Model Checking the Java Meta-Locking Algorithm

Samik Basu and Scott A. Smolka (Contact Author)  
Department of Computer Science  
State University of New York at Stony Brook  
Stony Brook, NY 11794-4400 USA  
{bsamik, sas}@cs.sunysb.edu

Abstract

We apply the XMC tabled-logic-programming-based model checker to the Java meta-locking algorithm. Meta-locking is a highly optimized technique for ensuring mutually exclusive access by threads to object monitor queues, and therefore plays an essential role in allowing Java to offer concurrent access to objects. Our abstract specification of the meta-locking algorithm is fully parameterized, both on the number of threads $M$, and the number of objects $N$. It also captures a sophisticated optimization of the basic meta-locking algorithm known as extra-fast locking and unlocking of uncontented objects. Using XMC, we show that for a variety of values of $M$ and $N$, the algorithm indeed provides mutual exclusion and freedom from deadlock and lockout. Collectively, our results provide strong testimony to the correctness of the meta-locking algorithm.

Subject Descriptors: D.1.3 Concurrent Programming; D.2.4 Software/Program Verification

General Terms: Verification

Additional Keywords: Model checking, Java, XMC, meta-locking, monitor queues, mutual exclusion, synchronized methods.

1 Introduction

Given the growing importance of Java™ as a concurrent object-oriented language for implementing internet-based applications, it is crucial that the language’s implementation on a Java virtual machine (JVM) be both efficient and correct. This is especially true of the language’s synchronization operations: in Java, every object is capable of providing mutually exclusive access to its data via synchronized methods and synchronized statements. Moreover, most Java programs synchronize extremely frequently, as standard class libraries, including commonly deployed data types such as vectors and buffers, have been designed for multi-threaded use. For example, measurements re-
ported in [ADG+99] show that the SPECjvm98 version of the javac source-to-bytecode compiler executes 765,000 synchronization operations per second.

To address this state of affairs, Agesen et al. [ADG+99] have proposed an efficient meta-locking algorithm for implementing the synchronization operations of Java. Meta-locking can be viewed as a two-tiered scheme for achieving monitor-style synchronization in objects. In particular, meta-locks provide mutually exclusive access by threads to an object's synchronization data, which is essentially a (possibly non-FIFO) queue of lock records. Each lock record represents a thread waiting to enter the object's monitor. A thread gains exclusive access to the monitor when its lock record reaches the head of the queue; at this point, the monitor is locked until the thread exits the monitor.

Two distinguishing features of the meta-locking scheme are its low space overhead (two bits per object) and fast execution time (lock + unlock executes in 11 SPARC instructions when there is no meta-lock contention). Moreover, it does not rely on busy-waiting which can unnecessarily detain a processor from serving other threads. Meta-locking has been implemented in the Solaris Production Java virtual machine (Java SDK 1.2) from Sun Microsystems and benchmarks show impressive performance on several large programs.

That the meta-locking scheme is correct is far from given. For example, to save space, it uses a delicate “virtual queue” technique (our terminology) to implement a first-come-first-served policy for handling meta-lock requests. In a virtual queue, a thread knows its predecessor on the queue but not its successor (notice the asymmetry). Therefore, a thread releasing a meta-lock must dynamically determine its successor in order to “hand-off” the meta-lock to this thread. For this purpose, a predecessor thread releasing a meta-lock enters a “race” with its successor thread to lock a mutex variable in the predecessor’s “execution environment” (EE) data structure. As described below, the winner of the race gets to update a certain variable in the predecessor's EE. Thus, each thread can determine if it won the race by noting whether the competitor has made the corresponding change.

If the successor wins, it writes its thread id in the predecessor’s EE. This is the first time that the predecessor has learned of its successor’s identity, and it will now hand-off the meta-lock to the successor and signal the successor that the hand-off is complete. If the predecessor wins, it still does not know the identity of its successor but it knows that its successor knows its identity! Therefore it updates its EE by making the meta-lock “up for grabs” to the successor. The successor will eventually complete the hand-off and signal the predecessor of this fact.

Besides the virtual queue idea for handling meta-lock requests, the algorithm’s correct operation critically depends upon two atomic swap operations, one in the routine for acquiring a meta-lock and the other in the routine for releasing a meta-lock. Both of these swap operations are used to determine if there is contention for the meta-lock in question, and both must be executed atomically in order to ensure mutual exclusion.

[ADG+99] also proposes a sophisticated optimization of the basic meta-locking algorithm for
the case of uncontended objects. The idea behind the optimization, referred to as the *extra-fast path* for locking and unlocking, is to fuse the meta-lock and monitor-lock operations when a thread attempting to lock an object finds the object in an unlocked state. The extra-fast path for locking uses one atomic (compare-and-swap) instruction rather than the two needed for meta-locking and meta-unlocking, and the total number of machine instructions that need to be executed is smaller as well.

The authors of [ADG+99] have recognized the importance of proving their meta-locking algorithm correct, and have consequently provided informal proof sketches for two important correctness properties: mutual exclusion and freedom from lockout. In this paper, we show how model checking, in particular, as implemented in the XMC logic-programming-based model checker [RRR+97], can be brought to bear on the problem. Model checking [CGP99] is the process of determining whether a system specification satisfies (is a model of) a correctness property given as a temporal logic formula. Our main results are the following:

- We have produced an abstract (suitable for model checking) specification of the meta-locking algorithm in XL, the input language of XMC. The specification is completely parameterized, both by the number of threads in the system and the number of objects in the system.

- For a variety of values of $M$ (the number of threads) and $N$ (the number of objects), we have used model checking to establish that the meta-locking algorithm does indeed provide mutual exclusion and freedom from deadlock and lockout. Some of these cases were particularly challenging, with state-space sizes as large as 2.5 million states. Collectively, our results provide strong testimony to the correctness of the meta-locking algorithm.

- We have also extended our XL specification of the meta-locking algorithm to capture the extra-fast path for locking and unlocking uncontended objects. The required changes are significant since the monitor-locking protocol must now be specified, in addition to the protocol for meta-locking. In contrast, to model the basic meta-locking algorithm, it sufficed to abstract away all details pertaining to monitor locking. The situation is further complicated by several interesting features of the monitor-locking protocol, including potential "out-of-order" access by threads to an object’s monitor lock. Our model-checking results on the extended specification indicate that mutual exclusion and freedom from deadlock and lockout at the meta-locking level continue to hold for this optimized version of the algorithm. However, the monitor-locking protocol of [ADG+99] is inherently unfair and our model-checking results bear this fact out.

In terms of related work, model checking has been used recently to verify the correctness of a variety of systems, including a number of industrial applications; see, e.g., [GR97]. Perhaps most closely related is the work reported in [PPTH91], where the Spin verification tool was used to detect a subtle race condition in the process sleep and wakeup primitives of the Plan 9 operating system;
and the work of [BFGP02], which uses model checking to statically verify the type safety of Java class files, a job usually carried out at runtime by the bytecode verifier of the JVM.

The rest of this paper develops along the following lines. Section 2 describes the XMC verification tool we used in the analysis of the meta-locking algorithm. The algorithm itself is presented in Section 3 while Section 4 discusses our XL encoding of the algorithm. Section 5 considers the extra-fast synchronization path and our modeling of this optimization. Section 6 contains our model-checking results, while Section 7 offers some concluding remarks. A preliminary version of this paper appeared as [BSW00].

2 The XMC Verification Tool

XMC [RRR⁺97], developed here at SUNY Stony Brook, is a model checker for a value-passing process calculus [Mil89] and the modal mu-calculus [Koz83]. A novelty of the system is that it is written in a highly declarative fashion, in just under 200 lines of XSB tabbled Prolog code. XSB [XSB01] is a logic-programming system created at SUNY Stony Brook using an extension of Prolog-style SLD resolution with tabled resolution. This enables XSB to terminate on programs having finite models, to compute the model of normal logic programs, and to avoid redundant subcomputations.

