Report

An ASM specification of C# threads and the .NET memory model

Author(s):
Stärk, Robert F.; Börger, Egon

Publication Date:
2003

Permanent Link:
https://doi.org/10.3929/ethz-a-006731851 →

An ASM specification of C# threads and the .NET memory model

Robert F. Stärk[1] and Egon Börger[2]

[1] Computer Science Department, ETH Zürich
staerk@inf.ethz.ch
[2] Dipartimento di Informatica, Università di Pisa
boerger@di.unipi.it

Abstract. We present a high-level ASM model of C# threads and the .NET memory model. We focus on purely managed, fully portable threading features of C#. The sequential model interleaves the computation steps of the currently running threads and is suitable for uniprocessors. The parallel model addresses problems of true concurrency on multiprocessor systems. The models provide a sound basis for the development of multi-threaded applications in C#. The thread and memory models complete the abstract operational semantics of C# in [2].

1 Introduction

Modern object-oriented programming languages like Java or C# support multi-threaded programming. They allow several threads to run concurrently sharing objects on the heap in the same address space. Each thread has its own frame stack, program counter, local variables and registers. The languages have special syntactical constructs for synchronization. Java has a `synchronized` statement and `synchronized` methods, while C# has a `lock` statement and several attributes that can be applied to classes and methods to control their run-time synchronization behavior.

Although the C# programming languages supports multi-threaded programming directly via special syntax, the underlying thread model is poorly documented and still considered to be part of the library. The Ecma standards for C# [4] and the *Common Language Infrastructure* [5] contain only a few paragraphs about threads. For example, the `lock` statement is defined in [4, §15.22] by a reduction to the library functions `Monitor.Enter` and `Monitor.Exit` which are not further specified there. Important issues, such as the order of writes to volatile and non-volatile fields, are just briefly mentioned in two paragraphs in [4, §10.10, §17.4.3]. Hence, a program developer has to rely solely on the class library documentation that comes with Microsoft's .NET framework Software Development Kit [11]. Unfortunately, that documentation is not very precise with respect to threads, locks and memory issues. Moreover, it is not identical with the (XML) specification of the types that comprise the standard libraries in [5, Partition IV, Profiles and Libraries]. For example, specifications

of `Thread.Interrupt`, `Thread.Suspend` and `Thread.Resume` are not included in [5].

If a programmer cannot rely on a simple and precise thread model, the task of writing reliable multi-threaded applications that are correctly synchronized and free of data races and deadlocks becomes very difficult and tedious. Multi-threaded programs depend on the scheduling policy of underlying run-time system and therefore synchronization errors are difficult to reproduce and to debug. Moreover, certain problems may only occur under heavy threading stress in production environments like web services which cannot be simulated during the development cycle. Tools that statically analyze multi-threaded programs for synchronization problems are in general neither sound nor complete. Nevertheless, in some cases the may report a high percentage of all possible conflicts (see [19]).

The Java Language Specification [7, Ch. 17] devotes a whole chapter to threads and locks. However, that specification has been found to be hard to understand and has subtle, often unintended, implications. Therefore, the Java community has proposed a new specification of the semantics of threads and locks often referred to as the *New Java Memory Model* [10]. Whether the new specification is easier to understand may be doubted. It justifies at least most of the common compiler optimizations which were prohibited by the old one. For a comparison and analysis of the different proposals we refer to [1].

The specification of threads in this article extends the modular definition of the semantics of C# in [2] by a new module $\mathrm{C}\#_{\mathcal{T}}$ for multi-threaded C#. We focus on purely managed, fully portable threading features of C# and the .NET common language runtime. We do not consider the .NET equivalents of Win32 threading primitives such as `WaitHandle` and their derived classes. We also do not model asynchronous delegates and synchronization domains. The starting point of our model has been the thread model for Java in [17]. That model however is only correct for uniprocessor systems and does not address problems of true concurrency.

For basic terminology on Abstract State Machines we refer the reader to [3,8].

2 Threads in Microsoft's .NET framework

The thread related features of C# are collected in the `System.Threading` namespace (see Fig. 1). The namespace contains the delegate type `ThreadStart` that denotes the type of functions with zero arguments and return type `void`. The most important classes of the namespace are the `Thread` and `Monitor` classes. Several thread related exception classes derived from `SystemException` are also declared in the namespace.

A thread can be in one or more states of the `ThreadState` enumeration (listed in Fig. 1). Unfortunately, the documentation does not state clearly which combinations of states are allowed for a thread and which are not. Moreover, some of the states are not real execution states of a thread but just boolean flags. The `Background` state, for example, tells the run-time system that it can kill the

thread and exit when all non-background threads have terminated (similar to the `Daemon` property of threads in Java). Other states, like `StopRequested`, are for internal use only and should not be exposed to the programmer in a public enumeration. The `Aborted` state has a rather obscure meaning (see below). If there is an `AbortRequested`, why is there no `InterruptRequested`?

The `ThreadState` property of the `Thread` class returns a snapshot containing the states of a thread as a bitset. This information, however, cannot be used for synchronization purposes, since it may already been outdated when it is obtained. Therefore, we do not model the `ThreadState` property below and use a different set of execution states in our model.

Threads are represented in C# by instances of class `Thread` in Fig. 2. Unlike in Java, this class is `sealed` (`final` in Java terminology) and cannot be subclassed. The constructor of the class takes a pointer to a `ThreadStart` function which will be executed when the new thread is started. The two static methods of the class, `Sleep` and `ResetAbort`, are implicitly called on the current thread.

The constructor of class `Monitor` in Fig. 2 is private, which means that no instances of this class can be created. The reason is, that in C# (like in Java) every object reference can be used as a monitor and therefore there is no need to create special monitors. The `Monitor` class contains only static methods. Its `Wait`, `Pulse` and `PulseAll` methods are similar to Java's `wait`, `notify` and `notifyAll` methods of class `java.lang.Object`.

The `Enter` and `Exit` methods of the `Monitor` class are used to syntactically reduce the lock statement of C# (where o is a fresh local variable):

$$\texttt{lock (}exp\texttt{)}\ stm \quad\Longrightarrow\quad \begin{cases} \texttt{object}\ o\ \texttt{=}\ exp\texttt{;} \\ \texttt{Monitor.Enter(}o\texttt{);} \\ \texttt{try \{}\ stm\ \texttt{\}} \\ \texttt{finally \{ Monitor.Exit(}o\texttt{); \}} \end{cases}$$

Unlike in Java, the `Monitor.Enter` and `Monitor.Exit` methods can be called explicitly in C# programs and hence C# cannot guarantee that a thread holds no more locks when it has terminated.

