We have also used Promela to model two optimisations to the futex-based mutex of Listing 10, one which is also presented in Drepper’s paper, and another that reflects an optimisation present in the `pthread_mutex_unlock` implementation in glibc.

Using `xchg` when locking The mutex of Section 5.3 uses the `cmpxchg` (compare-and-exchange) atomic operation. Modern architectures also feature an “exchange” instruction, `xchg`. Recall from Section 4 that `xchg` exchanges the value of an atomic variable with a new value, returning the old value, without also performing a comparison.

Drepper presents an optimised alternative of the mutex lock function that uses `cmpxchg` in the fast path, but then uses `xchg` thereafter [15, Section 6]. This is illustrated by the code of Listing 12 (the remainder of the mutex class is the same as in Listing 10).

This optimisation saves one atomic operation per loop iteration, because the `xchg(futex_word, 2)` in the loop at line 8 of Listing 12 *unconditionally* sets the futex word to 2 (with the `xchg` at line 5 ensuring that the futex word is set to 2 on loop entry if it was not already found to be 2 by the attempt to lock the mutex on the fast path). In contrast, in Listing 10 the `cmpxchg(futex_word, 0, 2)` at line 11 will not change the futex word if it is found to have value 1, hence the need for the `cmpxchg(futex_word, 1, 2)` at line 9.

```
void unlock() {
  if (xchg(futex_word, 0) == 2) {
    futex_wake(&futex_word, 1);
  }
}
```

Listing 13 Optimised `unlock` function that uses `xchg` instead of `fetch_sub`

We do not show the Promela code for this mutex variant, but it is available in our open source repository as file `drepper_mutex3.pml` [17].

Using `xchg` when unlocking Recall the `unlock` function from Listing 10. The result of the `fetch_sub` operation is used to detect whether the futex word had value 2, in which case it must be explicitly set to 0 and a `futex_wake` call must be issued.

When the `xchg` operation is available, the need for these two steps can be avoided: `xchg` can be used to directly set the futex word to 0, and the old value of the futex word (returned by `xchg`) can be inspected to determine whether a call to `futex_wake` is needed. The code for `unlock` with this simplification is shown in Listing 13.

This optimisation is not suggested in Drepper’s paper, and was brought to our attention by a question from an undergraduate student at Imperial College London during a lecture on futex-based mutexes. It turns out that this optimisation is in fact present in the glibc implementation of `pthread_mutex_unlock` [21, line 174].

Again, we omit the Promela code for this mutex variant, but it is available in our open source repository as file `drepper_mutex3b.pml` [17]; the name reflects the fact that although this optimisation was not proposed in Drepper’s paper, it constitutes a minor change to the optimised mutex implementation presented in the paper.

Checking correctness Our experiments (Section 8) confirm that checking the mutual exclusion property and freedom from invalid end states for these optimised mutex implementations leads to state space sizes of the same order of magnitude as for the unoptimised version, and comparable model checking times.

6 Model Checking Futex-based Mutexes from Gustedt

In a research report [25] and a later article [26], Gustedt presents a futex-based pair of lock-unlock primitives which can be seen as a mutex. While the original motivation was to implement C11’s generic atomics efficiently, Gustedt contributed a second version of his approach to the Musl C standard library [39] as an internal lock-unlock facility.

In this approach the futex word serves two roles. It is both a flag to indicate whether the lock is acquired or not,

```
 1 #define set_locked(VAL)  (0x80000000u | (VAL))
 2 #define is_locked(VAL)   (0x80000000u & (VAL))
 3 class Mutex {
 4 public:
 5   Mutex() : futex_word(0) {}
 6   void lock() {
 7     uint32_t cur = cmpxchg(futex_word, 0, set_locked(1))
       ;
 8     if (cur == 0) return;
 9     cur = futex_word.fetch_add(1) + 1;
10     for (;;) {
11       while (!is_locked(cur)) {
12         uint32_t prev =
13           cmpxchg(futex_word, cur, set_locked(cur));
14         if (prev == cur) return;
15         for (uint32_t i = 0;
16              i < BUSYWAIT && is_locked(cur);
17              i++) {
18           cur = futex_word.load();
19         }
20       }
21       while (is_locked(cur)) {
22         futex_wait(&futex_word, cur);
23         cur = futex_word.load();
24       }
25     }
26   }
27   void unlock() {
28     uint32_t prev = futex_word.fetch_sub(set_locked(1));

29     if (prev != set_locked(1)) {
30       futex_wake(&futex_word, 1);
31     }
32   }
33
34 private:
35   atomic<uint32_t> futex_word;
36 };
```
Listing 14 Gustedt's lock primitive [26] presented as a mutex

and a contention counter keeping track of how many threads have acquired or are trying to acquire the lock. In practice, the high-order bit of the futex word is used as the lock flag, we call it the *lock bit*, while the 31 lower bits are used as the contention counter. Also, for the sake of performance, this approach contains a bounded busy wait loop to enable trying to acquire the lock several times before resorting to a `futex_wait`.

In Section 6.1 and Section 6.2 we study the "research" and "Musl" versions of this approach, respectively.

6.1 Research version

Listing 14 shows an adaptation of Gustedt's lock and unlock primitives [26], presented as a C++ `Mutex` class. Two macros `set_locked` and `is_locked` enable easy setting and checking of the high-order lock bit. Local variable `cur` of the `lock` method reflects the expected value of the futex word. A thread T performs a first `cmpxchg` (line 7) to try to acquire the lock from an expected futex word value of zero, i.e. unlocked and with zero waiter threads. Upon success, `cmpxchg` returns zero and T exits the `lock` method (line 8): T has acquired the lock on the fast path with a single atomic operation. In this case the futex word value is `set_locked(1)`, i.e. the lock bit is set, and the contention counter is one, accounting for T who just acquired the lock.

If the first `cmpxchg` fails, T increments the futex word to add itself to the contention counter, and stores the resulting value in `cur` (line 9). T then enters a loop (line 10) that contains two "while" sub-loops testing whether the latest known value of the futex word, stored in the `cur` local variable, is locked or not.

In the first sub-loop (line 11), while the futex word appears to be unlocked, T attempts to acquire the lock with a `cmpxchg` (line 13), where upon success it would set the futex word lock bit. If the `cmpxchg` is successful, then T has acquired the lock and exits the `lock` method (line 14). Otherwise, T performs a bounded busy wait loop where it reads the futex word up to `BUSYWAIT` times (which in practice has a low value, e.g. 10). If during this busy wait loop the futex word appears to be unlocked, then the first sub-loop starts again (line 11) to try acquiring the lock. Otherwise, T moves on to the second sub-loop.

In the second sub-loop (line 21), while the futex word appears to be locked, T calls `futex_wait` (line 22) to put itself to sleep. When `futex_wait` returns—either because the futex word value did not match T's expected value in `cur`, or because another thread woke up T—then `cur` is updated with the value of the futex word. If this value is locked, then the second sub-loop starts again (line 21). Otherwise, T goes back to the first sub-loop where it tries to acquire the seemingly unlocked futex word.

