

Model Checking the Java Meta-Locking Algorithm

Samik Basu, Scott A. Smolka, Orson R. Ward
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, USA
{bsamik, sas, ward}@cs.sunysb.edu

Abstract

We apply the XMC model checker to the Java meta-locking algorithm, a highly optimized technique for ensuring mutually exclusive access by threads to object monitor queues. Our abstract specification of the meta-locking algorithm is fully parameterized, both on M , the number of threads, and N , the number of objects. Using XMC, we show that for a variety of values of M and N , the algorithm indeed provides mutual exclusion and freedom from lockout.

1 Introduction

Given the growing importance of JavaTM as a concurrent object-oriented language for implementing internet-based applications, it is crucial that the language's implementation on a Java virtual machine 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 reported in [1] 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. [1] 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, for the purposes of this paper, can be viewed as a FIFO queue of *lock records*. Each lock record represents a waiting thread ready to enter the object's

monitor. A thread gains access to the monitor when its lock record reaches the head of the queue.

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 re-

quests, 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.

The authors of [1] 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 [7], can be brought to bear on the problem. Model checking [2] 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 lockout. Some of these cases were particularly challenging, with state space sizes in the millions. Collectively, our results provide strong testimony to the correctness of the meta-locking algorithm.

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., [3]. Perhaps most closely related is the work reported in [6], 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.

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. Section 4 discusses our model of the algorithm. Section 5 contains our model-checking results, while Section 6 offers some concluding remarks.

2 The XMC Verification Tool

XMC [7], developed here at SUNY Stony Brook, is a model checker for a value-passing process calculus [5] and

E	\longrightarrow	$Comp$	(computation)
		$in(Term)$	(input communication)
		$out(Term)$	(output communication)
		$zero$	(unit deadlocked process)
		$E \circ E$	(sequential composition)
		$E \# E$	(choice)
		$if(Comp, E, E)$	(conditional)
		$E E$	(parallel composition)
		$E \setminus PortSet$	(restriction)
		$E @ PortMap$	(relabeling)
		$Proc$	(process invocation)
Def	\longrightarrow	$(Proc ::= E)^*$	(process definitions)

Figure 1. Syntax of XL.

the modal mu-calculus [4]. A novelty of the system is that it is written in a highly declarative fashion, in just under 200 lines of XSB tabled Prolog code. XSB [11] 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 [5]. 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 XL specifications is given by the grammar of Figure 1 where $Proc$ is a (parameterized) process name, represented as a term (e.g., $channel(N, Buf)$) and $Comp$ is a term (e.g., $X \text{ is } Y+1$) representing a computation. A terminating null process is represented by $true$, the empty computation. A process $if(C, S_1, S_2)$, 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. A process $in(chan, t)$ inputs a value that matches term t over port $chan$; $out(chan, t)$ outputs the value represented by term t over port $chan$. Process invocations may be recursive; in fact, since the language provides no iterative constructs, recursion is the only way to specify loops in processes. As in CCS, relabeling and restriction are used to derive instances of a generic process. $PortSet$ is a set of terms that represent the restricted ports, and $PortMap$ is a list of pairs of terms of the form s/t indicating that s is relabeled to t .

The complete specification of the Alternating Bit Protocol [10] in Figure 2 illustrates the features of XL. Text preceded by the $\%$ character are comments.

```

channel ::= in(get,Data) o (out(put,Data) #
                        out(drop) ) o channel.

sender(Seq) ::=
% Seq is seq. # next frame to be sent
out(dataOut,Seq) o
((in(ackIn,AckSeq) o
  if(AckSeq == Seq
    , NSeq is 1-Seq o sender(NSeq)
    % successful ack, next message
    , sender(Seq))
  % unexpected ack, resend message
) # sender(Seq)).
% upon timeout, resend message

receiver(Seq) ::=
% Seq is expct'd seq. # nxt frame to be rcv'd
in(dataIn,RecSeq) o
if(RecSeq == Seq
  , (NSeq is 1-Seq o out(ackOut,RecSeq) o
    receiver(NSeq))
  , out(ackOut,RecSeq) o receiver(Seq)).
% unexpected seq, resend ack

abp ::=
(sender(0) @ [s2r_in(X)/dataOut(X),
             r2s_out(X)/ackIn(X)]
| channel @ [s2r_in(X)/get(X),
             s2r_out(X) / put(X)] % sndr -> rcvr
| channel @ [r2s_in(X)/get(X),
             r2s_out(X) / put(X)] % rcvr -> sndr
| receiver(0) @ [s2r_out(X)/dataIn(X),
                r2s_in(X) / ackOut(X)]
)\{s2r_in(_),s2r_out(_),r2s_in(_),
   r2s_out(_)}.
```