XL, the specification language for XMC, is a highly expressive extension of value-passing CCS [Mil89]. Prolog terms and predicates are used respectively to represent values and computations. Thus specifications can make use of recursive data structures and computations.

The syntax of the XL specification language is given by the grammar of Figure 1 where \( \text{Comp} \) is a term representing a computation (e.g. \( X \text{ is } Y+1 \)). A terminating null process is denoted by \( \text{true} \). Process expression \( \text{if } C \text{ then } S_1 \text{ else } S_2 \) behaves like \( S_1 \) if computation \( C \) succeeds, and like
$S_2$ if $C$ fails. The computation $C$ in an if expression is assumed to leave the bindings of variables unchanged. The process $c?t$ inputs a value that matches term $t$ over channel $c$; $c!t$ outputs the value represented by term $t$ over channel $c$. Processes communicate by synchronizing input and output actions over identical channels. The expression action($t$) represents an observable action $t$. Logical properties of processes are defined over the observable actions. Finally, the expression Proc is a parameterized process name represented as a term (e.g. medium(Get, Put)). The parameters of a process term include non-channel variables and constants, where the variables are treated as local, and channels on which the process can communicate. Process invocations may be recursive; in fact, since the language provides no iterative constructs, recursion is the only way to specify loops in processes. Unlike CCS, XL does not have explicit restriction and relabeling operators. Relabeling is achieved by invocation of process terms with appropriate arguments for channel names (e.g. see definition of abp in Figure 2). The restricted ports of a process are the ports included among the parameters of the process definition.

The complete specification of the Alternate Bit Protocol [Tan96] in Figure 2 illustrates the features of XL. Text preceded by the % character are comments.

3 The Meta-Locking Algorithm

In this section we describe the meta-locking algorithm. The purpose of this algorithm is to allow a thread to gain access to an object’s monitor queue in order to insert a lock record into the queue. Java uses monitor-style synchronization on objects and a lock record represents a request by a thread to enter an object’s monitor. Manipulation of object monitor queues by threads must be done in a mutually exclusive fashion to ensure consistency of the queue structure, and the meta-lock algorithm is designed to provide such mutually exclusive access.

Figure 3 depicts the scenario where two threads, thread1 and thread2, have placed their respective lock records in the monitor queue of a certain object. The combination of the monitor queue and the contents of the object’s multi-use word (MUW), which contains a pointer to the head of the queue and is described in detail below, are referred to as the object’s synchronization data [ADG+99]. Threads thread3 and thread4, on the other hand are trying to gain access to the synchronization data so that they can also insert their lock records in the queue. Thus, in this scenario, thread1, whose lock record is at the head of the queue, has monitor-locked the object and can therefore access the object data. Thread2 is waiting for thread1 to release the monitor-lock, and threads thread3 and thread4 are trying to meta-lock the object synchronization data.

In the meta-locking algorithm, threads observe a certain protocol when manipulating an object’s synchronization data. The pattern of synchronization operations in the meta-locking algorithm is as follows:

1. Get the object’s meta-lock to ensure exclusive access to the object’s synchronization data.
medium(Get, Put) ::= Get ? Data; { Put ! Data # action(drop) }; medium(Get, Put).

sendnew(AckIn, DataOut, Seq) ::= action(sendnew); sender(AckIn, DataOut, Seq).

sender(AckIn, DataOut, Seq) ::= 
  % Seq is the sequence number of the next frame to be sent
  DataOut ! Seq;
  { 
    AckIn ? AckSeq;
    if AckSeq == Seq then { % successful ack, next message
      NSeq is 1-Seq;
      sendnew(AckIn, DataOut, NSeq)
    } 
    % unexpected ack, resend message
    else sender(AckIn, DataOut, Seq) 
    #
    % upon timeout, resend message
    sender(AckIn, DataOut, Seq)
  }.

receiver(DataIn, AckOut, Seq) ::= 
  % Seq is the expected next sequence number
  DataIn ? RecSeq;
  if RecSeq == Seq then { % correct sequence received
    NSeq is 1-Seq;
    action(recv);
    AckOut ! RecSeq;
    receiver(DataIn, AckOut, NSeq)
  }
  else {
    % unexpected seq, resend ack
    AckOut ! RecSeq;
    receiver(DataIn, AckOut, Seq)
  }.

abp ::= 
  sendnew(R2S_out, S2R_in, 0)
  | medium(S2R_in, S2R_out) % sender -> receiver
  | medium(R2S_in, R2S_out) % receiver -> sender
  | receiver(S2R_out, R2S_in, 0).

Figure 2: Specification of the Alternating Bit Protocol in XL.
2. **Manipulate** the synchronization data.

3. **Release** the meta-lock (if no other thread is waiting to acquire the meta-lock) or **hand off** the meta-lock (to a waiting thread, the next one “in line”).

A thread that encounters no contention while attempting to acquire a meta-lock is said to execute the **fast path** for that operation; otherwise it takes the **slow path**. The situation is exactly similar for the operation of releasing a meta-lock. The slow paths constitute the portion of the algorithm that implements the meta-lock hand-off. There is also an optimized scheme referred to as the **extra-fast path** that fuses the meta-lock and monitor-lock operations in the case of uncontended objects. We consider this optimization in Section 5.

The main data structures used by the meta-locking algorithm are the **execution environment** (EE), one per thread, and the **multi-use word**, one per object. Consider first the 32-bit multi-use word (MUW), which is actually the second word of a two-word object header (the first word points to the object’s class). The two least-significant bits are referred to as the **lock bits** of the MUW and allow for the encoding of four possible **lock states** of the object, as shown in Figure 4 and described here:
typedef struct execenv {
    Thread thread; /* ExecEnv is a subtype of Thread. */
    mutex_t metaLockMutex; /* Used by slow-path meta-lock/unlock. */
    condvar_t metaLockCondvar; /* To wait for meta-lock hand-off. */
    bool_t gotMetaLockSlow; /* Wait for predecessor to give bits. */
    bool_t bitsForGrab; /* Wait for successor to grab bits. */
    BitField metaLockBits; /* Space to get/give releaseBits. */
    ExecEnv *succEE; /* Next thread to get the meta-lock. */
    mutex_t monitorLockMutex; /* Used by slow-path lock/unlock. */
    condvar_t MonitorLockCondvar; /* To wait for monitor acquisition. */
    ... other fields ...
} ExecEnv;

Figure 5: Per-thread Execution Environment.

**NEUTRAL** Objects are created in the neutral state, remain in this state as long as no thread synchronizes on them, and return to NEUTRAL once synchronization ceases. In this state, the first 30 bits of the MUW are used by other parts of the Java Virtual Machine such as the garbage collector, and both lock bits are set to zero.

**LOCKED** The first 30 bits of the MUW contain a pointer to the lock record at the head of the object’s monitor queue, indicating which thread owns the monitor-lock and also stores the displaced hash and age information that is maintained in the MUW while the lock state is neutral. The lock bits are set to 01.

**WAITERS** This state is entered when a thread releases the monitor lock while other threads are waiting to acquire the lock or be notified. The object is no longer monitor-locked but the state must be distinguished from NEUTRAL since the first 30 bits of the MUW still point to synchronization data. The lock bits are set to 10.

**BUSY** The object is meta-locked. The first 30 bits of the MUW point to the EE of the thread that has the meta-lock. The lock bits are set to 11.