The `unlock` method is simpler: T starts by subtracting the value `set_locked(1)` from the futex word (line 28), effectively unsetting the lock bit and decrementing the contention counter in one atomic operation. Then, T looks up the previous value of the futex word. If it is `set_locked (1)` then the contention counter value was one, accounting for T only, so there is no thread to wake up. Otherwise T calls `futex_wake` (line 30) to wake up a waiter thread.

Modelling this approach in Promela requires revisiting the atomic increment/decrement operations to properly handle the lock bit, as shown in Listing 15.

Recall the futex word is a byte; we use its high-order bit as the lock bit. We define a series of macros for easy access and manipulation of this bit. When the lock bit is set, the value of the futex word is typically greater than `MAX_BYTE_VALUE`, which would be considered an overflow by our regular `fetch_inc` (Listing 6). We therefore define a new `lockbit_fetch_inc` macro which looks for an overflow on a value where the lock bit is unset. If there is an overflow then the contention counter is set to zero, and lock bit is inverted. This reflects what an overflow on the 31 lowest bits of a C `uint32_t` would do: if the lock bit was zero, it would become one, and vice-versa. As for `fetch_dec`, we directly define a macro named `lockbit`

```
 1 #define lock_bit_mask    (1 << 7)
 2 #define is_locked(w)     (lock_bit_mask & (w))
 3 #define set_locked(w)    (lock_bit_mask | (w))
 4 #define unset_locked(w) ((lock_bit_mask - 1) & (w))
 5
 6 inline lockbit_fetch_inc(location, result) {
 7   d_step{
 8     result = location;
 9     if
10     :: unset_locked(location) == MAX_BYTE_VALUE ->
11        location =
12          (is_locked(location) -> 0 : set_locked(0))
13     :: else -> location = location + 1
14     fi
15   }
16 }
17
18 inline lockbit_fetch_unlock_and_dec(location, result) {
19   d_step{
20     result = location;
21     location = unset_locked(location);
22     location = dec(location)
23   }
24 }
```

Listing 15 Specialised atomic operations handling the lock bit

```
 1 inline lock() {
 2   atomic {
 3     cmpxchg(futex.word, 0, set_locked(1), cur);
 4     if
 5     :: cur == 0 ->
 6        printf("T%d locks mutex on fast path\n", _pid);
 7        goto acquired_mutex
 8     :: else ->
 9        printf("T%d fails to lock mutex on fast path\n",
10               _pid)
11     fi
12   }
13   lockbit_fetch_inc(futex.word, cur);
14   cur = cur + 1;
15
16   retry: // Lock loop
17   if
18   :: !(is_locked(cur)) ->
19      atomic {
20        cmpxchg(futex.word, cur, set_locked(cur), tmp);
21        if
22        :: cur == tmp ->
23           printf("T%d locks mutex\n", _pid);
24           goto acquired_mutex
25        :: else
26        fi
27      }
28      cur = futex.word;
29      do // Busy wait loop
30      :: is_locked(cur) -> cur = futex.word
31      :: true -> goto retry
32      od;
33      goto retry
34   :: else ->
35      futex_wait(futex, cur);
36      cur = futex.word;
37      goto retry
38   fi;
39
40   acquired_mutex: tmp = 0; cur = 0;
41 }
42
43 inline unlock() {
44   d_step {
45     lockbit_fetch_unlock_and_dec(futex.word, prev);
46     printf(
47       "T%d unlocks: set futex word from %d to %d\n",
48       _pid, prev, futex.word);
49   }
50   if
51   :: prev != set_locked(1) -> futex_wake(futex, 1)
52   :: else
53   fi;
54   prev = 0;
55 }
```

Listing 16 Using Promela to model the Gustedt lock primitive of Listing 14

_fetch_unlock_and_dec since this is the only operation required to decrement the value of the futex word.

Equipped with these new helpers, the Promela lock and unlock inline macros for this mutex implementation are shown in Listing 16. These reflect the C++ code of Listing 14, and make use of goto to model the various loops. These macros rely on local variables cur and prev being in scope; these are declared in the root scope of the Thread proctype that models a thread.

The busy wait loop (line 29) is modelled without a loop counter. On each iteration, the loop may do one of its two non-deterministic branches. If cur is locked, then the branch at line 30 updates cur to the futex word value. Regardless of the value of cur, the branch at line 31 breaks the busy wait loop and goes back to starting the lock loop again: this branch models either cur being locked, or the busy wait counter reaching its limit. Note that the guard of this second branch is true, not else: if the guard of the first branch (cur is not locked) holds, either branch can execute. This effectively models a busy wait loop with an arbitrary busy wait limit—and avoids state-space explosion due to the range of values a busy wait counter would take.

Checking correctness As discussed in full in Section 8, we are able to use Spin to successfully verify the mutual exclusion property and freedom from invalid end states for up to 4 threads. Although exploiting symmetry leads to significantly smaller state spaces with 2, 3 and 4 threads, it does not allow verification to scale beyond 4 threads.

Exploring counter overflow The fact that the futex word contains a contention counter raises question about overflow of that counter. Such an overflow would lead to changing the value of the high-order lock bit, and resetting the contention counter value to zero. In our Promela model, this happens in Listing 15 on line 10 when lockbit_fetch_inc is called while the contention counter value is already MAX_BYTE_VALUE.

As an experiment, we can redefine MAX_BYTE_VALUE to be NUM_THREADS - 1, enabling potential overflow of the contention counter. With this change, Spin quickly reports errors and counterexamples illustrating the bugs triggered by the overflow.

In practice, the futex word is 32 bits so that 31 bits are used to store the contention counter. We would thus need 2^{31} threads in contention to have a chance of overflowing the counter, and one can assume it is unrealistic to have such

```
 1 #define set_locked(VAL)    (0x80000000u | (VAL))
 2 #define unset_locked(VAL)  (0x7FFFFFFFu & (VAL))
 3 #define is_locked(VAL)     (0x80000000u & (VAL))
 4 class Mutex {
 5 public:
 6   Mutex() : futex_word(0) {}
 7   void lock() {
 8     uint32_t cur = cmpxchg(futex_word, 0, set_locked(1))
      ;
 9     if (cur == 0) return;
10     for (uint32_t i = 0; i < BUSYWAIT; i++) {
11       if (is_locked(cur)) cur = unset_locked(cur) - 1;
12       uint32_t prev =
13         cmpxchg(futex_word, cur, set_locked(cur+1));
14       if (prev == cur) return;
15       cur = prev;
16     }
17     cur = futex_word.fetch_add(1) + 1;
18     for (;;) {
19       if (is_locked(cur)) {
20         futex_wait(&futex_word, cur);
21         cur = unset_locked(cur) - 1;
22       }
23       uint32_t prev =
24         cmpxchg(futex_word, cur, set_locked(cur));
25       if (prev == cur) return;
26       cur = prev;
27     }
28   }
29   void unlock() {
30     uint32_t prev = futex_word.fetch_sub(set_locked(1));
31     if (prev != set_locked(1)) {
32       futex_wake(&futex_word, 1);
33     }
34   }
35 
36 private:
37   atomic<uint32_t> futex_word;
38 };
```

Listing 17 Musl version of Gustedt approach [39], presented as a mutex

a number of threads contending for the same mutex. As opposed to Section 5.2, the fact that no error is reported when checking with up to four threads gives us strong confidence that there is no way for the contention counter to be overflown by just a few competing threads.