Figure 2. Specification of the Alternating Bit Protocol in XL.

3 The Meta-Locking Algorithm

In this section we describe the meta-locking algorithm. As in [1], we use Java-style pseudo-code for this purpose. In the meta-locking algorithm, threads observe a protocol when manipulating an object's synchronization data. As described in Section 1, an object's synchronization data is essentially a FIFO queue of lock records. 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.
2. *Manipulate* the synchronization data.

```

typedef struct execenv {
  Thread          thread;
  mutex_t         metaLockMutex;
  condvar_t       metaLockCondvar;
  bool_t          gotMetaLockSlow;
  bool_t          bitsForGrab;
  BitField        metaLockBits;
  ExecEnv         *succEE;
} ExecEnv;
```

Figure 3. Per-thread Execution Environment.

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.

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 multi-use word, which is actually the second word of a two-word object header (the first word points to the object's class). [1] distinguishes four forms of the multi-use word, but for our purposes, two suffice: a (possibly null) pointer to the lock record at the head of the object's monitor queue followed by a *lock-state* value of non-busy, when the object is not meta-locked; or a pointer to the EE of the thread that has the object's meta-lock followed by a lock-state value of busy, when the object is meta-locked. Besides its role in implementing meta-locking, the multi-use word is used for other purposes, e.g., to store garbage-collector-relevant data. Hence its name.

A thread releasing a meta-lock must update the object in question's multi-use word to reflect the new state of the object's monitor queue. [1] refers to the new multi-use word as the thread's *release bits*. The lock state of a thread's release bits will always be non-busy.

The definition of the EE data structure is given in Figure 3. `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 nothing more than 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). `metaLockCondavar` 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.

```

BitField getMetaLock(ExecEnv *ee,
Object *obj,
Object *obj) {
    BitField busyBits = ee | BUSY;
    BitField lockBits =
        SWAP(busyBits, multiUseWord(obj));
    return getLockState(lockBits) != BUSY ?
        lockBits : getMetaLockSlow(ee, lockBits);
}

void releaseMetaLock(ExecEnv *ee, Ob-
ject *obj,
BitField releaseBits) {
    BitField busyBits = ee | BUSY;
    BitField lockBits = CAS(releaseBits,
        busyBits, multiUseWord(obj));
    if (lockBits != busyBits)
        releaseMetaLockSlow(ee, releaseBits);
}

```

Figure 4. Fast paths for meta-lock operations.

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. gotMetaLockSlow 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 pseudo-code for acquiring and releasing meta-locks is presented in Figure 4. 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 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 5. 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 the mutex. 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.