For the purpose of modeling the basic meta-locking algorithm it is sufficient to consider only two lock states: BUSY and NON-BUSY. In this case, lock states NEUTRAL, LOCKED and WAITERS are abstracted into the single state NON-BUSY, as indicated in Figure 4. In Section 5, when we extend our model of the algorithm to include extra-fast synchronization, we will need to consider all four lock states of the object.

A thread releasing an object’s meta-lock must update the object’s MUW to reflect the new state of the object’s monitor queue. [ADG+99] refers to the new content of the MUW as the thread’s release bits. The lock state of a thread’s release bits will always be NON-BUSY.
BitField getMetaLock(ExecEnv *ee, Object *obj) {
    BitField busyBits = ee | BUSY;
    BitField lockBits =
        SWAP(busyBits, multiUseWord(obj));
    return getLockState(lockBits) != BUSY ?
        lockBits: getMetaLockSlow(ee, lockBits);
}
void releaseMetaLock(ExecEnv *ee, Object *obj, BitField releaseBits) {
    BitField busyBits = ee | BUSY;
    BitField lockBits = CAS(releaseBits,
        busyBits, multiUseWord(obj));
    if (lockBits != busyBits)
        releaseMetaLockSlow(ee, releaseBits);
}

Figure 6: Fast paths for meta-lock operations.

The definition of the EE data structure is given in Figure 5 as type definition ExecEnv. The
figure is taken essentially verbatim from [ADG+99]. ExecEnv is a subtype of Thread; since EEs
and threads correspond one to one, EE addresses are well-suited as unique thread identifiers. Thus,
Thread can be viewed as a thread id. metaLockMutex is the mutex variable a predecessor thread
releasing a meta-lock and a successor thread acquiring the meta-lock race to acquire (recall the
discussion of the algorithm given in Section 1). metaLockCondvar is a POSIX-style condition
variable that the successor uses to wait for the predecessor to finish its half of the hand-off, and
vice versa. metaLockBits is space for storing the predecessor’s release bits. bitsForGrab allows
the predecessor to wait for the successor to “grab” (copy) its release bits; it is set to true by the
predecessor when it wins the race. getMetaLockSlow allows the successor to wait for the predecessor
to copy over its release bits, when the successor wins the race. Finally, succEE will be written
by the successor in the predecessor’s EE when the successor wins the race. The remaining two
fields, monitorLockMutex and monitorLockCondvar, are utilized in the monitor locking/unlocking
protocol of [ADG+99]; we consider monitor locking and unlocking in Section 5.

The Java-style pseudo-code for acquiring and releasing meta-locks, reproduced from [ADG+99],
is presented in Figure 6. A thread attempts to acquire the meta-lock by executing an atomic swap
operation to replace the object’s multi-use word with a word consisting of a reference to the thread’s
EE and low-order bits representing the BUSY lock state. If the value of the multi-use word read by
the swap operation indicates that the object’s meta-lock is not busy, then the thread has acquired
the meta-lock and may proceed. This is the fast path for meta-lock acquisition. If, however, the
object’s meta-lock is found to be BUSY, then some other thread holds the meta-lock and the current
thread invokes getMetaLockSlow(). In this case, the threads contending for the meta-lock are
totally ordered by the order in which they executed the swap instruction. Every thread in the
order knows its predecessor from the EE in the multi-use word it read. Moreover, the first thread
in this order knows that it has no predecessor, since it acquired the meta-lock.

To release a meta-lock, a thread executes an atomic compare-and-swap (CAS) operation to
atomically compare the current value of the object’s multi-use word with what it had written there
when it attempted to acquire the lock. If it is still the same, then no contention has occurred
BitField getMetaLockSlow(ExecEnv *ee, BitField predbits) {
    BitField bits;
    ExecEnv *predEE = busyEE(predBits);
    mutexLock(&predEE->metaLockMutex);
    if (!predEE->bitsForGrab) {
        /* Won the race: */
        predEE->succEE = ee;
        do {
            condvarWait(&predEE->metaLockCondvar,
                        &predEE->metaLockMutex);
        } while (!ee->gotMetaLockSlow);
        ee->gotMetaLockSlow = FALSE;
        bits = ee->metaLockBits;
    } else {
        /* Lost the race: */
        bits = predEE->metaLockBits;
        predEE->bitsForGrab = FALSE;
        condvarSignal(&predEE->metaLockCondvar);
    }
    mutexUnlock(&predEE->metaLockMutex);
    return bits;
}

void releaseMetaLockSlow(ExecEnv *ee, BitField releaseBits) {
    mutexLock(&ee->metaLockMutex);
    if (ee->succEE) {
        /* Lost the race: */
        ee->succEE->metaLockBits = releaseBits;
        ee->succEE->gotMetaLockSlow = TRUE;
        ee->succEE-> = NULL;
        condvarSignal(&ee->metaLockCondvar);
    } else {
        /* Won the race: */
        ee->metaLockBits = releaseBits;
        ee->bitsForGrab = TRUE;
        do {
            condvarWait(&ee->metaLockCondvar,
                        &ee->metaLockMutex);
        } while (ee->bitsForGrab);
    }
    mutexUnlock(&ee->metaLockMutex);
}

Figure 7: Slow paths for meta-lock operations.

and the release bits are written. Otherwise, contention exists and the releasing thread will “hand
off” the meta-lock to the next thread in the order induced by the swap operations by calling
releaseMetaLockSlow().

The hand-off protocol is defined in the pseudo-code for slow-path meta-lock operations given
in Figure 7; the code in this figure is also reproduced from [ADG+99]. The main objective
of the slow-path operations is for the releasing predecessor thread to hand off the meta-lock to
the acquiring successor thread. To accomplish this, they both try to lock the metaLockMutex
variable in the predecessor’s EE. The race has two possible outcomes: the successor wins (the case to be
expected more frequently in practice) or the predecessor wins. The successor will know it won the
race if it finds that the predecessor’s bitsForGrab is false upon acquiring the mutex variable. In
this case it assigns its EE to the predecessor’s succEE and waits for the predecessor to complete
its half of the hand-off (releasing the mutex in the process). The predecessor, in turn, realizes it
has lost the race by finding succEE to be non-null upon acquiring the mutex variable. In this case,
it places its release bits in the successor’s metaLockBits, sets the successor’s gotMetaLockSlow to
true to indicate that those bits are valid, resets its succEE to its default value of null, signals the
successor that it has completed its transaction, and releases themutex. The successor, now back in possession of the mutex, resets its gotMetaLockSlow to its default value of false, reads the release bits from its EE, releases the mutex, and continues, having acquired the meta-lock.

The predecessor wins the race if it finds succEE to be null upon acquiring the mutex variable. In this case, it places its release bits in metaLockBits, sets bitsForGrab to true to indicate that those bits are valid, and waits for the successor to complete its side of the hand-off (releasing the mutex in the process). The successor, in turn, realizes it has lost the race by finding the predecessor's bitsForGrab true. In this case, it reads the release bits out of the predecessor's EE, resets bitsForGrab in the predecessor's EE to its default value of false, signals the predecessor that it has completed its transaction, and releases the mutex. The predecessor, now back in possession of the mutex, releases the mutex, and continues.

![Figure 8: Architecture of the specification for two threads and one object.](image-url)

Figure 8: Architecture of the specification for two threads and one object.
4 Modeling the Meta-Locking Algorithm in XMC

In this section we describe how we specified the meta-locking algorithm in XL, the input language of the XMC model checker. The complete XL source listing of the specification can be found at http://www.cs.sunysb.edu/~lmc/metaj/. The basic ingredients of any XL specification are parameterized processes that execute concurrently and that exchange values over channels. Channels are unidirectional and any number of processes can output values to or receive values from a given channel. However, a communication over a channel involves exactly two processes and is synchronous, requiring a handshake between communicants.