6.2 Musl version

Listing 17 shows a C++ class based on the Musl version of this mutex. This mostly differs from the research one by the busy wait loop, which here is tried at most once during each call to `lock()`.

Similar to the research version, the `lock` method has a `cur` variable that reflects the expected value of the futex word. A thread T entering `lock` starts with a fast path that assumes there is no contention: a `cmpxchg` expects a futex word value of zero (line 8). If successful, this operation sets the futex word to `set_locked(1)`, i.e. the lock bit is set and the contention counter is one, and T returns (line 9).

Otherwise, T enters the busy wait loop (line 10): it tries up to `BUSYWAIT` times to acquire the mutex, whether `cur` has its lock bit set or not. Note that T has not added itself to the contention counter yet, hence at line 11 if `cur` is locked then the expected value of the futex word once it gets unlocked would be `cur` with the lock bit unset, and the contention counter decremented by one since the other thread who released the lock would have remove itself from this counter. Accordingly, the busy wait loop `cmpxchg` (line 13) tries to set the futex word value to its current value plus one to account for T in the contention counter, and also with the lock bit set. If this `cmpxchg` succeeds, then the lock is acquired and T returns. Otherwise, `cur` is updated with the futex word value returned by the `cmpxchg`, and the busy wait loop starts again.

If T does not manage to acquire the lock during the busy wait loop, it continues and will not try the busy wait loop again during this `lock` call. T adds itself to the contention counter by incrementing the futex word value, and stores the result in `cur` (line 17). Next, T enters an unbounded loop (line 18). On each iteration of this loop, T checks whether `cur` is locked. If so, T calls `futex_wait` to put itself to sleep, and when woken up it assumes that the value of the futex word now reflects that an other thread has just released the lock, i.e. this value is `cur` with the lock bit unset, minus one for the contention counter. At line 24, `cur` reflects the best guess T has of the current value of an unlocked futex word, and T tries a `cmpxchg` using this value. Upon success, T has acquired the lock and can return (line 25). Otherwise, `cur` is updated with the futex word value returned by `cmpxchg`, and T proceeds to the next iteration of the unbounded loop.

The `unlock` method is similar to the one in research version: the futex word value is updated to unset its lock bit and decrement its contention counter. If it appears that there were waiter threads, then `futex_wake` is called.

The Promela code for this mutex variant is available in our repository as `gustedt_mutex2.pml` [17].

Checking correctness As with the research version, we are able to use Spin to verify the mutual exclusion property and freedom from invalid end states for up to 4 threads; again, symmetry reduction does not provide enough benefit to allow verification to scale to larger thread counts. Compared to the research version, the Musl version exhibits larger state spaces and associated verification times, while still staying in the same order of magnitude. This illustrates how using the busy wait loop in a different way, and other minor differences with the research version, may impact verification scalability. These experimental results are discussed in full in Section 8.

7 Model Checking Futex-based Condition Variables

A condition variable (`cv`) [29,28] allows inter-thread communication via three operations: `cv_wait`, `cv_signal`

```
bool mutex;
inline mutex_lock() {
  d_step {
    !mutex -> mutex = true
  }
}
inline mutex_unlock() {
  mutex = false
}
```

Listing 18 A simple mutex implementation in Promela

and cv_broadcast. The cv_wait operation takes a mutex as an argument, which must already be locked. It atomically unlocks the mutex and puts the calling thread to sleep. Once the thread is woken up, it locks the mutex again before returning. The cv_signal operation wakes up one thread chosen non-deterministically among the sleeping ones, while cv_broadcast wakes up all sleeping threads.

The cv_wait operation is atomic in the sense that by the time another thread locks the mutex, the first thread is in the list of threads sleeping on the condition variable. In particular, consider a pair of threads T0 and T1; first T0 holds the mutex and calls cv_wait, then T1 locks the mutex and calls cv_signal: the signal from T1 cannot be *lost*, i.e. it must wake up T0.

In the following, we focus on futex-based implementations of cv_wait and cv_signal: the cv_broadcast operation is typically similar to cv_signal save for using INT_MAX instead of 1 as the number of threads to wake up in futex_wake calls.

7.1 Model Checking Harness and Properties

Like for lock and unlock in Section 5.1, our harness makes use of to-be-defined inline macros cv_wait and cv_signal. The harness involves a loop in which threads nondeterministically repeat the process of either waiting or signalling. The harness is designed such that if cv_wait and cv_signal are implemented correctly there always exists a path to successful termination of every thread. In terms of verification, here we pay special attention to make sure the harness can enable catching *lost signal* bugs by checking freedom from invalid end states.

First, in Listing 18 we define a simple mutex: condition variables are use in conjunction with mutexes whose whose internals are irrelevant, so we can use straightforward Promela code to model mutexes. A mutex is represented by a global boolean variable (whose default value if false); locking involves atomically blocking until its value is false and then setting it to true, while unlocking simply involves resetting it to false.

The harness consists of a condition variable used by a single *signaller* thread and one or more *waiter* threads. The

```
active[NUM_WAITERS] proctype Waiter() {
  do
  :: mutex_lock() ->
     num_signals_req++;
     printf("T%d calls cv_wait()\n", _pid);
     cv_wait();
     printf("T%d returns from cv_wait()\n", _pid);
     mutex_unlock()
  :: break
  od;
  num_done++;
}
```

Listing 19 Condition variable harness: waiter threads

```
active proctype Signaller() {
  do
  :: num_signals_req > 0 ->
     mutex_lock();
     printf("T%d must signal, num_signals_req=%d\n",
            _pid, num_signals_req);
     cv_signal();
     num_signals_req--;
     mutex_unlock()
  :: else ->
     if
     :: true ->
        mutex_lock();
        printf("T%d signals without need\n", _pid);
        cv_signal();
        num_signals_req =
          (num_signals_req > 0 -> num_signals_req - 1
                               : 0);
        mutex_unlock()
     :: true ->
        printf("T%d won't signal until needed\n", _pid);
        if
        :: num_signals_req > 0 ->
           assert(num_done < NUM_WAITERS)
        :: num_done == NUM_WAITERS ->
           assert(num_signals_req == 0);
           break
        fi
     fi
  od
}
```

Listing 20 Condition variable harness: signaller thread

waiters call cv_wait an arbitrary number of times before terminating. The signaller calls cv_signal until all waiters are done, then it terminates. In order to catch *lost signal* bugs, we also make sure the signaller has an execution path where cv_signal is called only the necessary number of times to match calls to cv_wait, but no more.

To model all this, we start with a constant representing the number of waiters, and global variables to count the minimum number of signals that are needed and how many threads have terminated, before defining the behaviour of waiter threads via the Waiter proctype of Listing 19.

Each waiter loops on either locking the mutex, incrementing num_signals_req, calling cv_wait and then unlocking the mutex; or exiting the loop and incrementing num_done before terminating. Thus, each waiter may call cv_wait an arbitrary number of times before terminating.

