A Memory Model Sensitive Checker for C#

Thuan Quang Huynh and Abhik Roychoudhury

Department of Computer Science, National University of Singapore
{huynhqua,abhik}@comp.nus.edu.sg

Abstract. Modern concurrent programming languages like Java and C# have a programming language level memory model; it captures the set of all allowed behaviors of programs on any implementation platform — uni- or multi-processor. Such a memory model is typically weaker than Sequential Consistency and allows reordering of operations within a program thread. Therefore, programs verified correct by assuming Sequential Consistency (that is, each thread proceeds in program order) may not behave correctly on certain platforms! The solution to this problem is to develop program checkers which are memory model sensitive. In this paper, we develop such an invariant checker for the programming language C#. Our checker identifies program states which are reached only because the C# memory model is more relaxed than Sequential Consistency. Furthermore, our checker identifies (a) operation reorderings which cause such undesirable states to be reached, and (b) simple program modifications — by inserting memory barrier operations — which prevent such undesirable reorderings.

1 Introduction

Modern mainstream programming languages like Java and C# support multi-threading as an essential feature of the language. In these languages multiple threads can access shared objects. Moreover, synchronization mechanisms exist for controlling access to shared objects by threads. If every access to a shared object by any thread requires prior acquisition of a common lock, then the program is *guaranteed* to be "properly synchronized". On the other hand, if there are two accesses to a shared object/variable v by two different threads, at least one of them is a write, and they are not ordered by synchronization — the program is then said to contain a data race, that is, the program is *improperly synchronized*. Improperly synchronized programs are common for more than one reason — (a) programmers may want to avoid synchronization overheads for low-level program fragments which are executed frequently, (b) programmers may forget to add certain synchronization operations in the program, or (c) programmers forget to maintain a common lock guarding accesses to some shared variable v since there are often many lock variables in a real-life program.

Problem Statement The work in this paper deals with formal verification (and subsequent debugging) of multi-threaded C# programs which are improperly

synchronized. As a simple example consider the following schematic program fragment, and suppose initially `x = y = 0`. Moreover `l1`, `l2` are thread-local variables while `x`, `y` are shared variables.

```
x = 1;  |  l1 = y;
y = 1;  |  l2 = x;
```

If this program is executed on a uni-processor platform, we cannot have `l1 = 1, l2 = 0` at the end of the program. However, on a multiprocessor platform which allows reordering of writes to different memory locations this is possible. On such a platform, the writes to `x`, `y` may be completed out-of-order. As a result, the following completion order is possible $\langle \texttt{y} = 1, \texttt{l1} = \texttt{y}, \texttt{l2} = \texttt{x}, \texttt{x} = 1 \rangle$.

Since an improperly synchronized program can exhibit different sets of behaviors on different platforms, how do we even specify the semantics of such programs and reason about them? Clearly, we would like to reason about programs in a *platform-independent* fashion, rather than reasoning about a program's behaviors separately for each platform. Languages like Java, C# allow such platform-independent reasoning by defining a *memory model* at the programming language level. Now, what does a memory model for a programming language like C# mean? The C# memory model (also called the .NET memory model [15]) is a set of abstract rules which capture the behaviors of multi-threaded programs on *any* implementation platform — uni-processor or multi-processor. Given a multi-threaded C# program P, the set of execution traces of P permitted under the .NET memory model is a superset of the traces obtained by interleaving the operations of program P's individual threads. The operations in any thread include read/write of shared variables and synchronization operations like lock/unlock. The .NET memory model permits *certain operations* within a thread to be completed out-of-order, that is, the programming language level memory model essentially specifies which reorderings are allowed. So, to consider all program behaviors we need to take into account — (a) arbitrary interleavings of threads, and (b) certain (not all) reorderings within a thread. This makes the formal verification of improperly synchronized multi-threaded programs especially hard.

Basic Approach In this paper, we develop a memory-model sensitive invariant checker for the programming language C#. Our checker verifies a C# program at the level of bytecodes. The checker proceeds by representing and managing states at the level of C#'s stack-based virtual machine. Moreover, the checker's state space exploration takes the .NET memory model into account. In other words, it allows the reorderings permitted by .NET memory model to explore additional reachable states in a program. Thus, the programming language level memory model is treated as a formal contract between the program and the language implementation; we then take this contract into account during software verification.