Assuming a system with \( M \) threads and \( N \) objects, our XL specification of the meta-locking algorithm is given by the process \( \text{metaj}(M,N) \), consisting of the parallel composition of \( M+N+1 \) processes: one per thread, one per object, and one for a special hand-off process. These processes are linked together by a variety of communication channels and the messages exchanged over these channels contain the ids of the communicants. The purpose of these channels will be made clear in the ensuing discussion. An architectural diagram of the specification is given in Figure 8 for the case of two threads and one object.

A thread process is of the form \( \text{thread(Thread\_id, N)} \), where \( \text{Thread\_id} \) is an integer between \( 1 \) and \( M \) inclusively, uniquely identifying the thread in question.\(^1\) Although XL processes are parameterized, they do not have “state” per se. We therefore use thread ids instead of EEIs to uniquely identify threads and encode EE fields in the messages exchanged between threads and between threads and the hand-off process. The basic behavior of a thread process is to loop forever, each time nondeterministically selecting one of the \( N \) objects in the system for attempted meta-locking by calling \( \text{getmetalock(Thread\_id, Object\_id)} \) followed by \( \text{releasemetalock(Thread\_id, Object\_id)} \).

An object process is of the form \( \text{object(Multiuseword, Object\_id)} \), where \( \text{Object\_id} \) is an integer between \( 1 \) and \( N \) inclusively, uniquely identifying the object in question, and \( \text{Multiuseword} \) is of one of two forms: \( \text{Thread\_id} \) concatenated with \( \text{busy} \) or \( \text{Thread\_id} \) concatenated with \( \text{not\_busy} \). An object process supports the two types of atomic swap operations utilized in the fast-path meta-lock operations (Figure 6). The encoding of these operations is perhaps one of the most interesting aspects of the specification.

Consider first the atomic SWAP operation. The \( \text{getmetalock} \) process executes the output command \( \text{Swap} \! (\text{Object\_id}, (\text{Id,Lockstate}), (\text{Thread\_id,busy})) \) to swap its \( (\text{Thread\_id, busy}) \) with the multi-use word of object \( \text{Object\_id} \), the value of the latter ending up in \( (\text{Id, Lockstate}) \). Conversely, \( \text{object(Multiuseword, Object\_id)} \) executes the input command \( \text{Swap} \! (\text{Object\_id}, (\text{Multiuseword,Newword})) \) to complete the swap, then continuing as

\(^1\)Processes definitions such as the one for \( \text{thread} \) are also parameterized by channels over which the defined processes can communicate (see Section 2). To increase the readability of process definitions, we shall hereafter omit the channel parameters. As stated above, the channels used by a process in our XL specification of the meta-locking algorithm can easily be determined by consulting Figure 8.
 releasemetalock(Thread_id, Object_id) ::=          object(Multiuseword, Object_id) ::=  
 Cas_req ! (Object_id, Thread_id);              Cas_req ? (Object_id, Thread_id);
 Cas_object_to_thread ? (Thread_id, Object_id, Newword);       if Multiuseword == (Thread_id,busy) then {
 if Newword == null then        Cas_object_to_thread ! (Thread_id, 
 releasemetalockslow(Thread_id, Object_id)       Object_id, Multiuseword); 
  object((Thread_id,not_busy), Object_id) } 
 else 
  true. 
  else {
  Cas_object_to_thread ! (Thread_id, 
  Object_id, null); 
  object(Multiuseword, Object_id)}. 

Figure 9: XL specification of the atomic compare-and-swap operation.

hand_off ::= 
Mutex ? (Thread_id, Object_id); Mutex_other ? (Thread_id, Object_id); 
( 
  Bits_for_grab ! (Thread_id, Object_id, false) 
  # 
  Succ ! (Thread_id, Object_id, 0) 
); 
hand_off. 

Figure 10: XL specification of the hand-off process.

object(Newword, Object_id). Since the joint execution of these complementary output and input commands happens in one execution step of the system, the swap is guaranteed to be atomic. It is the power of Prolog-style logical variables and unification that makes such a succinct encoding of atomic swap possible.

The atomic CAS operation is encoded in a similar, albeit somewhat more complex, fashion. The relevant portions of the XL code are given in Figure 9. The thread and object processes initiate the operation by synchronizing on the message (Object_id, Thread_id) Thread_id transmits over channel Cas_req. The object then compares its multi-use word to the value (Thread_id,busy). If the comparison succeeds (indicating no lock contention is present) this value will be bound, via an input command, to Newword in the releasemetalock process, and the object process will continue as object((Thread_id,not_busy), Object_id). Otherwise, Newword will be bound to null, the object’s multi-use word will remain unchanged, and the thread process will call releasemetalockslow.

That all of this will happen “atomically” is guaranteed by the fact that the thread and object processes cannot be interrupted once the compare-and-swap operation is initiated; i.e. for the duration of the CAS operation, these two processes are only willing to communicate with each other.

The purpose of the hand-off process—see Figure 10—is to simulate the hand-off of the release bits from the predecessor thread to the successor thread. In particular, the hand-off process first synchronizes with the competing thread processes, one of which, the successor, will be executing
getmetalockslow(Thread_id, Pred_id, Object_id) ::= 
Mutex_other ! (Pred_id, Object_id) ;
Bits_for_grab ? (Pred_id, Object_id, Grab_bits) ;
if Grab_bits == false then {
    Succ ! (Pred_id, Object_id, Thread_id) ;
    Metalock_bits_other ? (Thread_id, Object_id) ;
    else {
        Metalock_bits ? (Pred_id, Object_id) ;
        thread_to_thread ? (Pred_id, Object_id) ;
        hand_off(Object_id).
    }
}
releasemetalockslow(Thread_id) ::= 
Mutex ! (Thread_id, Object_id) ;
Succ ? (Thread_id, Object_id, Succ_id) ;
if Succ_id \= 0 then 
    Metalock_bits_other ! (Succ_id, Object_id) ;
else 
    Bits_for_grab ! (Thread_id, Object_id, true) ;
    Metalock_bits ! (Thread_id, Object_id) ;
    thread_to_thread ! (Thread_id, Object_id) ;
    hand_off(Object_id).

Figure 11: XL specification of the slow-path operations.

getmetalockslow and synchronizing with the hand-off process via channel Mutex_other, and the other of which, the predecessor, will be executing releasemetalockslow and synchronizing with hand-off via Mutex. The hand-off process then nondeterministically decides who won by outputting false on the appropriate Bits_for_grab channel, if the successor won, or by outputting zero on the appropriate succ channel, if the predecessor won.

The XL code for getmetalockslow and releasemetalockslow in Figure 11 completes the picture as to how we modeled the hand-off protocol. Process getmetalockslow first tries to get the mutex of the predecessor and then examines Grab_bits, which it inputs via the message Bits_for_grab ? (Pred_id, Object_id, Grab_bits) to determine if it won the race. The “trick” here is that the value of Grab_bits may be supplied by either the hand-off process or the predecessor process. In the former case, the successor won the race and the value of Grab_bits is false. In the latter case, the predecessor won the race and the value of Grab_bits is true. If the successor wins, it sends its Thread_id to the predecessor and then waits for the predecessor to send a signal via channel Metalock_bits_other. The hand-off is complete after this signal is received. If the successor loses, it simply waits for the predecessor to send a signal via channel Metalock_bits.

In the case of the releasemetalockslow, the predecessor process receives Succ_id either from process hand_off (Succ_id is zero and the predecessor has won) or from the successor process (Succ_id is non-zero and the predecessor has lost). In the former case, it sends true for Grab_bits to the successor and completes the hand-off by emitting a signal from Metalock_bits. In the latter case, the predecessor sends a signal to the successor’s input channel Metalock_bits_other.