3 An ASM model for threads on uniprocessors

Whenever in C# an object is created on the heap, it gets two additional overhead fields associated with it. The first field is a pointer to the object's method table. This pointer makes it possible to obtain the run-time type (exact type) of the object. The second field contains an index of a SyncBlock. SyncBlocks are associated with an object on the fly when the object is used as a monitor. A SyncBlock structure contains information that is used for thread synchronization (cf. [14]).

```
namespace System.Threading {
  delegate void ThreadStart();
  enum ThreadState {...}
  ...
  sealed class Thread {...}
  sealed class Monitor {...}
  ...
  class ThreadStateException  {...}
  class ThreadAbortException {...}
  class ThreadInterruptedException {...}
  class SynchronizationLockException {...}
}
```

```
enum ThreadState {
  Running = 0,
  StopRequested = 1,
  SuspendRequested = 2,
  Background = 4,
  Unstarted = 8,
  Stopped = 16,
  WaitSleepJoin = 32,
  Suspended = 64,
  AbortRequested = 128,
  Aborted = 256
}
```

Fig. 1. The `System.Threading` namespace and the `ThreadState` enumeration.

```
sealed class Thread {
  Thread(ThreadStart start);
  void Start();
  bool Join(int msec);
  static void Sleep(int msec);
  void Abort();
  static void ResetAbort();
  void Interrupt();
  void Suspend();
  void Resume();
  ...
}
```

```
sealed class Monitor {
  private Monitor() { }
  ...
  static void Enter(object obj);
  static void Exit(object obj);
  static bool Wait(object obj, int msec);
  static void Pulse(object obj);
  static void PulseAll(object obj);
  ...
}
```

Fig. 2. The `Thread` class and the `Monitor` class.

3.1 The vocabulary for threads and monitors

We abstract from implementation details and assume that the dynamic function $runTimeType\colon Ref \to Type$ returns for every object reference its run-time type. The set of threads can then be defined as follows:

$$Thread = \{ref \in Ref \mid runTimeType(ref) = \texttt{Thread}\}$$

The subuniverse $Monitor \subseteq Ref$ is equipped with a dynamic function $lockOwner$ which returns the thread that currently owns the lock of the monitor, a $lockCount$ which counts how many times a thread has to exit the monitor before the lock is released, a $readyQueue$ (also known as *lock queue*) which holds the ordered queue of blocked threads that are ready to acquire the lock, and a $waitQueue$ which holds the ordered queue of threads that are waiting on the monitor.

$lockOwner\colon Monitor \to Thread \cup \{None\}$
$lockCount\colon Monitor \times Thread \to \mathbb{N}$
$readyQueue\colon Monitor \to List(Thread)$
$waitQueue\colon Monitor \to List(Thread)$

When an object *ref* is used as a *Monitor* the functions are initialized as follows:

$$\begin{array}{ll} lockOwner(ref) := None & readyQueue(ref) := [\,] \\ lockCount(ref, thread) := Undef & waitQueue(ref) := [\,] \end{array}$$

The possible execution states of a thread (explained in detail below) are:

$$\begin{array}{rl} ExecState ::= & Unstarted \mid Active \mid Suspended \mid Sleeping \mid Joined \\ \mid & Syncing \mid Waiting \mid Pulsed \mid Dead \end{array}$$

The function *execState* returns the unique execution state of a thread.

$$execState\colon Thread \to ExecState$$

Every thread has a *joinSet* that comprises the threads that are joined to it and are waiting for its termination, a *wakeupTime* that stores the time when the thread expires, a *monObj* with the monitor a thread is waiting for or wants to acquire, a *joinedThread* in case the thread is joined, and several flags indicating whether an abort has been requested or initiated, whether an interrupt has been requested, or whether a suspend has been requested.

$$\begin{array}{ll} joinSet\colon Thread \to Powerset(Thread) & abortRequested\colon Thread \to Bool \\ wakeupTime\colon Thread \to \mathbb{N} \cup \{\infty\} & abortInitiated\colon Thread \to Bool \\ monObj\colon Thread \to Monitor & interruptRequested\colon Thread \to Bool \\ joinedThread\colon Thread \to Thread & suspendRequested\colon Thread \to Bool \end{array}$$

When an object *ref* of type `Thread` is created, the dynamic functions are initialized as follows:

$$\begin{array}{ll} joinSet(ref) := \emptyset & abortRequested(ref) := False \\ wakeupTime(ref) := Undef & abortInitiated(ref) := False \\ monObj(ref) := Undef & interruptRequested(ref) := False \\ joinedThread(ref) := Undef & suspendRequested(ref) := False \\ execState(ref) := Unstarted & \end{array}$$

The local state of a thread comprises a frame stack of activation records, the currently executed method, the current position in the method body (program counter), the local environment and the already computed values of expressions (operand stack).

$$\begin{array}{ll} frames\colon Thread \to List(Frame) & locals\colon Thread \to (Loc \to Adr) \\ meth\colon Thread \to Meth & values\colon Thread \to (Pos \to Result) \\ pos\colon Thread \to Pos & \end{array}$$

The current thread is denoted by '**self**' in the ASM rules below.

Fig. 3 shows a classification of the execution states of a thread and relates them to the items of the `ThreadState` enumeration in Fig. 1. A thread is *Running* if it is not *Unstarted* and not already *Dead*. A thread is considered to be *Passive* (or *WaitSleepJoin*) if it is *Running* but neither *Active* nor *Suspended*.

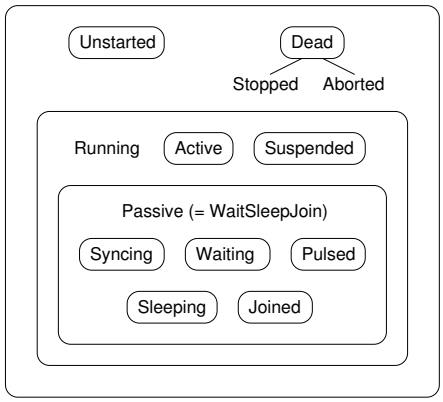

Fig. 3. The execution states of a thread.

$$Running(thread) \iff execState(thread) \notin \{Unstarted, Dead\}$$
$$Passive(thread) \iff WaitSleepJoin(thread) \iff execState(thread) \in \{Syncing, Waiting, Pulsed, Sleeping, Joined\}$$

The items `Stopped` and `Aborted` of the `ThreadState` enumeration can be obtained as follows:

$$Stopped(thread) \iff execState(thread) = Dead \wedge \neg abortRequested(thread)$$
$$Aborted(thread) \iff execState(thread) = Dead \wedge abortRequested(thread)$$

The reason for this separation is not known to us.

3.2 The methods of the `Thread` class

Fig. 4 and 5 contain diagrams for the execution states of a thread. Methods that are invoked by another thread on the current thread are displayed in grey boxes, whereas methods invoked by the current thread itself are put into white boxes. If there is no outgoing arrow for a thread method from an execution state, then this can mean either that such an invocation is not possible, e.g. since a static method can only be invoked by an active thread, or that the invocation is not allowed and throws a `ThreadStateException`.