Furthermore, we note that programmers usually understand possible behaviors of a multi-threaded program by using a stronger model called *Sequential*

Consistency [10]. An execution model for multi-threaded programs is sequentially consistent if for any program P (a) any execution of P is an interleaving of the operations in the constituent threads (b) the operations in each constituent thread execute in program order. Thus, if we are model checking an invariant φ, our checker may uncover counter-example traces which (a) violate φ , and (b) are not allowed under Sequential Consistency. Disallowing such counter-example traces requires disabling reorderings among operations. This is usually done by inserting *memory barriers or fence operations*; a memory barrier is an operation such that instructions before the barrier must complete before the starting of instructions after the barriers. Since memory barriers are expensive operations (in terms of performance) we use a maxflow-mincut algorithm to insert minimal number of barriers/fences for ruling out program states which are unreachable under Sequential Consistency.

Technical Contributions Our work involves the following steps — which taken together constitute the technical contributions of this paper.

- *Memory Model Specification* We first understand and formally specify the .NET memory model. Previous works [23] have investigated this issue and discussed certain corner cases in the .NET memory model description. Unlike [23], our specification is not operational/executable, making it more accessible to system designers (who may not have formal methods background).
- *The Checker* We use the .NET memory model specification to develop a memory model sensitive invariant checker at the level of bytecodes. It allows all execution traces permitted by .NET memory model. The checker issues operations in program order but allows them to complete out-of-order as long as the reordering is permitted by .NET memory model.
- *Memory Barrier Insertion* Our checker is useful for uncovering all execution traces allowed by the .NET memory model. However, when the programmer finds "unexpected" execution traces using our checker how does (s)he disallow this behavior? We use the well-known maxflow-mincut algorithm [7] to rule out program states unreachable under Sequential Consistency. The mincut yields (a minimal number of) places in the program where the memory barriers are to be inserted.

In Section 3 we show a simple working example to explain our identification and removal of undesirable program behaviors.

2 Related Work

Programming language level memory models are relatively new. In the recent years, substantial research efforts have been invested in developing the Java Memory Model (*e.g.* see [1, 11, 13]). These works mostly focus on what should be the programming language level memory model for Java.

For the .NET memory model, a formal executable specification based on Abstract State Machines has been discussed in [23]. In this paper, we formally

present the .NET memory model in a tabular non-operational format — clearly showing which pairs of operations can be reordered. This makes the formal specification more accessible to system designers as well. Furthermore, even though our memory model specification itself is not executable (unlike [23]) we show how it can be exploited for exploring the state space of a program.

As far as program verification is concerned, typically most works on multi-threaded program verification are oblivious of the programming language memory model. For all such works, the execution model implicitly assumed is Sequential Consistency — operations in a thread proceed in program order and any interleaving among the threads is possible. Integrating programming language level memory models for reasoning about programs has hardly been studied. In particular, our previous work [21] integrated an operational specification of the Java Memory Model for software model checking. Also, the work of [24] integrates an executable memory model specification for detecting data races in multi-threaded Java programs.

Our checker verifies programs at the level of bytecodes; its state space representation has similarities with the Java Path Finder (JPF) model checker [9]. However, JPF is not sensitive to Java memory model, and it implicitly considers sequential consistency as the program execution model. In fact, works on bytecode level formal reasoning (*e.g.*, see [18] and the articles therein) typically have not considered the programming language level memory model.

The work of [12] develops a behavioral simulator to explore program behaviors allowed by the Java memory model. Apart from the differences in programming language (Java and C#) there are at least two conceptual differences between our work and [12]. First of all, their explorer works at the level of abstract operations such as read/write/lock/unlock whereas our checker operates at the lower (and more realistic) bytecode level. Secondly, and more importantly, our tool does not only explore all program executions allowed by the .NET memory model. It can also suggest which barriers are to be inserted for disallowing program executions which are not sequentially consistent but are allowed by the (more relaxed) .NET memory model. This technique is *generic* and is not restricted to C#.