Figure 12 illustrates the high-level behavior of a thread process trying to meta-lock/unlock an object following the fast or slow paths of synchronization described above. The state-transition labels reveal the state of the object as seen by the thread after completing the atomic operations.
Figure 12: State-transition diagram illustrating high-level thread behavior.

(SWAP and CAS).

5 Extra-Fast Locking and Unlocking of Uncontended Objects

In [ADG+99], Agesen et al. propose an extra-fast synchronization path for uncontended objects. In this section, we describe how we extended our XL specification of the basic meta-locking algorithm to formally capture this optimization. Extra-fast synchronization uses the fact that, most of the time, threads do not contend to access object data; this in turn implies that objects spend the majority of time in the NEUTRAL state. As described in Section 3, two levels of locking are involved when a thread synchronizes on an object: meta-locking to access the object’s synchronization data and monitor locking to access the object data. Specifically, whenever a thread tries to access object data, it first meta-locks the object to insert its lock record in the monitor queue and then monitor-locks the object (when its lock record reaches the head of the queue) to access the object data. The extra-fast path optimization fuses the meta-lock and monitor-lock operations into one step thus reducing the number of lock and unlocks needed to access uncontended object data.

5.1 Description of the Extra-Fast Locking Protocol

To describe the extra-fast synchronization path, it is first necessary to consider the normal monitor-locking protocol proposed by [ADG+99]. We do this now, starting with monitor-lock acquisition.

**Acquiring a monitor-lock.** A thread attempting access to object data first meta-locks the object by either the fast or slow path as described in Section 3. The subsequent behavior of the thread depends on the state in which it found the object during meta-locking. There are three
possible cases:

1. **NEUTRAL.** The thread releases the meta-lock with the release bits containing the thread's id and a lock-state of **LOCKED**.

2. **WAITERS.** The object is unlocked and one or more threads are waiting to monitor-lock the object. The thread rearranges the synchronization data by inserting its lock record at the head of the queue and releases the meta-lock in a lock-state of **LOCKED**. This leads to **out-of-order access** of objects by threads and is actually the result of the thread winning a race with the waiting threads for the monitor lock on the object. This reflects the decision by the authors of [ADG+99] to give equal preference to waiting threads and newly arriving threads.

3. **LOCKED.** The thread appends its lock record to the monitor queue and suspends on the condition variable **monitorLockCondvar**, waiting to be awakened by a monitor-lock releasing thread. [ADG+99] refer to this path as the **monitorEnterSlow** operation. Once awakened, the thread again meta-locks the object and checks the object's state. If it is different from **LOCKED**, it updates the synchronization data to indicate that it holds the monitor lock and releases the meta-lock; otherwise the thread releases the meta-lock and waits again.

**Releasing a monitor-lock.** When a thread monitor-unlocks an object, it first meta-locks the object. In this case, the state in which the thread finds the object can only be **LOCKED**. What it does next depends on the size of the synchronization data:

1. If there are no other threads waiting to enter the object monitor, the thread releases the monitor-lock and changes the state of the object to **NEUTRAL**.

2. If the length of the monitor queue is greater than one, then one or more threads are waiting to access the object. In this case, the thread removes its lock record from the queue, awakens the threads waiting on **monitorLockCondvar**, changes the object state to **WAITERS**, and releases the meta-lock. [ADG+99] refer to this path as the **monitorExitSlow** operation.

**Extra-fast path.** Note that in the above cases, a thread is required to perform at least two atomic operations (**SWAP** in acquiring the meta-lock and **CAS** in releasing the meta-lock) to monitor-lock an object. The extra-fast optimization is aimed at reducing the number of atomic operations and also the total number of machine instructions that need to be executed. In this protocol, a thread, attempting to access object data, first reads the object's multi-use word. If the object's lock state is **NEUTRAL**, the thread copies the hash and age bits into a fresh lock record and builds a new multi-use word containing the address of the lock record and the **LOCKED** state. A **CAS** operation is then performed to atomically change the multi-use word to the new value if it has not changed since

---

2The condition variable **monitorLockCondvar** is a field in the thread's execution environment (see Figure 5), and a pointer to the thread's execution environment is contained in the lock record.
it was read. If the CAS succeeds, then the object enters the LOCKED state; otherwise, the normal monitor-locking protocol is followed.

In the extra-fast path for unlocking, a thread first performs an atomic CAS operation to check if its lock record is still the only lock record in the monitor queue. If the CAS operation succeeds, implying that no other thread is trying to access the object, the thread will have removed its lock record from the queue and updated the object multi-use word to the NEUTRAL state in one atomic operation. Otherwise, the normal monitor-unlocking protocol is followed. For efficiency reasons, [ADG+99] further stipulates that lock records are allocated with eight-byte alignment, so that three bits are zero in the address of a lock record. In the LOCKED state, this extra bit is used to summarize the state of the queue field of the first lock record; if the bit is zero, the queue field is NULL. Our XL specification of extra-fast unlocking given below takes a much more abstract approach to keeping track of the state of the monitor queue.

To better understand the meta-locking algorithm in general, and the extra-fast optimization in particular, an example execution scenario is presented in Figure 13. The example focuses on a single object and the attempts by various threads to monitor-lock and unlock the object. The execution scenario commences with Thread1—as a result of finding the object in the NEUTRAL state and uncontended—having successfully monitor-locked the object via the extra-fast synchronization path. Thread2 then attempts to lock the object but finds it in the LOCKED state. It must therefore meta-lock the object, setting the state of the object to BUSY, so it can update the object’s synchronization data. Thread2 then releases its meta-lock, restoring the state of the object to LOCKED (by Thread1), and then waits on condition variable monitorLockCondvar. Next, Thread3 enters the picture, and like Thread2, updates the synchronization data and waits on monitorLockCondvar. At this point, Thread1 releases its monitor lock, wakes up the first thread on the monitor queue that is waiting to acquire the lock (Thread2), and updates the object’s state to WAITERS. Now, before Thread2 can meta-lock the object, newly arriving Thread4 obtains the meta-lock and subsequently monitor-locks the object. Finally, Thread4 releases its monitor lock and awakens Thread3.

5.2 Modeling the Extra-Fast Locking Protocol

In this section, we describe the extensions we made to our XL specification of the basic meta-locking algorithm in order to model the extra-fast path of synchronization. The changes to the specification are significant since, to capture this optimization, we must model the monitor-locking protocol of [ADG+99] in addition to the protocol for meta-locking. In contrast, to model the basic meta-locking algorithm, it sufficed to completely abstract away all details pertaining to monitor locking. The situation is further complicated by several interesting features of the monitor-locking

---

3 Section 6.3 of [ADG+99] discusses an alternative to the single monitor-queue approach wherein separate queues for waiting and awakened threads are maintained. Our XL specification given in Section 5.2, which models a “pool” of waiting threads, one of which is chosen nondeterministically to be awakened by a lock-releasing thread, closely follows this alternative strategy. Modeling a pool of waiting threads, as opposed to an ordered queue, leads to significantly smaller state spaces.
Thread1 has monitor-locked the object via extra-fast path

Thread2 updates the monitor queue, releases the meta-lock and waits for wake-up signal

Thread1 releases monitor-lock, wakes up one of the waiting threads (Thread2) and updates object state to Waiters

Thread4 monitor unlocks the object before Thread2 is scheduled and wakes up Thread3; object state becomes waiters

Figure 13: Example of meta-locking algorithm, with extra-fast optimization, in action.
protocol, including potential “out-of-order” access by threads to an object’s monitor lock. That is, despite the fact that the basic data structure utilized by the protocol is a queue of lock records, access to the monitor need not follow a queue discipline, due to the decision to give equal preference to awaiting threads and newly arrived threads.