The Signaller proctype of Listing 20, modelling the signaller thread, is slightly more complex. It loops on ei-

```
 1 class CondVar {
 2 public:
 3   CondVar() : futex_word(0) {}
 4   void cv_wait(mutex &m) {
 5     m.unlock();
 6     futex_wait(&futex_word, 0);
 7     m.lock();
 8   }
 9   void cv_signal() { futex_wake(&futex_word, 1); }
10
11 private:
12   atomic<uint32_t> futex_word;
13 };
```

Listing 21 Incorrect futex-based condition variable based on the "Simple but very wrong" example in Denis-Courmont's article [9]

ther detecting that a signal is required (line 3), in which case it locks the mutex, calls `cv_signal` to issue a signal, decrements `num_signals_req` and unlocks the mutex (lines 4–9); or it sees that no signal is required (line 10). In this case, it non-deterministically decides to either call `cv_signal` even though there is no apparent need for it (lines 13–19), or to block until either a signal is needed (line 23), or all waiters are done in which case it breaks out of the loop to terminate (line 25). The `if` branches starting with `true` (lines 12 and 20) model the "internal" decision of the signaller. In particular, once it has decided to block, it must not signal again unless it detects the need for a signal.

On the one hand, this harness allows the signaller to produce an arbitrary number of signals, even if no waiter is currently waiting for a signal. On the other hand—and this is crucial to detect lost signal bugs—when the signaller sees that no signal is needed, it may decide to stop signalling until either a signal is needed, or all waiters are done. This ensures that each call to `cv_wait` is matched by at least one call to `cv_signal`, but potentially no more than the number of signals that are strictly needed. In the execution path where there is only one signal per wait, if any signal is lost this will lead to a scenario where (a) some waiter is stuck in the `cv_wait` call at line 6, and (b) the signaller is blocked at line 23 because no signals are currently required. Thus the lost signal will lead to the model checker reporting an invalid end state.

The rest of this section covers various futex-based implementations of `cv_wait` and `cv_signal`, as presented by Denis-Courmont [9]. Each implementation depends on the declaration of a single futex global variable named `futex`.

7.2 Take 1: Naive and Incorrect

The C++ class of Listing 21 represents a naive attempt at a condition variable implementation, from the "Simple but very wrong" section in Denis-Courmont's article [9].

The `cv_wait` operation unlocks the mutex before calling `futex_wait` with a plain value of 0 (the initial value

```
1 inline cv_wait() {
2   mutex_unlock();
3   futex_wait(futex, 0);
4   mutex_lock()
5 }
6
7 inline cv_signal() {
8   futex_wake(futex, 1)
9 }
```

Listing 22 Modelling the incorrect condition variable of Listing 21 in Promela

```
 1 class CondVar {
 2 public:
 3   CondVar() : futex_word(0) {}
 4   void cv_wait(mutex &m) {
 5     futex_word.store(1);
 6     m.unlock();
 7     futex_wait(&futex_word, 1);
 8     m.lock();
 9   }
10   void cv_signal() {
11     futex_word.store(0);
12     futex_wake(&futex_word, 1);
13   }
14
15 private:
16   atomic<uint32_t> futex_word;
17 };
```

Listing 23 Incorrect futex-based condition variable based on the "Toggle" example in Denis-Courmont's article [9]

of the futex word) to put the thread to sleep. On waking, it locks the mutex again before returning. The `cv_signal` operation just calls `futex_wake` to wake up one of the sleeping threads. This is modelled in Promela using the inline macros of Listing 22.

Invoking Spin on the harness with this version leads to an invalid end state error, with a counterexample illustrating the issue. After the mutex is unlocked in `cv_wait` (line 2), the signaller thread might call `cv_signal`, which in turn calls `futex_wake`, before the waiter calls `futex_wait` (line 3); the signal is lost. In this case, if the signaller decides to block until another signal is needed, then the waiter thread has no chance to be woken up: the system is in deadlock.

7.3 Take 2: Toggle State

Listing 23 shows a second take, from the "Toggle" section of Denis-Courmont's article [9], which involves using the futex word as a toggle state, oscillating between values 0 and 1, to avoid lost signals. In `cv_wait`, the futex word is set to 1 before releasing the mutex and calling `futex_wait` with a value of 1. In `cv_signal`, the value of the futex word is set to 0 before calling `futex_wake`. This toggling is meant to avoid losing the signal in the counterexample considered in Section 7.2: if a thread T is between lines 6 and 7 when another thread calls `cv_signal`, then the futex

```
inline cv_wait() {
  futex.word = 1;
  mutex_unlock();
  futex_wait(futex, 1);
  mutex_lock();
}

inline cv_signal() {
  futex.word = 0;
  futex_wake(futex, 1);
}
```

Listing 24 Modelling the incorrect condition variable of Listing 23 in Promela

word is set to 0, such that T will not sleep when it eventually calls `futex_wait`.

This approach is transcribed to the Promela code of Listing 24. Using Spin on this approach confirms that there is no issue when only two threads are considered. However, as soon as we have three threads, Spin outputs a counterexample illustrating a lost signal:

Thread	Message
T1	calls cv_wait()
T2	must signal, num_signals_req=1
T2	woke up 0 thread(s)
T0	calls cv_wait()
T1	futex_wait, value match: 1; sleep
T2	must signal, num_signals_req=1
T0	futex_wait, value match: 1; sleep
T2	wakes T0
T2	woke up 1 thread(s)
T2	won't signal until needed
T0	has woken
T0	returns from cv_wait()

Here, T0 and T1 are waiters, and T2 is the signaller. This counterexample starts like the lost signal counterexample we saw before. First, T1 calls `cv_wait`: it sets the futex word to 1 and unlocks the mutex. At this point, T2 signals: it sets the futex word to 0 and calls `futex_wake` but wakes up no thread. This is because T1 has not called `futex_wait` yet. Then, T0 calls `cv_wait`, so it sets the futex word to 1 again. At this point, both T1 then T0 call `futex_wait`, and thus go to sleep. Then, T2 signals and wakes up one of the sleeping threads, T0. At this point, we reach a deadlock: T1 is sleeping waiting for a signal operation to wake it up, however T2 does not have to signal anymore: it has signalled after each of the `cv_wait` calls issued so far. The first signal was lost: the corresponding toggled futex word value was overwritten by T0.

We note that Denis-Courmont first thought this approach was correct, before being pointed to a counterexample by a reader of the article. Our modelling and analysis show that using model checking at the design stage could lead to quick discovery of such issues.

```
1  class CondVar {
2  public:
3    CondVar() : futex_word(0) {}
4    void cv_wait(mutex &m) {
5      uint32_t old_value = futex_word;
6      m.unlock();
7      futex_wait(&futex_word, old_value);
8      m.lock();
9    }
10   void cv_signal() {
11     futex_word.fetch_add(1);
12     futex_wake(&futex_word, 1);
13   }
14
15 private:
16   atomic<uint32_t> futex_word;
17 };
```

Listing 25 Futex-based condition variable based on the "Sequence counter, close but no cigar" example in Denis-Courmont's article, which suffers from a potential deadlock [9]

```
1  inline cv_wait() {
2    val = futex.word;
3    mutex_unlock();
4    futex_wait(futex, val);
5    mutex_lock();
6    // Reset to avoid state space explosion
7    val = 0;
8  }
9
10 inline cv_signal() {
11   futex.word = inc(futex.word);
12   futex_wake(futex, 1);
13 }
```

Listing 26 Modelling the condition variable of Listing 25 in Promela

7.4 Take 3: Bionic, Unlikely Yet Possible Deadlock

Our third take on implementing condition variables, which Denis-Courmont entitles "Sequence counter, close but no cigar" [9], mimics the approach taken in Android's Bionic libc [1], where `cv_signal` increments the futex word to avoid deadlocks seen in our first take (Section 7.2). This is illustrated by the C++ class of Listing 25.