Finally, an alternative to our strategy of inserting memory barriers might be to mark all shared variables in the program as *volatile* [14]. We however note this does not work due to the weak definition and implementation of volatiles in C#. In particular, C# language documents [14] and C# implementations (*e.g.,* .NET 2.0) seem to allow reordering of volatile writes occurring before volatile reads in a program thread. On the other hand, memory barriers have a clear well-understood semantics but they incur performance overheads. For this reason, given an invariant property φ we insert *minimal* memory barriers in the program text which disallow all non-sequentially consistent execution traces violating invariant φ. Note that we are inserting memory barriers to disallow execution traces (in a state transition graph) which violate a *given* invariant property. Thus, we do not seek to avoid all data races, our aim is to avoid violations of a given program invariant.

3 A Working Example

We consider Peterson's mutual exclusion algorithm [20] to illustrate our approach. The algorithm uses two `lock` variables and one shared `turn` variable to ensure mutually exclusive access to a critical section; a shared variable `counter` is incremented within the critical section. Initially, we have `lock0 = lock1 = turn = counter = 0`.

Thread 1	Thread 2
1. `lock0 = 1;`	A. `lock1 = 1;`
2. `turn = 1;`	B. `turn = 0;`
3. `while(1){`	C. `while(1) {`
4. `if (lock1!=1)||(turn==0)`	D. `if (lock0!=1)||(turn==1)`
5. `break; }`	E. `break; }`
6. `counter++;`	F. `counter++;`
7. `lock0 = 0;`	G. `lock1 = 0;`

In this program we are interested in the value of the variable `counter` when the program exits. Under sequential consistency, the algorithm is proven to allow only a single thread running in the critical section at the same time and thus when the program exits, we always have *counter == 2*. However when we run the program in a relaxed memory model (such as the .NET memory model) we can observe *counter == 1* at the end. One execution trace that can lead to such an observable value is as follows.

Thread 1	Thread 2
`write lock0 = 1` (line 1)	
`write turn = 1` (line 2)	
`read 0 from lock1, break` (line 4,5)	
`read 0 from counter` (line 6)	
	`write lock1 = 1` (line A)
	`write turn=0` (line B)

At this point, `Thread 1` can write 1 to `counter` (line 6), then write 0 to `lock0` (line 7). However if the writes to `counter` and `lock0` are reordered, `lock0 = 0` is written while `counter` still holds the old value 0. Thread 2 reads `lock0 = 0`, it will break out of its loop and load the value of `counter` which is now still 0. So both threads will write the value 1 to `counter`, leading to *counter == 1* at the end of the program.

Finding out such behaviors is a complex and error-prone task if it is done manually. Moreover even after we find them, how do we disable such behaviors? A quick way to fix the problem is to disable all reorderings within each thread; this clearly ensures Sequential Consistency. Recall that a memory barrier requires all instructions before the barrier to complete before the starting of all operations after the barrier. We can disable all reorderings allowed by a given relaxed memory model by inserting a memory barrier after each operation which can possibly be reordered. This will lead to very high performance overheads.

Note that running the above code with all shared variables being volatile also does not work. In Microsoft .NET Framework 2.0 on Intel Pentium 4, the variable `counter` is still not always observed to be 2 at the end of the program. This seems to be due to the possibility of (volatile-write → volatile-read) reorderings, an issue about which the CLI specification is also ambiguous. We discuss this matter in more details in the next section.

In this paper, we provide a solution to the problem of finding additional behaviors under a relaxed memory model and then disabling those behaviors without compromising program efficiency. Using our checker we can first explore all reachable states under Sequential Consistency and confirm that *counter == 2* is guaranteed at the end of the program. This amounts to verifying the invariant property $AG((pc == end) \Rightarrow (counter == 2))$ expressed in Computation Tree Logic (CTL). Here *pc* stands for the program counter (capturing the control locations of both the threads) and *end* stands for the last control location (where both threads have terminated). We then check the same invariant property under the .NET memory model; this check amounts to exploring more reachable states from the initial state (as compared to the set of reachable states computed under Sequential Consistency). We find that under the .NET memory model, our property can be violated since *counter == 1* is possible at the end of the program. The checker does a full reachable state space exploration and returns all the counter-example traces, that is, all possible ways of having $counter \neq 2$ at the end of the program.