When a thread is created by invoking the constructor of the `Thread` class its execution state is *Unstarted*. The new thread is later started by invoking the `Thread.Start` method. A thread can only be started once, otherwise a `ThreadStateException` exception is thrown. If there has already been an abort requested for the thread, its execution state is immediately changed to *Dead*. Otherwise, the execution state of the thread is updated to *Active* and its local state is initiated. The new thread now runs concurrently with the thread that invoked the `Thread.Start` method. The YieldUp(*Norm*) means that the `Thread.Start` methods returns without blocking.

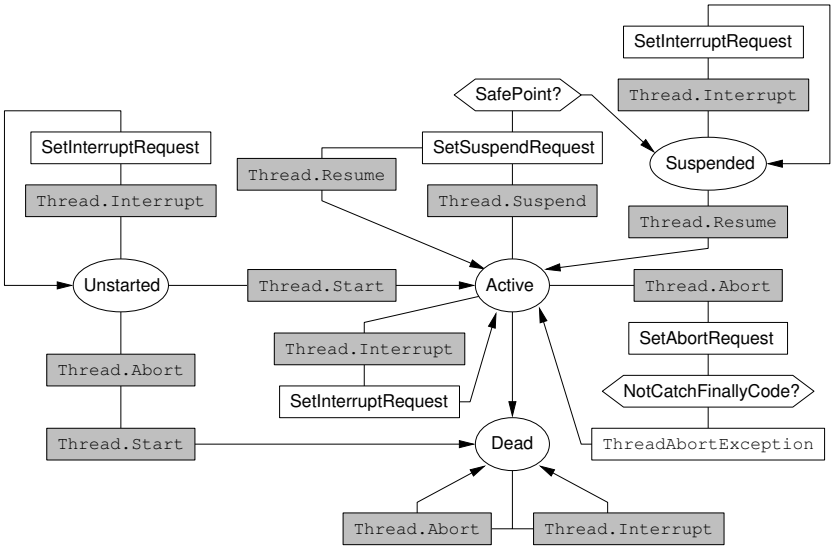

Fig. 4. Methods invoked by other threads on the current thread.

ThreadStart(*thread*) ≡
  if *execState*(*thread*) ≠ *Unstarted* **then** FailUp(`ThreadStateException`)
  else
    if *abortRequested*(*thread*) **then** *execState*(*thread*) := *Dead*
    else {ThreadInit(*thread*), *execState*(*thread*) := *Active*}
    YieldUp(*Norm*)

When a thread is created, a delegate of type `ThreadStart` has to be provided to the constructor of the `Thread` class. This delegate is later invoked, when the thread is started. Technically, this means the new thread executes the `Invoke` method of the `ThreadStart` delegate. If the invocation list of that delegate consists of a single method, then this method is executed. Otherwise, the methods of the invocation list are executed sequentially.

ThreadInit(*thread*) ≡
  let *d* = *getField*(*thread*, `Thread::delegate`)
  let *m* = `ThreadStart::Invoke()` **in**
    thread.*frames* := [ ]
    thread.*meth* := *m*
    thread.*pos* := *body*(*m*)
    thread.*values* := ∅
    thread.InitLocals(*m*, [*d*]) // `this` is the delegate *d*

The `Thread.Join` methods puts the current thread into the join set of the other thread and changes the execution state of the current thread from *Active* to *Joined*. Like every thread method that takes a timeout argument it checks first

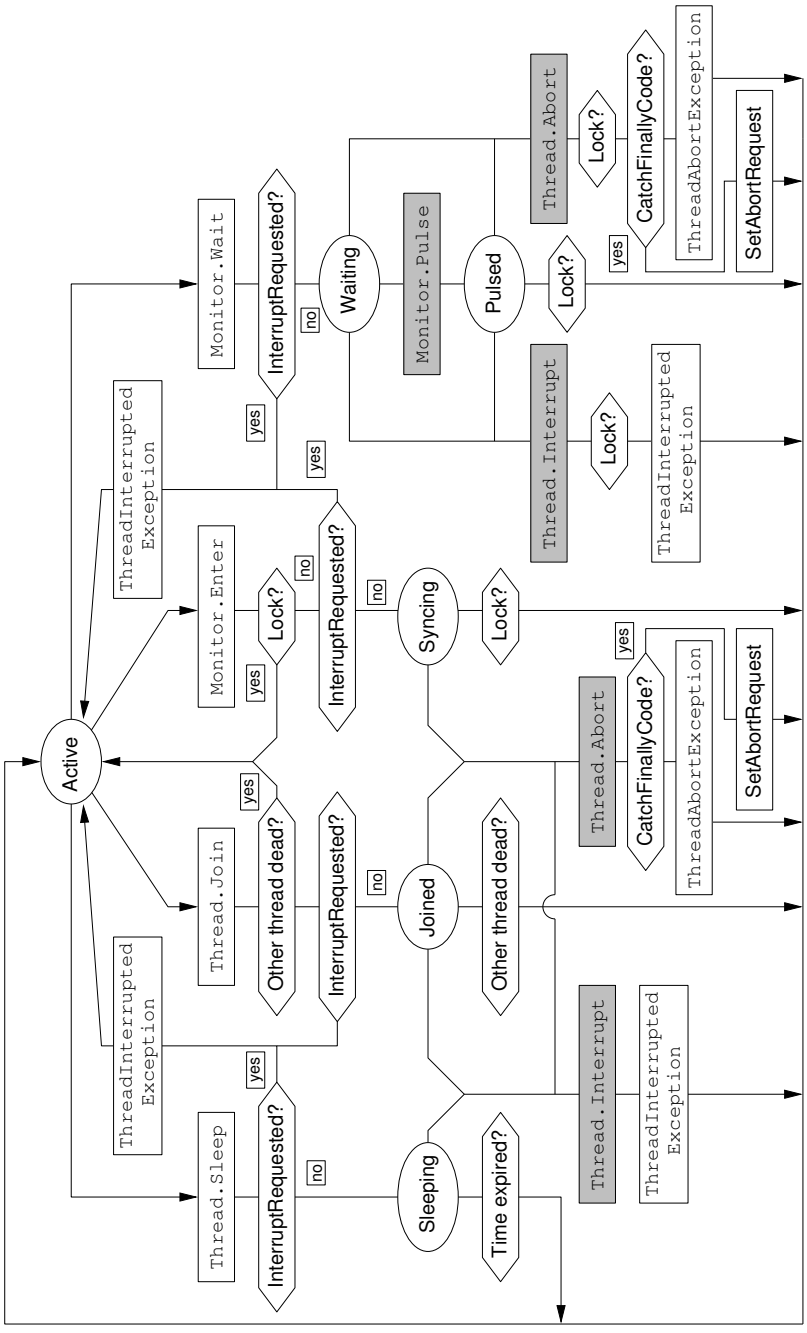

Fig. 5. Methods invoked by the current thread.

whether the argument is in the correct range. If an interrupt has been requested, then it throws a `ThreadInterruptedException` instead of joining (Mono 0.26 ignores the interrupt request [12]).