In `cv_wait`, the value of the futex word is saved in `old_value` before the mutex is released, after which a call to `futex_wait` with `old_value` is made.

In `cv_signal`, the futex word is incremented by 1, with a possible overflow, before calling `futex_wake`. This avoids the deadlock situation encountered in Section 7.2: if `cv_signal` is executed between unlocking the mutex (line 6) and calling `futex_wait` (line 7) in `cv_wait`, the futex word value will be different from the value used in the call to `futex_wait` which thus will not block.

This approach is modelled in Promela via the code of Listing 26, which assume that a local variable `val` has been declared at the root scope of the `Waiter` proctype. However, Spin still reports a possible deadlock: if between lines 3 and 4, `cv_signal` is called enough times to overflow the futex word and bring it back to the old value saved in line 2,

```
 1 class CondVar {
 2 public:
 3   CondVar() : futex_word(0) {}
 4   void cv_wait(mutex &m) {
 5     previous.store(futex_word);
 6     uint32_t val = previous.load();
 7     m.unlock();
 8     futex_wait(&futex_word, val);
 9     m.lock();
10   }
11   void cv_signal() {
12     uint32_t val = 1 + previous.load();
13     futex_word.store(val);
14     futex_wake(&futex_word, 1);
15   }
16 
17 private:
18   atomic<uint32_t> futex_word;
19   atomic<uint32_t> previous; // Additional state
20 };
```

Listing 27 Futex-based condition variable based on the "Back to sequence counting" example in Denis-Courmont's article, which still suffers from a potential deadlock [9]

```
 1 byte cv_previous; // Additional state
 2 
 3 inline cv_wait() {
 4   cv_previous = futex.word;
 5   val = cv_previous;
 6   mutex_unlock();
 7   futex_wait(futex, val);
 8   mutex_lock();
 9   // Reset to avoid state space explosion
10   val = 0;
11 }
12 
13 inline cv_signal() {
14   futex.word = inc(cv_previous);
15   printf("T%d sets futex.word to %d\n",
16          _pid, futex.word);
17   futex_wake(futex, 1);
18 }
```

Listing 28 Modelling the condition variable of Listing 27 in Promela

then the call to `futex_wait` does block, and we reach a deadlock. This issue is documented in Bionic, with an acknowledgement that it would be extremely unlikely to arise in practice: with a 32-bit futex word, we would need *exactly* 2^{32} calls to `cv_signal` in a row, at the moment when `cv_wait` is between lines 3 and 4, to trigger the deadlock.

Such issues are hard to foresee at design time. Model checking is valuable in illustrating rare risks of deadlocks, so that their practical acceptability can be evaluated.

7.5 Take 4: Even Rarer Yet Still Possible Deadlock

A fourth take on condition variables, entitled "Back to sequence counting" in Denis-Courmont's article [9], adds an extra piece of state to represent the condition variable, which consisted only of a futex so far. This extra `previous` variable stores the expected previous value of the futex word. This approach is captured by the C++ class of Listing 27.

We model this variation in Promela via the inline macros of Listing 28, which again requires local variable `val` to be in scope. Global variable `cv_previous` is used to model the additional state. When a thread calls `cv_wait`, it updates `cv_previous` to the current value of the futex word. This value is also stored in the `val` local variable before unlocking the mutex, and then `futex_wait` is called with this locally stored value. In `cv_signal`, the futex word is set to `cv_previous` incremented by one, before calling `futex_wait`.

Here, the idea is to avoid lost signals by modifying the value of the futex word in `cv_signal`, similarly to the approach in Section 7.4, yet to try and avoid the deadlock triggered by overflowing the futex word value. To this end, in `cv_signal` the futex word is updated to the value of `cv_previous` plus one (line 14), such that several consecutive calls to `cv_signal` all lead to the same value in futex word. For the futex word to keep on incrementing, lock-step pairs of calls to `cv_wait` (to set `cv_previous` to the futex word value) and `cv_signal` (to set the futex word value to `cv_previous` plus one) are required.

Denis-Courmont argues that here, an overflow-induced deadlock can only occur if while thread T is between unlocking the mutex (line 6) and calling `futex_wait` (line 7), 2^{32} other threads call `cv_wait` *concurrently*, and at least one other thread calls `cv_signal` concurrently between each `cv_wait`, such that the futex word overflows and that the `futex_wake` call of thread T puts thread T to sleep, never to be woken up [9].

However, Spin quickly finds a counterexample showing that such a deadlock can happen with only three threads. If thread T0 is between line 6 and 7, then we can have two threads T1 and T2 calling `cv_wait` and `cv_signal` in lock-step pairs enough times to overflow the futex word. This scenario is acknowledged by Denis-Courmont in a subsequent article [10]. Note that in practice, for a 32-bit futex word, this would require exactly 2^{32} *lock-step pairs of calls* to `cv_wait` and `cv_signal` while another thread has unlocked the mutex and is about to call `futex_wait`: this makes the probability of this deadlock even lower than the one in Section 7.4.

Despite the elaborate sequence required to trigger a deadlock, model checking is once again able to find a counterexample that is valuable for designers to assess the relevance of a solution.

8 Experiments

For the incorrect futex-based mutex discussed in Drepper's article (Section 5.2), and the various attempts at implementing condition variables (Section 7), we have demonstrated that model checking can be useful in quickly providing counterexamples to correctness. We now turn to the scalability of model checking when it comes to verifying the correctness of the mutex implementations that are not thought to suffer from correctness issues: the correct mutex from Drepper's article (Section 5.3) and its optimised variants (Section 5.4), and the two mutex implementations from Gustedt (Section 6). For these mutexes, we investigate how model checking scales as increasing numbers of threads contend for the mutex. Because these examples are inherently symmetric, we investigate the potential of *symmetry reduction* to allow model checking to scale to larger thread counts. We also investigate the effectiveness of two lossless memory reduction techniques that Spin provides.

8.1 Experimental Setup

Our experiments were executed on an AMD EPYC workstation running Linux 6.9.10, with C code generated by Spin compiled using GCC 13.2.0 with optimisation flag "`-O3`". Each run is limited to 100 GB of memory and six hours of run time.

For experiments without symmetry reduction, Spin version 6.5.2 was used.