However, more importantly, our checker does not stop at *detecting* possible additional (and undesirable) behaviors under the .NET memory model. After finding that the property $AG((pc == end) \Rightarrow (counter == 2))$ is violated under .NET memory model, our checker employs a memory barrier insertion heuristic to suggest an error *correction* strategy; it finds three places in each thread for inserting memory barriers. We only show the modified code for Thread1; Thread2's modification is similar.

```
lock0 = 1; MemoryBarrier; turn = 1;
while(1){
        MemoryBarrier; if((lock1 != 1) || (turn == 0)) break;
}
counter++; MemoryBarrier; lock0 = 0;
```

The inserted memory barriers are sufficient to ensure that the algorithm will work correctly under the relaxed memory model of C# (while still allowing the compiler/hardware to reorder other operations for maximum performance). This claim can again be verified using our checker — that is, by running the checker on the program with barriers under the relaxed .NET memory model we can verify that $AG((pc == end) \Rightarrow (counter == 2))$ holds. Moreover, the number of inserted barriers is also "optimal" — that is, at least so many barriers are needed to disallow all possible violations of $AG((pc == end) \Rightarrow (counter == 2))$ under the .NET memory model.

4 .NET Memory Model and its Implementation

In this section, we first describe the programming language level memory model for C#, also called the .NET memory model, based on the information in two Microsoft's official ECMA standard document [15] and [14].

We present which reorderings are allowed by .NET memory model as a reordering table. We first describe the bytecode types it considers and then present allowed bytecode reorderings. The bytecode types are:

- Volatile reads/writes: Reads/writes to volatile variables (Variables in a C# program can be marked by the programmer by the keyword "volatile" indicating that any access to such a variable should access its master copy).
- Normal reads/writes: Reads/writes to variables which have not been marked as volatile in the program.
- Lock/unlock: The synchronization operations.

Among these operations, the model allows the reorderings summarized by Table 1. The model leaves a lot of possibility for optimization as long as program dependencies within a thread are not violated (*e.g.*, `store x; load x` is never executed out-of-order due to data dependency on `x`). While data-dependency removal may allow more optimizations, the CLI documents explicitly *prohibit* doing so — see execution order rules in section 10.10 of [14]. Furthermore, here we are presenting the memory model in terms of *allowed bytecode reorderings* and not in terms of reorderings of abstract program actions. Optimizations which remove dependencies are usually performed by the compiler (the hardware platforms respect program dependencies) and hence would already be reflected in the bytecode.

Reorder	2nd bytecode					
1st bytecode	Read	Write	Volatile Read	Volatile Write	Lock	Unlock
Read	Yes	Yes	Yes	No	Yes	No
Write	Yes	Yes	Yes	No	Yes	No
Volatile-Read	No	No	No	No	No	No
Volatile-Write	Yes	Yes	*Yes*	No	Yes	No
Lock	No	No	No	No	No	No
Unlock	Yes	Yes	Yes	No	No	No

Table 1. Bytecode reordering allowed by the .NET memory model

Our reordering table is constructed based on the following considerations.

- Normal Reads and Writes are freely reordered.
- Locks and Unlocks are never reordered.
- Volatile Reads and writes have acquire-release semantics, that is, operations after (before) volatile-read (volatile-write) cannot be moved to before (after) the volatile-read (volatile-write).

An interesting case is when a volatile write is followed by a volatile read (to a different variable). If we adhere to a strict ordering of all volatile operations, this reordering is disallowed.[1] But it seems that Microsoft's .NET 2.0 allows this reordering on Peterson's mutual exclusion example shown in Section 3 e.g., the reads in Line 4 (or Line D) can get reordered w.r.t. writes in Lines 1,2 (Lines A, B) thereby leading to violation of mutual exclusion. The ECMA documents [15] and [14] are also silent on this issue; they only mention that operations cannot be moved before (after) a volatile read (volatile write), thus leaving out the case when a volatile write is followed by a volatile read.