ThreadJoin(*thread*, *msec*) ≡
  if *msec* < −1 **then** FailUp(`ArgumentOutOfRangeException`)
  elseif *execState*(*thread*) = *Unstarted* **then**
    FailUp(`ThreadStateException`)
  elseif *execState*(*thread*) = *Dead* **then** YieldUp(*True*)
  elseif *interruptRequested*(**self**) **then** ThrowInterruptedException
  else
    SetWakeupTime(*msec*)
    joinSet(*thread*) := *joinSet*(*thread*) ∪ {**self**}
    joinedThread(**self**) := *thread*
    execState(**self**) := *Joined*

The thread will become active again, when the other thread has terminated or *msec* milliseconds have passed. An argument of −1 milliseconds means an infinite amount of time.

SetWakeupTime(*msec*) ≡
  if *msec* = −1 **then** *wakeupTime*(**self**) := ∞
  else *wakeupTime*(**self**) := *currentTime* + *msec*

When an `ThreadInterruptedException` is thrown, the interrupt request of the current thread is cleared.

ThrowInterruptedException ≡
  FailUp(`ThreadInterruptedException`)
  interruptRequested(**self**) := *False*

When the `Thread.Join` method returns, it indicates with a boolean result, whether the other thread is dead. If the other thread is not dead, then it follows from the definition of the predicate *Expired* and the Wakeup rule in Sect. 3.4 that the amount of time has expired.

ThreadJoinReturn ≡
  if *execState*(*joinedThread*(**self**)) = *Dead* **then** YieldUp(*True*)
  else YieldUp(*False*)

The `Thread.Sleep` method puts the current thread to sleep for the specified amount of milliseconds. The execution state of the current thread is changed from *Active* to *Sleeping*. If an interrupt has been requested, the current thread throws a `ThreadInterruptedException` instead of going to sleep.

ThreadSleep(*msec*) ≡
  if *msec* < −1 **then** FailUp(`ArgumentOutOfRangeException`)
  elseif *interruptRequested*(**self**) **then** ThrowInterruptedException
  else
    execState(**self**) := *Sleeping*
    SetWakeupTime(*msec*)
    YieldUp(*Norm*)

In order to abort another thread with the `Thread.Abort` method, the current threads needs the appropriate security permission. If the other thread is suspended, a `ThreadStateException` is thrown, although the documentation [11] says that in that case the other thread is resumed by the system. The `Thread.Abort` sets an abort request for the other thread. The effect of this request is that a `ThreadAbortException` is thrown asynchronously in the other thread (see Sect. 3.4).

```
ThreadAbort(thread) ≡
  if ¬SecurityPermission(self, ControlThread) then
    FailUp(SecurityException)
  elseif execState(thread) = Suspended then
    FailUp(ThreadStateException)
  else
    if ¬abortRequested(thread) ∧ ¬abortInitiated(thread) then
      abortRequested(thread) := True
    YieldUp(Norm)
```

The static method `Thread.ResetAbort` can be invoked by the current thread to cancel the automatic re-throwing of a `ThreadAbortException` at the end of catch blocks. It clears the flag that indicates that the abort has been initiated. Only threads that have the appropriate security permission can cancel an abort. The documentation [11] says that the method throws a `ThreadStateException` if the method was not invoked on the current thread. This can never happen, since it is a static method.

```
ThreadResetAbort ≡
  if ¬SecurityPermission(self, ControlThread) then
    FailUp(SecurityException)
  elseif ¬abortInitiated(self) then FailUp(ThreadStateException)
  else {abortInitiated(self) := False, YieldUp(Norm)}
```

The `Thread.Interrupt` method sets an interrupt request for another thread. The effect of the request is that a `ThreadInterruptedException` is injected into the other thread, if it is in a passive state (see Sect. 3.4). Otherwise, the exception is thrown by the other thread, when it changes its execution state from running into a passive state. If the other thread stays active forever, the interrupt request is ignored.

```
ThreadInterrupt(thread) ≡
  if ¬SecurityPermission(self, ControlThread) then
    FailUp(SecurityException)
  else
    if ¬interruptRequested(thread) then
      interruptRequested(thread) := True
    YieldUp(Norm)
```

The `Thread.Suspend` method sets a suspend request for another thread, if it is running. The request will asynchronously be processed by the run-time system (see Sect. 3.4).

ThreadSuspend(*thread*) ≡
  if ¬SecurityPermission(**self**, `ControlThread`) **then**
    FailUp(`SecurityException`)
  elseif *execState*(*thread*) ∈ {*Unstarted*, *Dead*} **then**
    FailUp(`ThreadStateException`)
  else
    if ¬*suspendRequested*(*thread*) ∧ *execState*(*thread*) ≠ *Suspended*
    then *suspendRequested*(*thread*) := *True*
    YieldUp(*Norm*)

A thread can be resumed by invoking `Thread.Resume` only if it is suspended or a suspend request is pending. If the thread is suspended, then its execution state is changed back to active such that it can be scheduled for execution by the run-time system again. If a suspend has been requested, the request is cleared.

ThreadResume(*thread*) ≡
  if ¬SecurityPermission(**self**, `ControlThread`) **then**
    FailUp(`SecurityException`)
  elseif *execState*(*thread*) ≠ *Suspended* ∧ ¬*suspendRequested*(*thread*)
  then FailUp(`ThreadStateException`)
  else
    if *execState*(*thread*) = *Suspended* **then** *execState*(*thread*) := *Active*
    if *suspendRequested*(*thread*) **then** *suspendRequested*(*thread*) := *False*
    YieldUp(*Norm*)

If a ThreadSuspend(t) is executed in parallel with a ThreadResume(t) on a thread t that already has a suspend request, the resume has priority over the suspend.[1]

3.3 The methods of the `Monitor` class

The methods of the `Monitor` class are static and are used to acquire and release locks of monitors. If they are invoked with the `null` reference, an exception is thrown. The `Monitor.Enter` method is used to acquire the lock of a monitor. If the current thread already owns the lock, the *lockCount* is increased. If the lock is free and the *readyQueue* of the monitor is empty, the thread immediately gets the lock. Otherwise, the thread changes its state from *Active* to *Syncing* and the thread is added to the *readyQueue* of the monitor. In case of a pending interrupt, a `ThreadInterruptedException` is thrown.

MonitorEnter(*mon*) ≡
  if *mon* = *Null* **then** FailUp(`ArgumentNullException`)
  elseif *lockOwner*(*mon*) = **self then**
    lockCount(*mon*, **self**) := *lockCount*(*mon*, **self**) + 1
    YieldUp(*Norm*)
  elseif *lockOwner*(*mon*) = *None* ∧ *Empty*(*readyQueue*(*mon*)) **then**

[1] One could as well forbid the parallel execution by a run constraint (see Sect. 4.1).