To specify the extra-fast path, all four lock-states of an object’s multi-use word, as described in Section 3, must be considered: the abstraction that merged lock states NEUTRAL, LOCKED and WAITERS into the single abstract lock state NON-BUSY is no longer appropriate. In particular, when a thread attempts to monitor lock an object, there is now a need to distinguish lock state NEUTRAL (from which extra-fast locking may proceed) from any other lock state. Further, should extra-fast locking fails, it is now necessary to distinguish lock state WAITERS from lock state LOCKED (from which the slow path for monitor-locking must be followed). When a thread releases the monitor lock of an object, the lock state must be set to NEUTRAL if there are no other lock records in the queue, and to WAITERS otherwise. Note that, in this extended model, the specification for \textit{getmetalock} needs to record the multi-use word it obtained from either (a) the object (\textit{fast path})
monitor_enter(Thread_id, Object_id) ::= 
/* thread tries extra fast monitor locking */
Cas ! (Object_id,Thread_id,extra_fast_lock_req);
Cask_object_to_thread ? (Thread_id,Object_id,
m(Thread_id,locked),Newword);
/* Newword = null => extrafast path failed */
if Newword == null then {
getmetalock(Thread_id,Object_id,Release_bits);
To_Count_gate ! (Object_id,adding,0);
Release_bits = m(Owner,LockState);
if LockState == neutral then {
/* Establish locking by this thread */
releasemetalock(Thread_id, Object_id,
m(Thread_id,locked))
else {
if LockState == locked then {
/* monitorenter_slow is invoked */
monitor_enter_slow(Thread_id, Object_id,
Release_bits))
else {
if LockState == waiters then {
/* monitor lock the object */
releasemetalock(Thread_id, Object_id,

m(Thread_id,locked))
})
/* extrafastpath succeeded */
else To_Count_gate ! (Object_id,adding,0).

monitor_exit(Thread_id, Object_id) ::= 
/* thread tries extra fast unlocking */
Cas ! (Object_id,Thread_id,
extra_fast_unlock_req);
Cask_object_to_thread ? (Thread_id, Object_id,
m(0,neutral),Newword);
/* Newword = null => extrafast path failed */
if Newword == null then {
getmetalock(Thread_id,Object_id,Release_bits);
To_Count_gate ! (Object_id,removing,0);
Release_bits = m(Owner,LockState);
/* no other thread is metalocking at the moment */
if LockState == locked /
    Owner == Thread_id then {
    From_Wait ? (Object_id, status, W_Length);
    From_Count_gate ? (Object_id, status,
    Q_Length);
    /* other threads are waiting for wake-up signal*/
    if W_Length > 0 then
        monitor_exit_slow(Thread_id, Object_id)
    else {
        if Q_Length > 0 then
            New_Release_bits = m(0, waiters)
        else New_Release_bits = m(0, neutral))
    releasemetalock(Thread_id, Object_id,

    New_Release_bits)
})
/* extrafast unlocking succeeds */
else To_Count_gate ! (Object_id,removing,0).

monitor_enter_slow(Thread_id, Object_id,
Release_bits) ::= 
releasemetalock(Thread_id, Object_id,Release_bits);
/* wait for the wakeup signal */
To_Wait ! (Object_id, adding, 0);
From_Wait ? (Object_id, signaltowait, 0);
/* acquire metalock to check object state */
getmetalock(Thread_id, Object_id,New_Release_bits);
New_Release_bits = m(Owner,LockState);
/* some other thread has locked the object */
if LockState == locked then
    monitor_enter_slow(Thread_id, Object_id,
    New_Release_bits)
else /* current thread gets the monitor lock */
    releasemetalock(Thread_id, Object_id,
    m(Thread_id, locked)).

monitor_exit_slow(Thread_id, Object_id) ::= 
/* signal the waiting threads */
Wake_up ! (Object_id, signaltowait);
releasemetalock(Thread_id, Object_id,m(0,waiters)).

(a) (b)

Figure 15: XL specification of monitor_enter and monitor_exit processes.
or (b) the hand-off process (slow path). In the meta-locking model (Section 4), this information is not required as state of the object on release of meta-lock is always the abstract state NON-BUSY.

Additionally, for each object in the system, we introduce two processes: \texttt{count(Object\_id, Len)} to keep track of the number of lock records in the queue, and \texttt{wait\_pool(Object\_id, Len)} to record the number of threads waiting to be awakened by a lock-releasing thread. Note that these two numbers need not be the same: the thread in the monitor queue that has monitor-locked the object is not waiting; nor are those threads that have been awakened by a lock-releasing thread but have not yet have attempted to reacquire the meta-lock (only the attempt to reacquire the meta-lock would force it again to wait on \texttt{monitorLockCondvar}). The example execution scenario depicted in Figure 13 illustrates this state of affairs.

An architectural diagram revealing the extensions we made to our previous specification of the meta-locking algorithm (Figure 8) is presented in Figure 14.

In our model of the extra-fast version of the meta-locking algorithm, the basic behavior of a thread process becomes the following. A thread process loops forever, each time non-deterministically selecting one of the \textit{N} objects in the system for attempted monitor access by calling \texttt{monitor\_enter(Thread\_id, Object\_id)} followed by \texttt{monitor\_exit(Thread\_id, Object\_id)}. The \texttt{monitor\_enter} process tries to access the object following the extra-fast path as described above. If the object is not in state \textit{NEUTRAL}, the extra-fast path fails and the thread attempts to meta-lock the object synchronization data by calling \texttt{getmetalock}. Once the thread obtains the meta-lock (following the \textit{fast} or \textit{slow} path, as the case may be), it checks the lock state of the object (using \texttt{Release\_bits} of process \texttt{getmetalock}) to decide whether to invoke \texttt{monitor\_enter\_slow} or \texttt{releasemetalock} (which changes the object state to \textit{LOCKED}).

Process \texttt{monitor\_exit} works in a similar fashion. It first tries to release the monitor lock following the extra-fast path. If the extra-fast path fails, it calls \texttt{getmetalock} and checks the number of threads waiting in the monitor queue. If there are one or more, a wake-up signal is emitted via process \texttt{wait\_pool}. Finally, \texttt{releasemetalock} is invoked and object’s lock state is changed to \textit{WAITERS}. Figure 15 contains the XI specification of the \texttt{monitor\_enter} and \texttt{monitor\_exit} processes.

A thread calling \texttt{monitor\_enter} also sends a message (of type \textit{adding}) to process \texttt{count} of the corresponding object indicating it has placed a lock record in the queue. Similarly, a thread sends a message (of type \textit{removing}) to \texttt{count} while calling \texttt{monitor\_exit}. Process \texttt{count} enables a thread that is releasing a monitor lock to query the length of the monitor queue in order to determine whether it can follow the extra-fast path.

A thread calling \texttt{monitor\_enter\_slow} sends a message to process \texttt{wait\_pool} to indicate that it is waiting for a wake-up signal from a lock-releasing thread. Conversely, a thread releasing a monitor lock communicates with the \texttt{wait\_pool} process to check whether there are any threads waiting for a wake-up signal. If so, it sends a wake-up signal to \texttt{wait\_pool}, which relays this signal to the waiting threads and decrements the count of waiting threads by one. Figure 16 contains the
count(Object_id, X:integer) :=
From_Count_gate! (Object_id,status,X);
  count(Object_id,X)
#
To_Count_gate? (Object_id,adding,0);
  New_X is X + 1;
  count(Object_id,New_X)
#
To_Count_gate? (Object_id,removing,0);
  New_X is X - 1;
  count(Object_id,New_X).

wait_pool(Object_id, X) :=
To_Wait? (Object_id, adding, 0);
  New_X is X + 1;
  wait_pool(Object_id, New_X)
#
Wake_up? (Object_id, signaltopholds);
  From_Wait! (Object_id, signaltopholds,0);
  New_X is X - 1;
  wait_pool(Object_id, New_X)
#
From_Wait! (Object_id, status, X);
  wait_pool(Object_id, X).