Our checker implements the .NET Common Language Infrastructure (CLI) instruction set specified in [15]. We allow reordering of operations by (a) requiring all bytecodes to issue in program order and (b) allow certain bytecodes (whose reordering is allowed by the memory model) to complete out-of-order. Allowing reorderings according to the .NET memory model involves additional data structures in the state representation of our checker. In particular, for each thread we now need to maintain a list of "incomplete" bytecodes — bytecodes which have been issued but have not completed. The execution model allows a program thread to either execute its next bytecode or complete one of the incomplete bytecodes. We now proceed to elaborate on the state space representation and the reachability analysis.

5 Invariant Checker

The core of our checker is a virtual machine that executes .NET Common Language Infrastructure (CLI) bytecode using explicit state representation. It supports many threads of execution by interleaving issuing and completing of bytecodes from all threads. We implemented only a subset of the CLI features. Features such as networking, I/O, class polymorphism and exception handling are not included in the implementation.

5.1 State Representation

We first consider the global state representation without considering the effects of the reorderings allowed by .NET memory model. To describe a global state we use the notion of *data units* of the CLI virtual machine. The virtual machine uses *data units* to hold the value of variables and stack items in the program. Each data unit has an identifier (for it to be referred to), and a modifiable value. The type of the modifiable value can be (a) one of the primitive data types, (b) reference types (pointers to objects), or (c) objects. New data units are created when a variable or a new object instance is allocated, or when a load instruction is executed. A global state of a program now consists of the following data units, corresponding to the different memory areas of the CLI virtual machine [15].

[1] Note that even if we allow (volatile-write → volatile-read) reorderings, we can still ensure that all writes to volatile variables are seen in the same order from all threads of execution.

Program counter for each thread Each thread has a program counter to keep track of the next bytecode to be issued.

Stack for each thread Each thread has a stack which is used by most bytecodes to load/store data, pass parameters and return values from functions (or certain arithmetic / branch operations).

Heap The virtual machine has a single heap shared among all threads. Object instances and arrays are allocated from the heap. A data unit is created for each object as well each of its fields.

Static variables A data unit is allocated for each static variable on its first use and this data unit is maintained for subsequent accesses.

Frame Frames store local variables, arguments and return address of a method. Each time a method is called, a new frame is created and pushed into frame stack; this frame is popped when the method returns. Each local variable/argument is assigned one data unit.

All of the above data areas of the virtual machine are included in the global state space representation of a program. Now, in order to support the memory model, a new data structure is added to each thread: a list of incomplete bytecodes (given in program order). Each element of this list is one of the following type of operations — read, write, volatile read, volatile write, lock, unlock (the operation types mentioned in the .NET memory model, see Table 1). This completes the state space representation of our checker. We now describe the state space traversal.

5.2 Search Algorithm

Our checker performs reachability analysis by an explicit state depth-first search (starting from the initial state) over the state space representation discussed in the preceding. Given any state, how do we find the possible next states? This is done by picking any of the program threads, and letting it execute a single step. So, what counts as a single step for a program thread? In the usual model checkers (which implicitly assume Sequential Consistency), once a thread is chosen to take one step, the next operation from that thread forms the next step. In our checker the choices of next-step for a thread includes (a) issuing the next operation and (b) completing one of the pending operations (*i.e.*, operations which have started but not completed). The ability to complete pending operations out of order allows the checker to find all possible behaviors reachable under a given memory model (in this case the .NET memory model).

Thus, the search algorithm in our checker starts from the initial state, performs depth-first search and continues until there are no new states to be traversed. In order to ensure termination of this search, our checker of course needs to decide whether a given state has been already encountered. In existing explicit state software model checkers, this program state equivalence test is often done by comparing the so-called *memory image* in the two states, which includes the heap, stacks, frames and local variables. Our checker employs a similar test; however it also considers the list of incomplete operations in the two states.