Lock(**self**, *mon*)
lockCount(*mon*, **self**) := 1
YieldUp(*Norm*)
elseif *interruptRequested*(**self**) **then** ThrowInterruptedException
else
readyQueue(*mon*) := *readyQueue*(*mon*) · [**self**]
monObj(**self**) := *mon*
execState(**self**) := *Syncing*
YieldUp(*Norm*)

To lock a monitor means to update the *lockOwner* of the monitor.

Lock(*thread*, *mon*) ≡ *lockOwner*(*mon*) := *thread*

The `Monitor.Exit` method decrements the lock count by one. If the lock count becomes 0, the lock is released.

MonitorExit(*mon*) ≡
if *mon* = *Null* **then** FailUp(`ArgumentNullException`)
elseif *lockOwner*(*mon*) ≠ **self then**
FailUp(`SynchronizationLockException`)
else
if *lockCount*(*mon*, **self**) = 1 **then** Unlock(**self**, *mon*)
lockCount(*mon*, **self**) := *lockCount*(*mon*, **self**) − 1
YieldUp(*Norm*)

To release a lock means to update the *lockOwner* of the monitor to *None*.

Unlock(*thread*, *mon*) ≡ *lockOwner*(*mon*) := *None*

The documentation [11] says that a thread can only exit a monitor if it owns the lock. The following code, however, runs in the .NET framework 1.1 as well as in Rotor [16] without throwing a `SynchronizationLockException`.

```
Object o = new object();
Monitor.Enter(o);
Monitor.Exit(o);
Monitor.Exit(o);  // Bug. Thread does not own lock.
```

The `Monitor.Wait` method appends the current thread to the *waitQueue* of the monitor and temporarily releases the lock of the monitor. The execution state of the current thread is changed from *Active* to *Waiting*.

MonitorWait(*mon*, *msec*) ≡
if *mon* = *Null* **then** FailUp(`ArgumentNullException`)
elseif *msec* < −1 **then** FailUp(`ArgumentOutOfRangeException`)
elseif *lockOwner*(*mon*) ≠ **self then**
FailUp(`SynchronizationLockException`)
elseif *interruptRequested*(**self**) **then** ThrowInterruptedException
else
SetWakeupTime(*msec*)
waitQueue(*mon*) := *waitQueue*(*mon*) · [**self**]

UNLOCK(**self**, *mon*)
monObj(**self**) := *mon*
execState(**self**) := *Waiting*

The thread remains in the *waitQueue* of the monitor until the monitor is pulsed or *msec* milliseconds have passed. At the return, the method indicates with a boolean result whether the time has expired.

MONITORWAITRETURN ≡
if *wakeupTime*(**self**) < *currentTime* **then** YIELDUP(*True*)
else YIELDUP(*False*)

The `Monitor.Pulse` method moves the first element of the *waitQueue* of the monitor to the *readyQueue*. If the *waitQueue* is empty, the method just returns. Note, that we allow also that a thread with an abort or an interrupt request can be pulsed (this point is discussed also in [10]).

MONITORPULSE(*mon*) ≡
if *mon* = *Null* **then** FAILUP(`ArgumentNullException`)
elseif *lockOwner*(*mon*) ≠ **self then**
FAILUP(`SynchronizationLockException`)
else
if ¬*Empty*(*waitQueue*(*mon*)) **then**
MOVETOREADYQUEUE(*first*(*waitQueue*(*mon*)), *mon*)
YIELDUP(*Norm*)

When a thread is moved from the *waitQueue* to the *readyQueue*, its execution state is changed from *Waiting* to *Pulsed*.

MOVETOREADYQUEUE(*thread*, *mon*) ≡
readyQueue(*mon*) := *readyQueue*(*mon*) · [*thread*]
waitQueue(*mon*) := *delete*(*thread*, *waitQueue*(*mon*))
execState(*thread*) := *Pulsed*

The `Monitor.PulseAll` methods moves all waiting threads into the *readyQueue* of the monitor.

MONITORPULSEALL(*mon*) ≡
if *mon* = *Null* **then** FAILUP(`ArgumentNullException`)
elseif *lockOwner*(*mon*) ≠ **self then**
FAILUP(`SynchronizationLockException`)
else
forall *thread* ∈ *waitQueue*(*mon*) **do** *execState*(*thread*) := *Pulsed*
readyQueue(*mon*) := *readyQueue*(*mon*) · *waitQueue*(*mon*)
waitQueue(*mon*) := [ ]
YIELDUP(*Norm*)

In Java, the wait and the ready queues are not FIFO queues but unordered sets. The `Object.notify` method of Java chooses an arbitrary element from the wait set of an object and it is not guaranteed that every thread in the wait set is ever chosen. The proposal for the new Java memory model [10] even allows so-called *spurious wake-ups*. This means that the system is allowed to remove a

thread from the wait set of an object without any reason. Note, that the POSIX thread function `pthread_cond_signal()` is also allowed to wake up more than one thread.

3.4 Scheduling of threads, timing, locking and asynchronous exceptions

Although priorities can be assigned to threads in C#, it is not guaranteed that they are honored by the scheduling algorithm. The main ASM rule for sequential C# therefore chooses repeatedly one of the possible threads and executes one (small) step in the computation of the thread. In this way the computation steps of the currently running threads are interleaved.

ExecSequentialCsharp ≡
  choose *thread* ∈ *Thread* **do** ExecStep(*thread*)

The next computation step of a thread depends on its current execution state.

ExecStep(*thread*) ≡
  if *execState*(*thread*) = *Active* **then** ExecActive(*thread*)
  elseif *Expired*(*thread*) **then** Wakeup(*thread*)
  elseif *CanAcquireLock*(*thread*) **then** AcquireLock(*thread*)
  elseif *Passive*(*thread*) ∧ *HasRequest*(*thread*) **then**
    AbortOrInterruptPassive(*thread*)

A thread is expired, if its wakeup time has been passed. The system time in milliseconds is given by the monitored function *currentTime*.

Expired(*thread*) ⟺ *execState*(*thread*) ∈ {*Waiting*, *Sleeping*, *Joined*} ∧
  wakeupTime(*thread*) ≤ *currentTime*

A thread can acquire the lock, if it can acquire the lock of its current monitor object *monObj* that was set in MonitorEnter or MonitorWait.

CanAcquireLock(*thread*) ⟺ *CanAcquireLock*(*thread*, *monObj*(*thread*))

The lock of a monitor can be acquired, if the lock is free and the thread is the first thread in the *readyQueue* of the monitor.