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

```

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->metaLockCondavar,
                       &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 5. Slow paths for meta-lock operations.

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 $\text{meta}_j(M, N)$, 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, the names of many of which are parameterized by thread and object ids to indicate the intended communicants. The purpose of these channels will be made clear in the ensuing discussion. An architectural diagram of the specification is given in Figure 6 for the case of two threads and one object.

A thread process is of the form $\text{thread}(\text{Thread_id}, N)$, where Thread_id is an integer between 1 and M inclusively, uniquely identifying the thread in question. Although XL processes are parameterized, they do not have “state” per se. We therefore use thread ids instead of EEs 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 $\text{getmetalock}(\text{Thread_id}, \text{Object_id})$ followed by $\text{releasemetalock}(\text{Thread_id}, \text{Object_id})$.

An object process is of the form $\text{object}(\text{Multiuseword}, \text{Object_id})$, where Object_id is an integer between 1 and N inclusively, uniquely identifying the object in question, and Multiuseword is of one of two forms: Thread_id concatenated with busy or Thread_id concatenated with not_busy . An object process supports the two types of atomic swap operations utilized in the fast-path meta-lock operations (Figure 4). The encoding of these operations is perhaps one of the most interesting aspects of the specification.

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

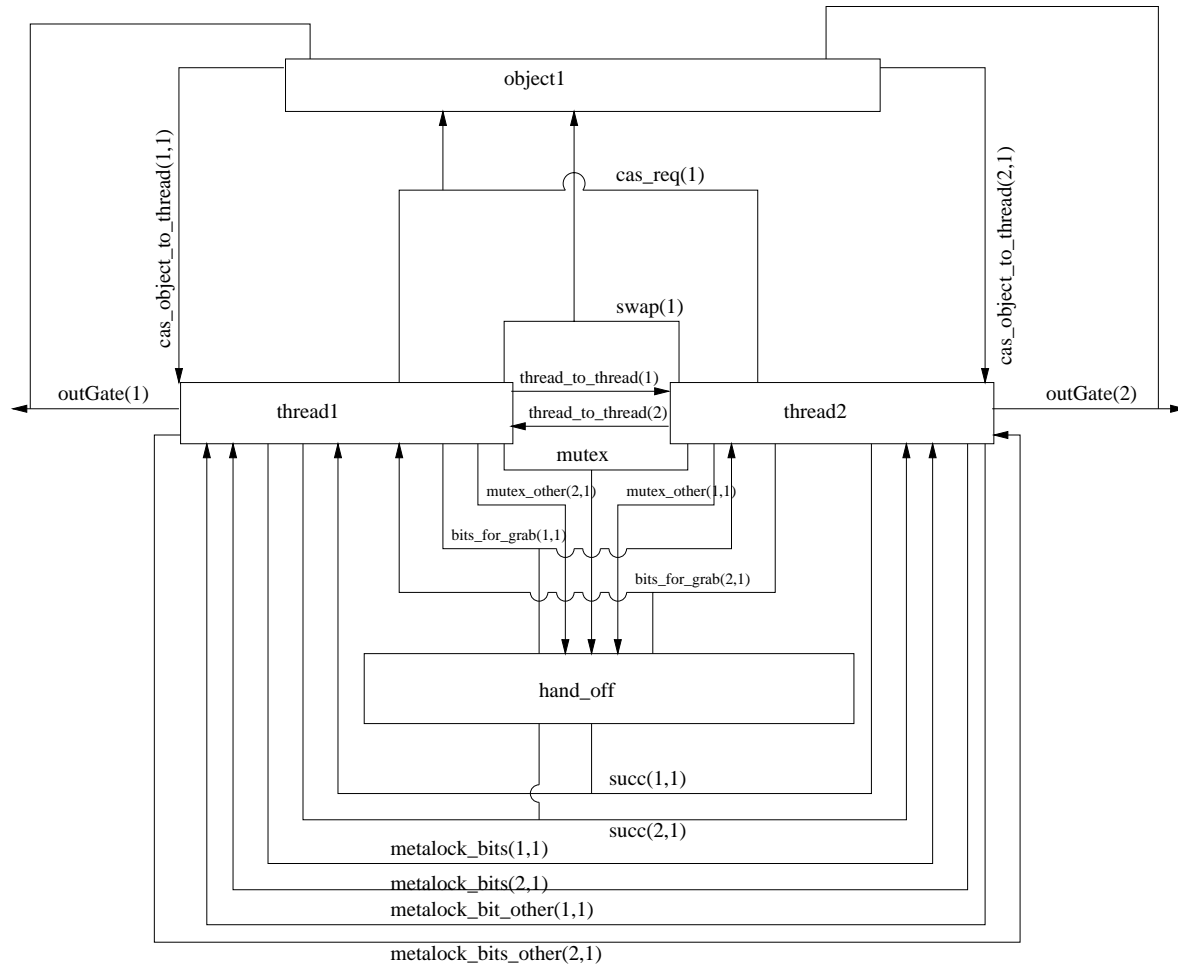


Figure 6. Architecture of the specification for two threads and one object.