Figure 16: (a) Process count. (b) Process wait_queue.

Consider next the behavior of an object process, the XL specification of which appears in Figure 17. As before (see the description of the object process in Section 4), process object coordinates with processes getmetalock and releasemetalock to effect an atomic SWAP and CAS (compare-and-swap) operation, respectively. Process object now additionally implements, in conjunction with processes monitor-enter and monitor_exit, atomic CAS operations as part of the extra-fast synchronization path.

To distinguish between the different types of CAS operations object may now engage in, a "request-type" field is included as the third argument of the messages received by object over channel cas(Object_id). As before, the second (and final) field of these messages is the Thread_id of the thread initiating the CAS operation with the object.

If a message of request-type extra_fast_lock_req is received, then the object is interacting with process monitor_enter (Figure 15(a)). In this case, it compares its multi-use word to the value m(0, neutral).\footnote{The predicate m is the type constructor for multi-use words in our XL specification. Also, the first 30 bits of the multi-use word of an object in the NEUTRAL lock state do not affect the logic of the extra-fast synchronization path. As such, they are represented abstractly in our specification as the value 0.} If the comparison succeeds—meaning that extra-fast locking may proceed—this value will be bound, via an input command, to Newword in process monitor_enter, and Newmult in process object will be bound to (Thread_id, locked). Consequently, the object will continue as object(m(Thread_id, locked), Object_id). Otherwise, Newword gets the value null and process monitor_enter calls getmetalock to follow the normal monitor-locking protocol.

If a message of request-type extra_fast_unlock_req is received, then the object is interacting with process monitor_exit (Figure 15(b)). In this case, it communicates with process count over
object(Multiuseword:multiword,Object_id:integer) ::= 

Swap ? (Object_id,Multiuseword,Multival) ; object(Multival,Object_id)
# 
Cas ? (Object_id,Thread_id,extra_fast_lock_req);
/* extrafast monitorlocking succeeds */
if Multiuseword == m(0,neutral) then {
    Cas_object_to_thread ! (Thread_id,Object_id,Multival,Multiuseword);
    Newmult = Multival
/* extrafast locking fails */
else {
    Cas_object_to_thread ! (Thread_id,Object_id,Multival,null);
    Newmult = Multiuseword};
object(Newmult, Object_id)
# 
Cas ? (Object_id,Thread_id,extra_fast_unlock_req);
From_Count_gate ? (Object_id,status,Q_length);
/* extrafast monitor unlocking succeeds */
if Multiuseword == m(Thread_id, locked) \ Q_length == 1 then {
    Cas_object_to_thread ! (Thread_id,Object_id,Multival,Multiuseword);
    Newmult = Multival
/* extrafast monitor unlocking fails */
else {
    Cas_object_to_thread ! (Thread_id,Object_id,Multival,null);
    Newmult = Multiuseword};
object(Newmult,Object_id).
# 
Cas ? (Object_id,Thread_id,release_metalock_req);
/* fast path of releasemetalock succeeds */
if Multiuseword == m(Thread_id,busy) then {
    Cas_object_to_thread ! (Thread_id,Object_id,Release_bits,Multiuseword);
    Newmult = Release_bits
/* fast path of releasemetalock fails */
else {
    Cas_object_to_thread ! (Thread_id,Object_id,Release_bits,null);
    Newmult = Multiuseword };
object(Newmult, Object_id).

Figure 17: XL specification of the object process.
channel `From_Count_gate` to obtain the length of monitor queue, which is bound to `Q_length`. It then checks if its multi-use word is equal to the value `(Thread_id, locked)` and whether `Q_length` is equal to 1. If so, extra-fast unlocking occurs: `Newword` in process `monitor_exit` is bound to the object’s multi-use word and `Newmult` in process `object` is bound to `(0, neutral)`. Otherwise, `Newword` is bound to null and `monitor_exit` calls process `getmetalock` in order to follow the normal monitor-locking protocol.

If a message of request-type `release_metalock_req` is received, then the object is interacting with process `releasemetalock`. This case has already been described in Section 4 and Figure 9. Now, however, in our model of the extra-fast version of the algorithm, we need to extend our previous encoding of process `releasemetalock` to properly update the lock-state of the object after the meta-lock has been released. In particular, if the meta-lock is released via the fast-path, the lock-state is updated according to the `Release_bits` provided by process `releasemetalock` in conjunction with the CAS operation.

We are left to consider the behavior of a thread that is unable to follow the extra-fast synchronization path due to object contention, and must therefore follow the “normal” protocol for monitor-locking and unlocking. In the case of monitor-locking, a thread comes to this realization when it finds the value of `Newword` to be null in process `monitor_enter`. At this point it gets the meta-lock and monitor-locks the object (by releasing the meta-lock with the release bits set to its thread id and a lock-state of LOCKED) if it finds the object in a lock-state of NEUTRAL or WAITERS. Otherwise, the object is already locked and `monitor_enter_slow` is invoked.

Process `monitor_enter_slow` releases the meta-lock and declares itself a waiter by outputting an adding message to process `wait_pool` over channel `To_Wait` and blocking at the input statement over channel `From_Wait`. A message will eventually be delivered over this channel by way of a lock-releasing thread, and the waiting thread that succeeds in executing the matching input statement will be allowed to resume its attempt to monitor-lock the object.

Note that the choice of thread to execute the matching input statement over channel `From_Wait` is purely nondeterministic. In essence, this means that we are modeling a pool of waiting threads, rather than, for example, a queue of waiting threads. This choice of data structure is consistent with the strategy outlined in Section 6.3 of [ADG+99], which makes it “possible to find and give preference to awakened waiters without searching [the monitor queue].”

In the case of monitor unlocking, a thread realizes it must follow the normal protocol when it finds the value of `Newword` to be null in process `monitor_exit`. At this point it follows the protocol for monitor unlocking described above in Section 5.1. The noteworthy aspects of our XL encoding of this protocol are the following. Process `monitor_exit` communicates with processes `count` and `wait_pool` to respectively determine the values of `Q_length`, the size of the pool of threads waiting to monitor-lock the object, and `W_length`, the number of threads in this pool that are waiting for a wakeup signal from a lock-releasing thread. `Q_length` is used by the process to decide if the meta-lock should be released with a lock-state of WAITERS or NEUTRAL. `W_length` is used to determine
if \texttt{monitor\_exit\_slow} should be invoked. Process \texttt{monitor\_exit\_slow} completes the slow path for monitor unlocking by signaling the threads waiting for a wakeup signal and then releasing the meta-lock. The former is accomplished via a message to process \texttt{wait\_pool} (over channel \texttt{Wake\_up}) which in turn relays this message to the awaiting threads via a message over channel \texttt{From\_Wait}.

6 Model-Checking Results

We have used XMC to model check our XL specifications of the meta-locking algorithm with regard to three essential correctness properties: mutual exclusion, freedom from deadlock, and freedom from lockout.\footnote{The terminology \textit{freedom from lockout} is from [ADG+99], and refers to the liveness property that every thread that attempts to obtain the meta-lock will eventually obtain the meta-lock.} Collectively, satisfaction of these properties ensures the algorithm's correctness.