For experiments with symmetry reduction, Spin version 6.1.0 was used, together with Git revision 6d26219 of the TopSPIN symmetry reduction tool [12,11]. This is because (as discussed in Section 8.2 below) TopSPIN is not compatible with more recent versions of Spin. TopSPIN relies on the computational group theory package GAP [24], with GAP version 4.13.1 being used in our experiments.

The times that we present are averages taken over 3 runs, and overall we observed a coefficient of variation of less than 10%.

8.2 Changes to Support Symmetry Reduction

Symmetry reduction is a state-space reduction technique that can be used when a model features many replicated components [8,16,41,38]. In the simplest case—which applies in the context of our futex-based mutex examples—a model comprises multiple identical processes and is *fully symmetric*. In a fully symmetric model with N processes, a model state has the form $(L_1, L_2, \dots, L_N, G)$, where L_i represents the local state of process i ($1 \leq i \leq N$), and G represents the state of any global variables. In our examples, each local state L_i would capture the program counter and local variables of a thread contending for the mutex, while G would capture global state such as the futex word and the array recording which threads are waiting.

Two states $(L_1, L_2, \dots, L_N, G)$ and $(M_1, M_2, \dots, M_N, G')$ are *equivalent* under symmetry if $(M_1, M_2, \dots, M_N)$ is a permutation of $(L_1, L_2, \dots, L_N)$—i.e. it can be obtained by shuffling the L_i—and G' is the result of applying this permutation to the state of the global variables captured by G (e.g., in our context, shuffling the values of the array that records which threads are waiting). It can be shown that if the property ϕ under analysis is invariant under symmetry (that is, it does not depend on the identity of any particular process), then ϕ holds in state s if and only if it holds in state s' where s and s' are equivalent under symmetry [8, 16]. It thus suffices to check just one state per symmetric equivalence class. With N processes, a symmetric equivalence class can be as large as $N!$, thus symmetry reduction has the potential to substantially reduce the number of states that need to be explored as N increases. The main challenge associated with putting symmetry reduction into practice (explored in the literature and described in a pair of survey articles [41,38]) involves efficiently detecting equivalence between states.

Our mutex implementations are fully symmetric, and the properties of freedom from invalid end and mutual exclusion are invariant under symmetry, so these models are good candidates for symmetry reduction. To experiment with symmetry reduction, we use the TopSPIN tool [12]. TopSPIN performs a static analysis of Promela source code to automatically identify symmetry [14], and then uses techniques from computational group theory to automatically classify the group of identified symmetries and determine a strategy for exploiting symmetry during model checking [13].

We found that TopSPIN is not compatible with recent versions of Spin, but remains compatible with versions of Spin up to Spin 6.1.0. Therefore, as discussed in Section 8.1, we Spin 6.1.0 for our symmetry reduction experiments.

We had to create alternative versions of our Promela models to cater for a number of restrictions of the TopSPIN tool, as follows:

No active proctypes TopSPIN does not support Promela's concept of active proctypes, so our alternative models use an `init` process that launches further processes via `run` statements in an `atomic` scope. This causes a tiny increase in state space size—a single extra state—because it takes one step for the `init` process to launch the other processes.

Changes to arrays TopSPIN requires that any array that is indexed by a process ID must have a length equal to the total number of processes, regardless of type. In our case, this includes the `init` process and the safety monitor process. Our alternative versions thus use larger arrays, and array indexing operations are adapted to account for the fact that the `init` process has process ID 0, so that the IDs of `Thread` processes start from 1. This slightly enlarges the state vector associated with each state of the model.

Removal of certain `printf` statements Our models feature `printf` statements that include literal values corresponding to process IDs. The type inference behind TopSPIN's symmetry detection strategy cannot detect that these literals should be treated like process IDs, rather than as regular integer values. This leads to a failure to detect symmetry. We thus commented out such `printf` statements (which do not contribute to the number of reachable states) in our alternative versions.

By performing a number of model checking runs without symmetry reduction, we confirmed that these changes lead to alternative models with state space sizes identical to those of the original models, save for the additional state caused by the use of an `init` process.

8.3 Experimental Results

Results without symmetry reduction State space sizes and verification times (in seconds) for model checking the correct mutex implementations with varying numbers of threads are shown in Table 1. These results are for Spin working in its default safety verification mode, where partial order reduction is enabled and no state compression is used. Symmetry reduction is not employed.

The “Drepper” column show results for Drepper’s correct mutex implementation (see Section 5.3). The “Drepper (opt. lock)” and “Drepper (opt. lock/unlock)” columns show results for the versions of this implementation where the `lock` function, and both the `lock` and `unlock` functions, are optimised respectively (see Section 5.4). The “Gustedt (research)” column shows results for the research version of Gustedt’s mutex implementation (see Section 6.1), while the “Gustedt (Musl)” column shows results for the Musl version of this mutex (see Section 6.2).

For Drepper’s mutex and its variants, while the size of the state space grows rapidly as the thread count increases, verification succeeds comfortably (within a matter of minutes) for up to 6 threads. Available memory resources are exceeded when verification is attempted with 7 threads (hence why only results for up to 6 threads are shown in the table). The optimised versions of this mutex implementation exhibit somewhat larger state spaces, but within the same order of magnitude.

Verification scales less well for the Gustedt mutexes, which suffer from more serious state-space explosion. For the research version, the state space size increases by a factor of more than 270 when moving from 2 to 3 threads, and by *another* factor of more than 587 when moving from 3 to 4 threads. These factors are even larger for the Musl version, with a growth factor of more than 441 when moving from 2 to 3 threads, and more than 696 when moving from 3 to 4 threads. For both mutexes, verification with more than 4 threads exceeded available memory resources, indicated by a ‘-’ entry in the table.

We attribute the rapid state space growth for the Gustedt mutexes, compared with the Drepper mutexes, to the presence of a busy wait loop, and the number of values the futex word might have: in Gustedt, the futex word stores both a lock bit and a contention counter, whereas in Drepper correct versions, the futex word can only have three values (0, 1 or 2).

Results with symmetry reduction Results for model checking when TopSPIN is used to provide symmetry reduction are shown in Table 2. The main take-away from comparing Table 2 (with symmetry reduction) with Table 1 (no symmetry reduction) is that symmetry reduction is very useful for scaling the analysis of Drepper’s mutex and its variants to larger thread counts, but not useful in this regard for the Gustedt mutexes.

For the Drepper mutexes, comparing state space sizes with and without symmetry reduction shows symmetry reduction leads to a reduction factor of $420\times$ for configurations involving 6 threads. Symmetry reduction enables verifying configurations of these mutexes with up to 12 threads before available memory resources are exhausted.

Symmetry reduction also leads to a substantial reduction in state space size for the Gustedt mutexes, offering a reduction of more than $20\times$ for configurations of these mutexes involving 4 threads. However, the incredibly rapid state space explosion exhibited by these models means that even with the help of symmetry reduction, verification does not scale beyond 4 threads before exhausting available memory resources.

From the timing results we can see that while symmetry reduction leads to significantly smaller state space, it also induces a per-state overhead computation: for example, verifying “Drepper” with 6 threads without symmetry reduction results in a throughput of 362,119 states per seconds, while with symmetry reduction it is down to 209,407 states per seconds. Considering “Drepper” with symmetry reduction with 12 threads, throughput reduces further to 24,788 states per second. This is because configurations with more threads have larger symmetry groups (hence the bigger state space reduction factor), but the larger group means that more computation is required per state in order for symmetric equivalence between states to be detected.