```

hand_off ::=
in(mutex, (Thread_id, Object_id)) o
in(mutex_other(Thread_id, Object_id)) o
(
  out(bits_for_grab(Thread_id, Object_id), false)
  #
  out(succ(Thread_id, Object_id), 0)
) o
hand_off.

```

Figure 8. XL code for the hand-off process.

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 7. The thread and object processes initiate the operation by synchronizing on a `Thread_id` message over channel `cas_req(Object_id)`. 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.

The purpose of the hand-off process—see Figure 8—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()` 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 9 completes the picture as to how we modeled the hand-off protocol. `getmetalockslow()` first tries to

get the mutex of the predecessor and then examines `Bits_for_grab`, which it inputs from channel `bits_for_grab(Pred_id, Object_id)`, to determine if it won the race. The “trick” here is that the value of `Bits_for_grab` 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 `Bits_for_grab` is false. In the latter case, the predecessor won the race and the value of `Bits_for_grab` 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 `Bits_for_grab` 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 port `metalock_bits_other`.

5 Model-Checking Results

We have used XMC to model check our XL specification of the meta-locking algorithm with regard to three essential correctness properties: mutual exclusion, freedom from deadlock, and freedom from lockout. Collectively, satisfaction of these properties ensures the algorithm’s correctness.

XMC is a model checker for the modal mu-calculus [4] and, as such, we have expressed the properties of interest as mu-calculus formulas. 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 description of XMC’s syntax for the mu-calculus can be found in [9]). 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:

```

nomutualex(I) += diam([out(outGate(_),
  got_metalock(I))], formula(I)) \ /
  diam(-nil, nomutualex(I)).

formula(I) += diam([out(outGate(_),
  got_metalock(I))], tt) \ /
  diam(-[out(outGate(_),
  released_metalock(I))], formula(I)).

```

The formula states that there exists a path such that a thread gets the meta-lock for object `I` and subsequently some other

```

releasemetallock(Thread_id, Object_id) ::=
out(cas_req(Object_id), Thread_id) o
in(cas_object_to_thread(Thread_id, Object_id),
    Newword) o
if(Newword == null,
    releasemetallockslow(Thread_id, Object_id),
    true).

```

```

object(Multiuseword, Object_id) ::=
in(cas_req(Object_id), Thread_id) o
if(Multiuseword == (Thread_id, busy),
    out(cas_object_to_thread(Thread_id,
        Object_id), Multiuseword) o
    object((Thread_id, not_busy), Object_id),
    out(cas_object_to_thread(Thread_id,
        Object_id), null) o
    object(Multiuseword, Object_id)).

```

Figure 7. XL code for the atomic compare-and-swap operation.

```

getmetallockslow(Thread_id, Pred_id,
    Object_id) ::=
out(mutex_other(Pred_id, Object_id)) o
in(bits_for_grab(Pred_id, Object_id),
    Bits_for_grab) o
if(Bits_for_grab == false,
    out(succ(Pred_id, Object_id), Thread_id) o
    in(metallock_bits_other(Thread_id,
        Object_id)),
    in(metallock_bits(Pred_id, Object_id))) o
in(thread_to_thread(Pred_id),
    hand_off(Object_id)).

```

```

releasemetallockslow(Thread_id) ::=
out(mutex, (Thread_id, Object_id) o
in(succ(Thread_id, Object_id), Succ_id) o
if(Succ_id \= 0,
    out(metallock_bits_other(Succ_id,
        Object_id)),
    out(bits_for_grab(Thread_id, Object_id),
        true) o
    out(metallock_bits(Thread_id, Object_id))) o
out(thread_to_thread(Thread_id),
    hand_off(Object_id)).

```

Figure 9. XL code for slow-path operations.

thread also gets the meta-lock for object I before the previous thread releases the meta-lock. Note that the messages output on channel `outGate` are signals that reflect the state of the meta-locks in the system. There is one `outGate` channel per thread and `outGate(_)` denotes “any” `outGate`. Intuitively, this formula will be false for the meta-locking algorithm, and hence mutual exclusion is ensured, because the swap operations between an object and a thread are atomic. Consequently, two threads cannot simultaneously swap in their `busyBits` for the object’s multi-use word and both find the object’s lock state to be non-busy.

The following formula states that a deadlocked system state is reachable:

```

dlreach += box(-nil, ff) \ /
    diam(-nil, dlreach).

```

As the results below indicate, this formula is false and thus the model is free from deadlock.

The formula `liveness(I, J)` encodes livelock freedom and is true if thread I , after requesting the meta-lock for object J , is assured of eventually getting the meta-lock.