Formally, two states $\mathtt{s}$ and $\mathtt{s}'$ with two sets of data units $D = \{d_1, d_2, ..., d_n\}$ and $D' = \{d'_1, d'_2, ..., d'_n\}$ are equivalent if and only if the program counters in all threads are equal and there exists a bijective function $f : D \rightarrow D'$ satisfying:

- For all $1 \leq i \leq n$, the value stored in d_i and $f(d_i)$ are equal.
- A static variable $\mathtt{x}$ in $\mathtt{s}$ is allocated data unit d_i if and only if it is allocated data unit $f(d_i)$ in $\mathtt{s}'$.
- Data unit d_i is the k^{th} item on the stack (or frame, local variable, argument list, list of incomplete bytecodes) of the j^{th} thread in $\mathtt{s}$ iff $f(d_i)$ is the k^{th} item on the stack (or frame, local variable, argument list, list of incomplete bytecodes) of the j^{th} thread in $\mathtt{s}'$.
- The reference type data unit d_i points to data unit d_j in $\mathtt{s}$ if and only if $f(d_i)$ points to $f(d_j)$ in $\mathtt{s}'$.

In our implementation, the global state representation is saved into a single sequence so that if two state's sequences are identical, the two states are equivalent. Like the Java Path Finder model checker [9], we also use a hash function to make the state comparison efficient.

Search Optimizations By allowing program operations to complete out-of-order, our checker explores more behaviors than the normal model checkers based on Sequential Consistency. We employ the following search optimization to speed up our checker. For each thread, we classify its bytecodes into two categories — thread-local and non thread-local. In particular, the following are considered thread-local bytecodes — load/store of local variables, method invocation, memory allocation, computation and control transfer operations; all others are considered non thread-local bytecodes. Now, our checker exploits this categorization by trying to atomically execute sequences of thread-local bytecodes in a thread. Furthermore, our checker *does not allow two thread-local operations to execute out-of-order even if such reordering is allowed by the .NET memory model*. The justification of this optimization is simple — even if thread-local operations execute out-of-order, the effects of such additional behavior are not observable by other threads.

6 Disabling Undesirable Program Behaviors

Given a multi-threaded C# program, we are interested in computing the set of reachable states from the initial state. The set of reachable states under the .NET memory model is guaranteed to be a superset of the reachable state set under Sequential Consistency. In this section, we discuss tactics for disallowing the additional states reached under the .NET memory model. Since these additional states are reached due to certain reordering of operations within program threads, we can avoid those states if such reorderings are disabled by inserting barriers/fences in the program text.

While doing reachability analysis we build (on-the-fly) the state transition graph. Each vertex represents one state, each directed edge represents a transition from one state to another. Consider the state transition system constructed

for the .NET memory model. Because this memory model is more relaxed than Sequential Consistency, we can divide the graph edges into two types: solid edges correspond to transitions which can be performed under the Sequential Consistency (complete the bytecodes in order within a thread) and dashed edges correspond to transitions which can *only* be performed under .NET memory model (requires completing bytecodes out-of-order). From initial state, if we traverse only solid edges we can visit all states reachable under Sequential Consistency. We color the corresponding vertices as white and the remaining vertices as black. The black vertices denotes the additional states which are reached due to the reorderings allowed by the relaxed memory model (see Figure 1 for illustration). Note that if (a) we are seeking to verify an invariant property φ under Sequential Consistency as well as the .NET model, (b) φ is true under Sequential Consistency and (c) φ is false under the .NET memory model — the states violating φ must be black states. However, not all the black states may denote violation of φ as shown in the schematic state transition graph of Figure 1.

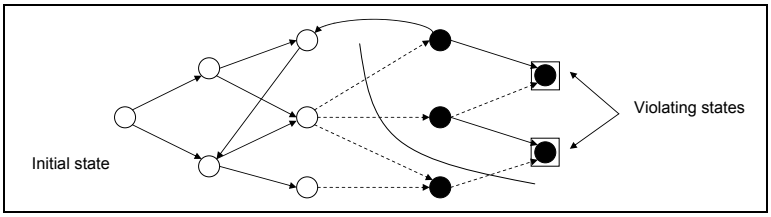

Fig. 1. State transition graph under a relaxed memory model; only white states can be reached under Sequential Consistency. A cut is shown separating the initial state from "violating" states.