CanAcquireLock(*thread*, *mon*) ⟺
  execState(*thread*) ∈ {*Syncing*, *Pulsed*} ∧ *lockOwner*(*mon*) = *None* ∧
  thread = *first*(*readyQueue*(*mon*))

A thread has a request, if it has an abort or an interrupt request.

HasRequest(*t*) ⟺ *abortRequested*(*t*) ∨ *interruptRequested*(*t*)

When the execution state of the chosen thread is *Active*, the next step in the computation of the thread is executed by the rule ExecCsharp(*thread*). In case of a pending abort the system waits until the thread has left any finally block or catch clause before it aborts the threads. In case of a pending suspend, the system waits until the thread reaches a so-called safe point before suspending the thread. Safe points are points that are also safe for garbage collection.

ExecActive(*thread*) ≡
  if *abortRequested*(*thread*) ∧ ¬*CatchFinallyCode*(*thread*) **then**
    Abort(*thread*)
  elseif *suspendRequested*(*thread*) ∧ *SafePoint*(*thread*) **then**
    Suspend(*thread*)
  else ExecCsharp(*thread*)

A thread aborts by throwing a `ThreadAbortException`. The fact that the thread is responding to the abort request is recorded in the *abortInitiated* flag.

Abort(*thread*) ≡
  Fail(*thread*, `ThreadAbortException`)
  ClearAbortRequest(*thread*)
  abortInitiated(*thread*) := *True*

When the abort request is cleared, any pending interrupt request is also cleared.

ClearAbortRequest(*thread*) ≡
  abortRequested(*thread*) := *False*
  if *interruptRequested*(*thread*) **then**
    interruptRequested(*thread*) := *False*

Like any other exception, a `ThreadAbortException` is propagated upwards in the frame stack of the thread. If it crosses a try block with catch clauses and a possible finally block, the catch clauses are searched for a matching handler. If there exists one, the corresponding catch block is executed. The finally block is executed afterwards. At the end of the catch block, however, the `ThreadAbortException` is re-thrown by the system. More precisely, if an abort has been initiated and a catch block terminates but not with an exception, then a new `ThreadAbortException` is thrown at the end of the catch block. If the catch block terminates abruptly with an exception, that exception is propagated upwards. Hence, the rules for exception handling of C#$_{\mathcal{E}}$ of [2] have to be refined. If the current position of the thread (indicated by the black triangle) is at the end of a catch block with result *res* and *res* is not an exception, a `ThreadAbortException` is thrown. Hence a `ThreadAbortException` cannot be swallowed unless the `Thread.ResetAbort` method is called (see Sect. 3.2).

ExecCsharpStm$_T$ ≡ **match** *context*(*pos*)
  `try` *Exc*(*ref*) ... `catch(...)` ▶*res* ... →
    excStack := *pop*(*excStack*)
    if *abortInitiated*(**self**) ∧ *res* ∉ *Exc* **then**
      FailUp(`ThreadAbortException`)
    else YieldUp(*res*)

When a suspend request is pending and the thread has reached a safe point, the system changes its execution state from *Active* to *Suspended* and clears the request.

Suspend(*thread*) ≡
  execState(*thread*) := *Suspended*
  suspendRequested(*thread*) := *False*

If a sleeping, joined or waiting thread has expired, the system has to wakeup the thread. If the thread is *Sleeping*, its execution state is changed to *Active*. If the thread is *Joined*, it is removed from the *joinSet* and returns from the `Thread.Join` method (see Sect. 3.2). If the thread is *Waiting* it is moved to the *readyQueue* of the monitor and has to re-acquire the lock (its state is changed to *Pulsed*).

WAKEUP(*thread*) ≡
  if *execState*(*thread*) = *Sleeping* **then** *execState*(*thread*) := *Active*
  if *execState*(*thread*) = *Joined* **then let** t = *joinedThread*(*thread*) **in**
    joinSet(t) := *joinSet*(t) \ {*thread*}
    execState(*thread*) := *Active*
  if *execState*(*thread*) = *Waiting* **then**
    MOVETOREADYQUEUE(*thread*, *monObj*(*thread*))

If a thread can acquire the lock, it becomes the owner of the lock. Its execution state is changed from *Pulsed* or *Syncing* to *Active*. If it acquires the lock for the first time, the *lockCount* is initialized to 1. Otherwise, when the thread has temporarily released the lock by invoking the `Monitor.Wait` method, the old lock count is still valid.

ACQUIRELOCK(*thread*) ≡
  let *mon* = *monObj*(*thread*) **in**
    LOCK(*thread*, *mon*)
    if *execState*(*thread*) ≠ *Pulsed* **then** *lockCount*(*mon*, *thread*) := 1
    readyQueue(*mon*) := *tail*(*readyQueue*(*mon*))
    execState(*thread*) := *Active*

If a passive thread has an abort or an interrupt request, an exception is injected into the thread. If both, an abort request and an interrupt request are pending, the abort request has priority (unless the thread executes catch or finally code).

ABORTORINTERRUPTPASSIVE(*thread*) ≡
  if *abortRequested*(*thread*) ∧ ¬*CatchFinallyCode*(*thread*) **then**
    INJECTEXCEPTION(*thread*, `ThreadAbortException`)
    CLEARABORTREQUEST(*thread*)
    abortInitiated(*thread*) := *True*
  elseif *interruptRequested*(*thread*) **then**
    INJECTEXCEPTION(*thread*, `ThreadInterruptedException`)
    interruptRequested(*thread*) := *False*

When a `ThreadAbortException` or a `ThreadInterruptedException` is injected into a passive thread, the execution state of the thread is changed to *Active* in order that the exception can propagate upwards and probably terminate the thread. If the thread is waiting, it is moved to the *readyQueue* and has to re-acquire the lock, since in this case the thread is still in a critical section of code and possible exception handlers and finally block should only be executed under the exclusive control of the monitor.

INJECTEXCEPTION(*thread*, *exception*) ≡
  FAIL(*thread*, *exception*)

if *execState*(*thread*) ∈ {*Syncing*, *Sleeping*, *Joined*} **then**
execState(*thread*) := *Active*
if *execState*(*thread*) = *Waiting* **then**
MoveToReadyQueue(*thread*, *monObj*(*thread*))

A thread terminates when the frame stack of the thread is empty again and the `Invoke` method of the delegate of the thread terminates. The method can terminate normally or abruptly with an exception. In any case, the execution state of the thread is updated from *Active* to *Dead* and the threads that are joined are notified by changing their states from *Joined* to *Active* such that they can return from the `Thread.Join` method. If the thread terminates with an exception, the exception may or may not be reported as unhandled exception to the console.

ExecCsharpStm$_T$ ≡ **match** *context*(*pos*)
res → **if** *pos* = *body*(*meth*) ∧ *Empty*(*frames*) **then**
forall *thread* ∈ *joinSet*(**self**) **do** *execState*(*thread*) := *Active*
joinSet(**self**) := ∅
execState(**self**) := *Dead*
if *exception*(*res*) **then** ReportUnhandledExc(*res*)

When a thread is *Dead*, it remains in this execution state and cannot be re-activated (see also Fig. 4).