Effectiveness of two memory reduction techniques We investigate whether two memory reduction techniques provided by Spin can be useful in improving the scalability of model checking: *collapse* compression, which stores states in a compressed form [30, Chapter 9], and the *minimised automaton* method, which uses an alternative data structure to represent the state space in a compact manner [32]. In contrast to some other memory reduction techniques that Spin

Table 1 State space sizes and model checking times (in seconds) for verifying correct mutex implementations with various thread counts, without symmetry reduction. A '-' entry indicates that verification could not be completed within available memory resources, which was also the case for configurations involving 7 threads.

	Drepper		**Drepper (opt. lock)**		**Drepper (opt. lock/unlock)**		**Gustedt (research)**		**Gustedt (Musl)**	
Threads	States	Time	States	Time	States	Time	States	Time	States	Time
2	268	0.0	351	0.0	320	0.0	1,051	0.0	1,087	0.0
3	9,494	0.0	10,913	0.0	9,923	0.0	284,431	0.1	479,692	0.2
4	254,228	0.2	292,693	0.2	274,152	0.2	167,239,830	217.0	334,107,290	374.0
5	6,006,939	8.0	7,222,655	7.8	7,053,427	7.9	-	-	-	-
6	132,426,930	365.7	168,134,930	364.0	170,710,110	358.3	-	-	-	-

Table 2 State space sizes and model checking times (in seconds) for verifying correct mutex implementations with various thread counts, with symmetry reduction. A '-' entry indicates that verification could not be completed within available memory resources, which was also the case for configurations involving 13 threads.

	Drepper		**Drepper (opt. lock)**		**Drepper (opt. lock/unlock)**		**Gustedt (research)**		**Gustedt (Musl)**	
Threads	States	Time	States	Time	States	Time	States	Time	States	Time
2	136	0.0	182	0.0	168	0.0	537	0.0	531	0.0
3	1,690	0.0	1,834	0.0	1,793	0.0	48,799	0.0	79,092	0.0
4	12,549	0.0	14,289	0.0	13,805	0.0	7,115,838	13.6	13,256,896	21.9
5	69,464	0.2	84,116	0.2	79,839	0.2	-	-	-	-
6	314,111	1.5	394,260	1.6	385,565	1.5	-	-	-	-
7	1,219,936	9.2	1,599,879	9.7	1,598,859	9.6	-	-	-	-
8	4,205,948	47.2	5,739,971	49.0	5,785,936	48.2	-	-	-	-
9	13,158,276	199.3	18,631,188	208.0	18,951,940	207.0	-	-	-	-
10	37,966,278	759.7	55,546,516	802.0	56,893,240	805.7	-	-	-	-
11	102,252,580	2,633.3	154,248,840	2,816.7	158,733,570	2,786.7	-	-	-	-
12	259,450,920	10,466.7	401,739,510	10,933.3	414,483,280	11,000.0	-	-	-	-

supports, both of these methods are *lossless*: they do not lead to any states being missed.

A summary of our findings is shown in Table 3, where data are taken for all runs, both with and without symmetry reduction, where full verification was possible. We did not find any cases where memory reduction led to more tractable verification; i.e. there were no cases where a memory reduction strategy made full verification possible (within our resource limits) when it was not possible without a memory reduction strategy.

The "Space saving (total)" row shows that, on average, the space savings afforded by these two reduction techniques are very modest when one looks at the total memory associated with model checking. This is because in all model checking runs we executed Spin with a very large search depth, so that depth-first search would be possible on the large state spaces associated with high thread counts. The *collapse* and *minimised automaton* techniques do not affect the space required for the depth-first search stack, and so the memory required for this stack dwarfs the savings that these approaches bring. The median space saving is particularly low because we use an unnecessarily large stack size even for small thread counts. However, a large amount of stack space was needed for larger thread counts.

If we ignore the amount of memory devoted to storing the depth-first search stack, we see that the memory reduction techniques have a more pronounced impact on the remaining memory that is used. This is illustrated by the "Space saving (no DFS stack)" row, with mean space savings of more than 20% for *collapse* and more than 70% for *minimised automaton*, respectively.

However, these savings do not come for free: the final row of the table, "Time overhead" shows the increase in model checking time, on average, associated with using these techniques. The *collapse* technique incurs a relatively modest performance overhead, but verification using the *minimised automaton* approach is several times slower.

In summary: in this domain, the deep nature of the state spaces mean that these memory reduction techniques are not effective if one wishes to perform full depth-first search-based verification, because full verification requires a large amount of memory for the depth-first search stack, which is not reduced by either technique.

9 Related Work

There is a significant literature on formal verification of interprocess communication primitives. Bogunovic *et al.* verified mutual exclusion algorithms with SMV [5], with an analysis of liveness and fairness. Mateescu and Serwe analysed 27 different shared-memory mutual exclusion protocols with CADP, assessing both correctness and performance [35, 36]. Bar-David and Taubenfeld used model checking techniques to automatically discover mutual exclusion algorithms [2].

Table 3 Data showing the average percentage savings afforded by memory reduction using *collapse* compression and the *minimised automaton* approach, together with data on the performance overhead associated with the approaches.

	Collapse		*Minimised automaton*	
	Mean	Median	Mean	Median
Space saving (total)	2.31%	0.03%	4.14%	0.48%
Space saving (no DFS stack)	20.33%	25.31%	71.21%	68.27%
Time overhead	1.64×	1.29×	8.64×	5.23×

More recently, Kokologiannakis and Vafeiadis developed a specific dynamic partial order reduction (DPOR) technique to better handle the barrier synchronisation primitive [34]. In terms of using model checking for education, Hamberg and Vaandrager wrote about their experience using UPPAAL in a course on operating systems [27].

We are not aware of formal verification of futex-based synchronisation primitives. Futexes are primarily a Linux system call [22,23]. Besides the two reference publications from Franke *et al.* [20] and Drepper [15], Benderski wrote a good introduction on the topic [3]. Note that the futex system call itself has suffered from bugs that affected userspace applications, such as the Java Virtual Machine [37].

In our evaluation we considered the application of symmetry reduction, which has received significant attention in the field of model checking since it was first proposed [8, 16], and is the subject of two survey articles [41,38]. A feature of the TopSPIN tool that we have used for symmetry reduction is that it detects symmetry in an automated manner, but as discussed in Section 8.2 this does put some constraints on how a model must be expressed in order for the detection algorithm to succeed, and these constraints can lead to a slight waste in resources when working with PID-indexed arrays (arrays that are indexed by process ID). An alternative symmetry reduction tool for Spin, called SymmSpin [6], requires the presence of symmetry to be identified in an upfront manner through the use of a *scalarset* data type, which is based on prior work on symmetry reduction for the Murphi verification system [33]. While requiring manual type annotations, the use of scalarsets to identify particular types of processes that should be regarded as symmetric has the potential to avoid the issues related to PID-indexed arrays that affect TopSPIN. However, our understanding is that the SymmSpin tool is no longer maintained.