XMC is a model checker for the modal mu-calculus [Koz83] and, as such, we have expressed the properties of interest as mu-calculus formulas. The mu-calculus is a low-level yet highly expressive temporal logic whose expressive power subsumes that of many other temporal logics, including LTL, CTL, and CTL* [Eme90]. XMC has its own syntax for the mu-calculus, the details of which can be found in [S+98]. A basic component of this syntax are the observable actions. Given an XL system specification \textit{Sys}, an observable action is a term \textit{t} with the special designation \textit{action}(\textit{t}) in \textit{Sys} (see Section 2). In the case of the meta-locking algorithm, the observable actions are those of the form \textit{got\_metalock(I,J)}, \textit{released\_metalock(I,J)}, and \textit{requesting\_metalock(I,J)}, where \textit{I} is a thread id and \textit{J} is an object id.

Consider first the formula for mutual exclusion, which ensures that at most one thread can acquire the meta-lock of an object at any time. A direct encoding of this property in the mu-calculus requires a greatest fixed point computation. Since XMC computes least fixed points more efficiently than greatest fixed points, the formula we have actually used is one that is true if there is no mutual exclusion:

\[
\text{nomutex}(I) \equiv \langle \text{got\_metalock}(_,I) \rangle \text{formula}(I) \lor \langle \text{nomutex}(I) \rangle.
\]
\[
\text{formula}(I) \equiv \langle \text{got\_metalock}(_,I) \rangle \text{tt} \lor \langle \text{released\_metalock}(_,I) \rangle \text{formula}(I).
\]

The formula states that there exists a path such that a thread gets the meta-lock for object \textit{I} and subsequently some other thread also gets the meta-lock for object \textit{I} before the previous thread releases the meta-lock. This formula is false in all instances of the basic and extended models we considered, and hence mutual exclusion is ensured. Intuitively, this is the case because the swap operations between an object and a thread are atomic. Consequently, two threads cannot simultaneously swap in their \texttt{busyBits} for the object's multi-use word and both find the object's lock state to be \texttt{NON-BUSY}.

The following formula states that a deadlocked system state is reachable:
deadlock \Leftarrow \emptyset \text{ ff } \Rightarrow \leftrightarrow \text{ deadlock.}

This formula is false in all instances of the basic and extended models we considered, and thus these models are free from deadlock.

The formula \textit{liveness}(I,J) encodes lockout freedom and is true if thread I, after requesting the meta-lock for object J, is assured of eventually getting the meta-lock. \footnote{The formula actually states that, after thread I requests the meta-lock for object J, along all paths, the following should hold true: (a) after every meta-lock request for the same object J, eventually \textit{gotmetalock}(I,J) is true; and (b) after any action other than a meta-lock request, \textit{gotmetalock}(I,J) is true eventually. The formula may seem more complex than what is needed to express livelock freedom, however, the XMC model checker currently supports only the alternation-free fragment of the modal mu-calculus; therefore the formula must be “beefed up” so that only fair execution paths with respect to object J are considered.}

\text{\textit{liveness}(I,J) \Leftarrow [\textit{requesting\_metalock}(I,J)] \text{ formula}(I,J) \ \land \ \neg \text{ liveness}(I,J).}

\text{\textit{formula}(I,J) \Leftarrow <\textit{got\_metalock}(I,J)> \text{ tt } \land \}
\text{\textit{([requesting\_metalock(_, J1)] ([J1 \Leftarrow J] \ \land \ \text{ formula}(I,J)) \ \land \}
\text{\neg \{\text{requesting\_metalock(_, _)}\} \text{ formula}(I,J)).}

This formula is true in all instances; intuitively, this is the case since threads contending for a meta-lock are totally ordered by the order in which they executed the swap instruction, and such an order is inherently fair.

Formula \textit{liveness1}(I,J) states that it is always the case that a lock record inserted into a monitor queue by a thread eventually reaches the head of the queue (at which point the thread will have monitor-locked the object). It is pertinent to the extended specification, which models monitor locking as well as meta-locking.

\text{\textit{liveness1}(I,J) \Leftarrow [\textit{requesting\_monitor}(I,J)] \text{ f1}(I,J) \ \land \ \neg \text{ liveness}(I,J).}

\text{\textit{f1}(I,J) \Leftarrow <\textit{got\_monitor}(I,J)> \text{ tt } \land \text{\{\textit{[requesting\_monitor(_, J1)] ([J1 \Leftarrow J] \ \land \ \text{ f1}(I,J)) \ \land \}
\text{\neg \{\text{requesting\_monitor(_, _)}\] \text{ f1}(I,J))}.}

The formula, as expected, is not true in the extended model since lock records may be processed \textit{out-of-order} (see Section 5).

We also performed several “sanity checks” on the specifications, such as “breaking up” the atomic swap operation, which should lead to a violation of the mutual exclusion property, and inserting an infinite-loop process between invocations of the \textit{getmetalock} and \textit{releasemetalock} processes, which should lead to a violation of the lockout-freedom property. Our models of the algorithm indeed passed these sanity checks.

Tables 1 and 2 summarize our model-checking results for the basic model and the extended model, respectively. Results are given for a variety of values for the pair \((M, N)\), where \(M\) is the
number of threads and $N$ is the number of objects. The number of states and transitions for each configuration was computed using a kind of depth-first search query. The reported values for CPU time and memory usage are for formula \textit{deadlock}, as checking this formula requires a complete traversal of the specification’s underlying state space. A blank entry in either of these columns indicates that XMC was unable to terminate the case in question before exhausting memory. For each of the cases that did complete, XMC reported that \texttt{nomutex(I)} and \texttt{deadlock} were false, \texttt{liveness(I,J)} was true, and \texttt{liveness1(I,J)} was false, as expected. The first three of these formulas were checked on all instances of both the basic and extended models, while \texttt{liveness1(I,J)} was checked on all instances of the extended model only. As noted above, this formula models freedom from lockout for monitor locking, and is thus only applicable to the extended model. All results were obtained using version 2.5 of XSB and Mandrake Linux 8.1 on a 1.7GHz Xeon processor with 2GB of RAM.

### 7 Conclusions

We showed that it is possible to verify a critical component of a high-performance Java virtual machine using model-checking techniques. Our XL model of the Java meta-locking algorithm is fully parameterized, both on the number of threads $M$ and number of objects $N$. It also captures the \textit{extra-fast} mode of locking and unlocking of uncontended objects. Using XMC, we show that for
a variety of values of $M$ and $N$, the meta-locking algorithm provides mutual exclusion and freedom from lockout. Such results should enable Java virtual machine implementors to obtain a high level of confidence in the language’s implementation. On the other hand, our results demonstrate that the monitor-locking protocol of [ADG+99] is not lockout-free, with the protocol’s designers having chosen to give equal preference to awaiting threads and newly arrived threads.

Recent developments in the XMC model-checking project show that it is possible to use tabled resolution and deduction to verify, in a highly automated fashion, properties of parameterized systems [RKR+00]. A parameterized system, of which the meta-locking algorithm is an example, represents an infinite family of systems, each of which is finite state. As future work, we intend to apply these techniques to the meta-locking algorithm. If successful, this would allow us to perform model checking on the general algorithm, i.e. for any values of $M$ and $N$. Preliminary work in this direction has been reported in [RR01].

**Acknowledgments** The authors are grateful to Yifei Dong, C.R. Ramakrishnan, and David Warren for fruitful discussions about using XMC to analyze the meta-locking algorithm. Yifei also pointed out a problem with an earlier version of the liveness formula liveness($I$,$J$). Thanks also to Y.S. Ramakrishna, who brought the meta-locking algorithm to our attention, answered our questions about the algorithm, and provided extensive feedback on a draft of this paper. Research supported in part by NSF grants CCR-9705998, CCR-9988155 and CCR-0205376, and ARO grants DAAD190110003 and DAAD190110019.

**References**