4 A parallel model for C# threads

On a multiprocessor system different threads can execute code concurrently on different processors. We model this case using a special kind of distributed ASMs that are executable in tools like AsmL [6]. The main rule for the parallel thread model chooses repeatedly an arbitrary *set* of possible threads and executes *in parallel* the next computation step for each of the chosen threads.

ExecParallelCsharp ≡
choose *T* ⊆ *Thread* **do**
forall *thread* ∈ *T* **do** ExecStep(*thread*)

The ExecStep rule is the same as in the sequential model. However, since the computation steps are executed in parallel, conflicts can occur if the same location is updated by different threads to different values. Updates of the local state of a thread (frame stack, program counter, local environment, operand stack) are not critical, since the local state is parameterized by the thread (see Sect. 3.1). Updates to the shared memory are discussed in Sect. 5 below. An analysis of the transition rules shows that the following conflicts have to be avoided by imposing run constraints on ExecParallelCsharp.

4.1 Run constraints for the parallel thread model

1. The rule AcquireLock(*thread*) is not allowed to run in parallel with the rule MonitorEnter(*mon*), if *monObj*(*thread*) = *mon*. Otherwise, there would be a conflict for *lockOwner*(*mon*).

2. It is not allowed that two different threads execute MonitorEnter(mon) in parallel. Otherwise, conflicts for *lockOwner*(mon) and *readyQueue*(mon) could occur.
3. The rules AcquireLock($thread$) and MoveToReadyQueue(t, mon) are not allowed to run in parallel, if *monObj*($thread$) = mon. Otherwise, there is a conflict for *readyQueue*(mon).
4. The rule MonitorPulse(mon) or MonitorPulseAll(mon) is not allowed to run in parallel with the rule MoveToReadyQueue(t, mon). Otherwise, there would be a conflict for *readyQueue*(mon).
5. MoveToReadyQueue(t_1, mon) and MoveToReadyQueue(t_2, mon) are not allowed to run in parallel for $t_1 \neq t_2$. Otherwise, there would be a conflict for *readyQueue*(mon).
6. The rules Suspend($thread$) and ThreadResume($thread$) are not allowed to run in parallel. Otherwise, the ThreadResume($thread$) would be ignored by the system.

Note, that MoveToReadyQueue(t, mon) is used in the rules Wakeup(t) and AbortOrInterruptPassive(t) in case that the execution state of t is *Waiting*.

5 The .NET memory model

The .NET memory model is outlined in [5, Partition 1, §11.6.5, §11.6.7]. According to [13,15], it gives the following (weak) guarantees about the ordering of memory reads and writes:

- Reads and writes from the same thread to a location cannot be re-ordered.
- No read can move before a lock acquire (or volatile read).
- No write can move after a lock release (or volatile write).
- Writes cannot cross a `Thread.WriteMemoryBarrier()`.
- Neither reads nor writes can cross a `Thread.MemoryBarrier()`.

To application programmers the memory model is often (wrongly) explained in a stronger form. Each thread has its local cache. After acquiring the lock the thread's cache is invalidated, so that reads afterward are done from the main memory. After releasing the lock the thread's cache is flushed to main memory. Note that in .NET a read or write of a volatile location affects also the ordering of reads and writes of other locations.

For the ASM specification of the .NET memory model we follow the ASM specification of the *Local Consistency Memory Model* for Java in [1] and use a universe of events that is divided into disjoint subuniverses as follows:

$$\begin{aligned} \mathit{Event} ::= {} & \mathit{WriteEvent} \mid \mathit{LockEvent} \mid \mathit{UnlockEvent} \mid \mathit{ReadVolatileEvent} \\ & \mid \mathit{BarrierEvent} \mid \mathit{WriteBarrierEvent} \end{aligned}$$

Events are ordered during a run of a multi-threaded C# program by a dynamic predicate $\prec$. We denote by $\prec^+$ the transitive closure and by $\prec^*$ the reflexive, transitive closure of $\prec$. Each *WriteEvent* has two attributes, an address and a

```
class Foo {
  private Helper helper;
  public Helper GetHelper() {
    if (helper == null)          // quick check
      lock (this)
        if (helper == null) {    // double check
          Helper o = new Helper();
          Thread.MemoryBarrier();
          helper = o;
        }
    return helper;
  }
}
```

Fig. 6. The double checked locking pattern.

value, adr: *WriteEvent* $\rightarrow$ *Address* val: *WriteEvent* $\rightarrow$ *Value*. The latest event of a thread is recorded in $latest$: *Thread* $\rightarrow$ *Event*.

The memory model is now reduced to the question: "Which write event(s) can be seen by a memory read?" When a thread writes a value to an address, a new *WriteEvent* is created.

$\textsc{Write}(adr, val) \equiv$ **let** $e = new(\mathit{WriteEvent})$ **in**
$\{adr(e) := adr,\ val(e) := val\}$
$\textsc{InsertAfterLatest}(\mathbf{self}, e)$

The new *WriteEvent* is inserted in the event order immediately after the latest event of the current thread.

$\textsc{InsertAfterLatest}(thread, e) \equiv$
if $latest(thread) \neq \mathit{Undef}$ **then** $latest(thread) \prec e := \mathit{True}$
$latest(thread) := e$

Reading a value from an address means choosing an appropriate *WriteEvent* for that address and returning the value that has been written to that address.

$\textsc{Read}(adr) \equiv$
choose $e \in \mathit{WriteEvent}$ **with** $adr(e) = adr \wedge \neg \mathit{Overwritten}(\mathbf{self}, e)$ **do**
return $val(e)$

A read cannot see arbitrary write events but only those that are not overwritten with respect to the latest event of the current thread or any memory barrier.

$\mathit{Overwritten}(t, e) \iff$
$\exists w \in \mathit{WriteEvent}\,(adr(w) = adr(e) \wedge e \prec^+ w \wedge \mathit{Previous}(t, w))$

$\mathit{Previous}(t, w) \iff w \prec^* latest(t) \vee \exists b \in \mathit{BarrierEvent}\,(w \prec^* b)$

When a monitor is locked a new *LockEvent* is created and inserted in the event order after the latest *UnlockEvent* of the monitor as well as after the latest event of the current thread (in this way the write events of the last thread that owned the lock are synchronized with the current thread).

$\textsc{Lock}(thread, mon) \equiv$ **let** $e = new(\mathit{LockEvent})$ **in**
  if $\mathit{latestUnlock}(mon) \neq \mathit{Undef}$ **then** $\mathit{latestUnlock}(mon) \prec e := \mathit{True}$
  $\textsc{InsertAfterLatest}(thread, e)$
  forall $b \in \mathit{WriteBarrierEvent}$ **do** $b \prec e := \mathit{True}$
  $\mathit{lockOwner}(mon) := thread$

The function $\mathit{latestUnlock}\colon \mathit{Monitor} \to \mathit{UnlockEvent}$ records the latest unlock event of a monitor. The *UnlockEvent* created at the monitor exit prevents overwritten write events from being seen by the next thread that aquires the lock of the monitor.