10 Future Directions

We have presented a case study of modelling a series of futex-based implementations of mutexes and condition variables in Promela, and using Spin to verify safety properties. An immediate extension would be to consider fairness to enable verifying liveness properties, like the absence of starvation. We can also explore additional futex-based synchronisation primitives, for instance barriers.

To create an educational resource that would require little model checking expertise, we can think of doing verification directly on a C implementation by using a model checker that targets C code, such as CBMC [7] or CPAchecker [4]. We can even envision extracting C models from various C standard library implementations (e.g. glibc), to verify designs actually used in widespread libraries. Finally, it would be interesting to verify the implementation of the futex system call implementation itself in the Linux kernel and other OSes that have adopted futexes (e.g. OpenBSD).

References

1. Android. Bionic C library, pthread_cond implementation, 2023. https://android.googlesource.com/platform/bionic/+/refs/tags/android-13.0.0_r24/libc/bionic/pthread_cond.cpp, last accessed 2025-01-15.
2. Yoah Bar-David and Gadi Taubenfeld. Automatic discovery of mutual exclusion algorithms. In Faith Ellen Fich, editor, *Distributed Computing, 17th International Conference, DISC 2003, Sorrento, Italy, October 1-3, 2003, Proceedings*, volume 2848 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2003.
3. Eli Benderski. Basics of futexes, 2018. https://eli.thegreenplace.net/2018/basics-of-futexes/, last accessed 2025-01-15.
4. Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.
5. Nikola Bogunovic and Edgar Pek. Verification of mutual exclusion algorithms with SMV system. In *The IEEE Region 8 EUROCON 2003. Computer as a Tool.*, volume 2, pages 21–25. IEEE, 2003.
6. Dragan Bosnacki, Dennis Dams, and Leszek Holenderski. Symmetric Spin. *International Journal on Software Tools for Technology Transfer*, 4(1):92–106, 2002.
7. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
8. Edmund M. Clarke, Somesh Jha, Reinhard Enders, and Thomas Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.
9. Rémi Denis-Courmont. Condition variable with futex, 2020. https://www.remlab.net/op/futex-condvar.shtml, last accessed 2025-01-15.
10. Rémi Denis-Courmont. Other uses of futex, 2020. https://www.remlab.net/op/futex-misc.shtml, last accessed 2025-01-15.
11. Alastair F. Donaldson. Symmetry tools, 2024. https://github.com/afd/symmetrytools, last accessed 2025-01-15.

12. Alastair F. Donaldson and Alice Miller. A computational group theoretic symmetry reduction package for the Spin model checker. In Michael Johnson and Varmo Vene, editors, *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006, Proceedings*, volume 4019 of *Lecture Notes in Computer Science*, pages 374–380. Springer, 2006.
13. Alastair F. Donaldson and Alice Miller. Exact and approximate strategies for symmetry reduction in model checking. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 541–556. Springer, 2006.
14. Alastair F. Donaldson and Alice Miller. Automatic symmetry detection for promela. *J. Autom. Reason.*, 41(3-4):251–293, 2008.
15. Ulrich Drepper. Futexes are tricky, 2011. https://www.akkadia.org/drepper/futex.pdf, last accessed 2025-01-15.
16. E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods Syst. Des.*, 9(1/2):105–131, 1996.
17. Hugues Evrard and Alastair Donaldson. Model checking futexes: Code examples, 2022. https://github.com/mc-imperial/modelcheckingfutexes, last accessed 2025-01-15.
18. Hugues Evrard and Alastair F. Donaldson. Model checking futexes. In Georgiana Caltais and Christian Schilling, editors, *Model Checking Software - 29th International Symposium, SPIN 2023, Paris, France, April 26-27, 2023, Proceedings*, volume 13872 of *Lecture Notes in Computer Science*, pages 41–58. Springer, 2023.
19. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005.
20. Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium 2002*, pages 479–495, 2002. https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf, last accessed 2025-01-15.
21. Free Software Foundation, Inc. The GNU C library (glibc), version 2.38, source/nptl/pthread_mutex_unlock.c, 2023. https://www.gnu.org/software/libc/sources.html, last accessed 2025-01-15.
22. Futex manual page section 2 (system calls), 2023. https://man7.org/linux/man-pages/man2/futex.2.html, last accessed 2025-01-15.
23. Futex manual page section 7 (miscellaneous), 2023. https://man7.org/linux/man-pages/man7/futex.7.html, last accessed 2025-01-15.
24. The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.13.1*, 2024. https://www.gap-system.org, last accessed 2025-01-15.
25. Jens Gustedt. Futex based locks for C11's generic atomics. Research Report RR-8818, INRIA Nancy, December 2015.
26. Jens Gustedt. Futex based locks for C11's generic atomics (extended abstract). In *The 31st Annual ACM Symposium on Applied Computing*, Pisa, Italy, April 2016.
27. Roelof Hamberg and Frits Vaandrager. Using model checkers in an introductory course on operating systems. *ACM SIGOPS Operating Systems Review*, 42(6):101–111, 2008.
28. Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, 1975.
29. Charles Antony Richard Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
30. Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 1st edition, 2003.
31. Gerard J. Holzmann and Doron A. Peled. An improvement in formal verification. In Dieter Hogrefe and Stefan Leue, editors, *Formal Description Techniques VII, Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques, Berne, Switzerland, 1994*, volume 6 of *IFIP Conference Proceedings*, pages 197–211. Chapman & Hall, 1994.
32. Gerard J. Holzmann and Anuj Puri. A minimized automaton representation of reachable states. *International Journal on Software Tools for Technology Transfer*, 2(3):270–278, 1999.
33. C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.
34. Michalis Kokologiannakis and Viktor Vafeiadis. Bam: Efficient model checking for barriers. In *International Conference on Networked Systems*, pages 223–239. Springer, 2021.
35. Radu Mateescu and Wendelin Serwe. A study of shared-memory mutual exclusion protocols using CADP. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 180–197. Springer, 2010.
36. Radu Mateescu and Wendelin Serwe. Model checking and performance evaluation with CADP illustrated on shared-memory mutual exclusion protocols. *Science of Computer Programming*, 78(7):843–861, 2012.
37. Mechanical Sympathy Google group. Linux futex_wait() bug, 2015. https://groups.google.com/g/mechanical-sympathy/c/QbmpZxp6C64, last accessed 2025-01-15.
38. Alice Miller, Alastair F. Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Computing Surveys*, 38(3):8, 2006.
39. Musl. Musl C library, internal lock implementation, 2023. https://git.musl-libc.org/cgit/musl/tree/src/thread/__lock.c?id=47d0bcd4762f223364e5b58d5a381aaa0cbd7c38, last accessed 2025-01-15.
40. Theo C. Ruys. Low-fat recipes for SPIN. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*, volume 1885 of *Lecture Notes in Computer Science*, pages 287–321. Springer, 2000.
41. Thomas Wahl and Alastair F. Donaldson. Replication and abstraction: Symmetry in automated formal verification. *Symmetry*, 2(2):799–847, 2010.