```

liveness(I, J) -= box(out(outGate(I),
    requesting_metallock(J)), formula(I, J))
    \ / box(-nil, liveness(I, J)).

```

```

formula(I, J) += diam(out(outGate(I),
    got_metallock(J)), tt) \ / form(I, J)
    \ / box(-nil, formula(I, J)).

```

```

form(I, J) += diam(out(outGate(I),
    got_metallock(J)), tt) \ /
    box(-[out(outGate(_),
    requesting_metallock(_))], form(I, J)).

```

Intuitively, this formula is true of the model 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.

We also performed several “sanity checks” on the specification, 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 `getmetallock()` and `releasemetallock()` processes, which should lead to a violation of the lockout-freedom property. Our model of the algorithm indeed passed these sanity checks.

Table 1 summarizes our model-checking results 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

(M, N)	No. States	No. Trans.	Sec.	Bytes
(2, 1)	127	228	0.03	387304
(2, 2)	1757	4240	0.47	1166192
(2, 3)	16955	49572	6.75	7788336
(2, 4)	138425	469440	75.45	55163960
(2, 5)	1024375	3930500	1169.36	361041600
(2, 6)	7103125	30330000		
(3, 1)	1128	2724	0.37	1016816
(3, 2)	31745	100110	18.21	20475684
(3, 3)	556738	2091978	1230.94	347450776
(3, 4)	7800849	33547164		
(4, 1)	10989	33048	6.66	8326456
(4, 2)	540777	2076272	1744.66	422503004
(5, 1)	122898	438120	161.48	103164388
(6, 1)	1594827	6531228		

Table 1. Table of model-checking results.

values for CPU time and memory usage are for formula `dlreach`, as checking this formula requires a complete traversal of the specification's underlying state space. A blank entry in these columns indicates that XMC was unable to terminate on the case in question before exhausting memory. For each of the cases that did complete, XMC reported that `nomutualex(I)` and `dlreach` were false and `liveness(I, J)` was true, as desired. All results were obtained on a Sun Ultra-Enterprise workstation with two 336 MHz CPUs and 2 GB of RAM.

6 Conclusions

We showed that it is possible to verify a critical component of a high-performance Java virtual machine using model-checking techniques. As future work, we intend to apply the technique of [8] for verifying parameterized systems to attain a general correctness proof of the meta-locking algorithm, i.e., for any M and N .

Acknowledgements 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. Special thanks 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.

References

[1] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of OOPSLA '99*, 1999.

- [2] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.
- [3] J. F. Groote and M. Rem, editors. *Science of Computer Programming, Special Issue on Verification and Validation Methods for Formal Descriptions*, volume 29(1-2), July 1997.
- [4] D. Kozen. Results on the propositional μ -calculus. *Theoretical Computer Science*, 27:333–354, 1983.
- [5] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
- [6] R. Pike, D. Presotto, K. Thompson, and G. Holzmann. Process sleep and wakeup on a shared-memory multiprocessor. In *Proceedings of the Spring 1991 EurOpen Conference*, pages 161–166, Tromsø, Norway, 1991.
- [7] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. W. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of CAV '97*, LNCS Vol. 1254, 1997. Springer-Verlag.
- [8] A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, and S.A. Smolka. Verification of parameterized systems using logic-program transformations. In *Proceedings of TACAS 2000*. Springer-Verlag, 2000.
- [9] S.A. Smolka et al. Logic programming and model checking. In *Proceedings of PLILP/ALP '98*, LNCS Vol. 1490, 1998. Springer-Verlag.
- [10] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.
- [11] XSB. The XSB logic programming system v2.01, 1999. Available by anonymous ftp from `ftp.cs.sunysb.edu`.