$\textsc{Unlock}(thread, mon) \equiv$ **let** $e = new(\mathit{UnlockEvent})$ **in**
  $\mathit{latestUnlock}(mon) := e$
  $\textsc{InsertAfterLatest}(thread, e)$
  $\mathit{lockOwner}(mon) := \mathit{None}$

The function `Thread.MemoryBarrier` creates a new *BarrierEvent* which is inserted in the event order after the latest event of the current thread. (Barrier events are used in the definition of the *Overwritten* predicate above.)

$\textsc{MemoryBarrier} \equiv$ **let** $e = new(\mathit{BarrierEvent})$ **in**
  $\textsc{InsertAfterLatest}(\textbf{self}, e)$

The function `Thread.WriteMemoryBarrier` creates a *WriteBarrierEvent* which are inserted in the event order before any future lock event.

$\textsc{WriteMemoryBarrier} \equiv$ **let** $e = new(\mathit{WriteBarrierEvent})$ **in**
  $\textsc{InsertAfterLatest}(\textbf{self}, e)$

A read of a volatile field creates a new *ReadVolatileEvent*. The choosen write event (which was a volatile write) is inserted in the event ordering before the read event.

$\textsc{ReadVolatile}(adr) \equiv$
  let $r = new(\mathit{ReadVolatileEvent})$ **in**
    $\textsc{InsertAfterLatest}(\textbf{self}, r)$
    forall $b \in \mathit{WriteBarrierEvent}$ **do** $b \prec r := \mathit{True}$
    choose $e \in \mathit{WriteEvent}$ **with** $\mathit{adr}(e) = adr \wedge \neg \mathit{Overwritten}(\textbf{self}, e)$ **do**
      $e \prec r := \mathit{True}$
      return $\mathit{val}(e)$

A write to a volatile field uses the normal Write rule.

The so-called *double-checked locking pattern* in Fig. 6 uses a memory barrier to prevent another thread from seeing a non-null value of the `helper` field while the fields of the `Helper` object itself still contain their default null values which are overwritten in the constructor of the `Helper` class. (The constructor may be inlined by the JIT compiler.) Instead of using the memory barrier the `helper` field could be declared `volatile`.

```
class Account {
  private decimal balance = 0.0M;

  public void Deposit() {
    lock (this) {
      try { Monitor.Wait(this); }
      finally { balance += 100.00M; }
    }
  }

  public static void Main() {
    Account a = new Account();
    Thread t = new Thread(new ThreadStart(a.Deposit));
    t.Start();
    Thread.Sleep(100);
    lock (a) {
      Console.WriteLine(a.balance);  // Output: 0
      Monitor.Pulse(a);
      t.Interrupt();
      Thread.Sleep(100);
      Console.WriteLine(a.balance);  // Output: 100.00 (bug)
    }
  }
}
```

Fig. 7. A bug in Microsoft's .NET Framework version 1.1

It is not clear to us, whether the following example is allowed by the Ecma .NET memory model. Consider two threads that concurrently execute the following instructions, where initially `p.x == 0`, `p.y == 0`:

Thread 1	Thread 2
`r1 = p.x;`	`r2 = p.y;`
`p.y = 1;`	`p.x = 2;`

Is the result `r1 == 2` and `r2 == 1` possible? According to our specification of the memory model, it is not possible. However, if we allow the compiler to switch the assignments in both threads (under the assumption that `p.x` and `p.y` are independent variables), the result is plausible. Maybe the result is justified by the paragraph about execution order in [4, §10.10].

6 Conclusion

The ASM method forces the person who writes a specification to think in terms of an abstract implementation. This leads to questions and cases that are usually forgotten in other formal or informal approaches. Fig. 7 contains a bug in Microsoft's .NET Framework 1.1 [11] which was detected during the construction

of our thread model. The bug shows a situation where two threads execute at the same time code in two critical sections protected by the same monitor. This should not happen.

The main function in Fig. 7 creates an account, starts another thread with the `Deposit` method of the account and sleeps for 100 milliseconds. During the sleep, the deposit thread acquires the lock of the account and waits on the account in order to later deposit 100 dollars when it is pulsed. After the sleep, the main thread locks the account and executes its critical section. At the beginning, the balance is still `0`. The main thread pulses the account and moves the deposit thread from the wait queue into the ready queue of the account. Then it interrupts the deposit thread and sleeps again for 100 milliseconds (still holding the lock of the account). When it awakes, the balance has changed to `100.0M`. Why?

The change of the balance is only possible if the deposit thread executes the finally block, which is in its critical section, *without owning the lock of the account*. The same problem occurs if `Thread.Interrupt` is replaced by `Thread.Abort`. The Rotor SSCLI implementation [16] correctly prints `0` at the end of the lock statement in the main function. After that, however, it deadlocks for unknown reasons.

References

1. V. Awhad and C. Wallace. A unified formal specification and analysis of the new Java memory models. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 166–185. Springer-Verlag, 2003.
2. E. Börger, G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of C#. Technical report, University of Pisa and ETH Zürich, 2003.
3. E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
4. C# Language Specification. Standard ECMA–334, 2001. Web pages at `http://www.ecma-international.org/`.
5. Common Language Infrastructure (CLI). Standard ECMA–335, 2001. Web pages at `http://www.ecma-international.org/`.
6. Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at `http://research.microsoft.com/foundations/AsmL/`, 2001.
7. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(tm) Language Specification*. Addison Wesley, second edition, 2000.
8. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1993.
9. A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# Language Specification*. Addison-Wesley, 2003.
10. Java(TM) Memory Model and Thread Specification Revision. Web pages at `http://www.jcp.org/en/jsr/detail?id=133`.
11. Microsoft .NET Framework 1.1 Software Development Kit. Download from `http://msdn.microsoft.com/netframework/howtoget/`.

12. The Mono project. Web pages at http://www.go-mono.com/.
13. V. Morrison. The DOTNET Memory Model. dotnet@discuss.develop.com mailing list, 2002.
14. J. Richter. Safe thread synchronization. *MSDN Magazine, .NET column*, January 2003.
15. A. G. Robison. Memory consistency & .NET. *Dr. Dobb's Journal*, April 2003.
16. Rotor – Shared Source Common Language Infrastructure (SSCLI). Web pages at http://msdn.microsoft.com/net/sscli/ and http://www.sscli.net/.
17. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine—Definition, Verification, Validation*. Springer-Verlag, 2001.
18. D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly, 2003.
19. C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI 2003*, pages 115–128, June 2003.