Basic Mincut Formulation To prevent the execution from reaching the violating black states, we need to remove some of the edges from the graph. The solid edges cannot be removed because their corresponding transitions are allowed under Sequential Consistency. The dashed edges can be removed selectively by putting barriers. However note that the barriers will appear in the program text, so inserting *one barrier* in the program can disable *many dashed edges* in the state transition graph. We find out the minimal number of dashed edges to be removed so that the violating black states become unreachable; we then find out the memory barriers to be inserted in the program text for removing these dashed edges. Now we describe our strategy for computing the minimal number of dashed edges to be removed. We compute the minimum cut $C = \{e_1, e_2, ..., e_n\}$ where $e_1, \ldots, e_n$ are dashed edges in the state transition graph such that there is no directed path from the initial state to any violating black state (denoting violation of the invariant φ being verified) without passing through an edge in C. We find the minimal set of dashed edges by employing the well-known Ford-

Fulkerson maxflow-mincut algorithm [7]. To find the minimal number of dashed edges in the state transition graph as the mincut, we can set the capacity of each dashed edge to 1 and each solid edge to infinity.

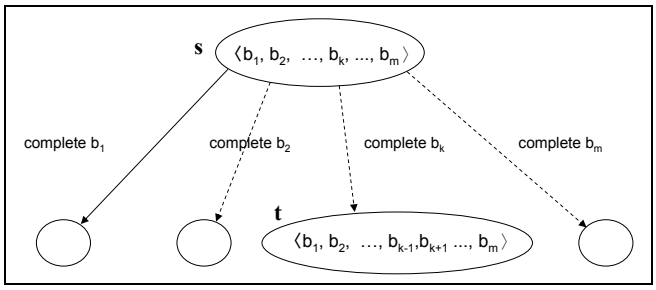

Fig. 2. Transitions from a state, a dashed edge indicates the transition requires an out-of-order completion of bytecodes

How can we locate the barrier insertion point in the program such that a given dashed edge in the state transition graph is removed? Recall that a dashed edge in the state transition graph denotes a state transition which is caused by out-of-order completion of a bytecode. In Figure 2 state s has m incomplete bytecodes $\langle b_1, b_2, \ldots, b_k, \ldots, b_m \rangle$ (given in program order). The transition that completes bytecode b_1 does not require an out-of-order completion of bytecodes while the transitions that complete b_k with $2 \leq k \leq m$ do. The removal of edge from state s to state t (corresponding to the completion of bytecode b_k, see Figure 2) is identified with inserting a barrier before bytecode b_k.

Modified Mincut Formulation Note that the minimal set of dashed edges in the state transition graph may not always produce the minimal number of barriers in the program text. At the same time, inserting minimal number of barriers in the program text may not be desirable in the first place since they do not indicate the actual number of barriers encountered during program execution.[2] However if we want to minimize the number of barriers inserted into the program, we can do so by simply modifying the capacities of the dashed edges in the state transition graph. We partition the dashed edges in the state transition graph into disjoint partitions s.t. edges e, e' belong to the same partition iff disabling of both e and e' can be achieved by inserting a single barrier in the program. We can then assign capacities to the edges in such a way that the sum of capacities of the edges in each partition is equal — thereby giving equal importance to each possible program point where a barrier could be inserted. The maxflow-mincut algorithm is now run with these modified capacities (of the

[2] A single barrier inside a loop which is iterated many times can introduce higher performance overheads than several barriers outside the loop.

dashed edges); the solid edges still carry a weight of infinity to prevent them from appearing in the min cut.

Complexity The Maxflow-mincut algorithm has time complexity of $O(m * f)$ where m is the number of edges in the state transition graph and f is the value of the maximum flow. The quantity f depends on how the capacities of the state transition graph edges are assigned. In all our experiments, f was less than 150 for all our test programs (using basic or modified mincut formulation).

7 Experiments

Benchmark	Description	# bytecodes
`peterson`	Peterson's Mutual exclusion algorithm [20]	120
`tbarrier`	Tournament barrier algorithm — *Barrier* benchmark from [8]	153
`dc`	Double-checked locking pattern [22]	77
`rw-vol`	Read-after-Write Java volatile semantic test by [19]	92
`rowo`	Multiprocessor diagnostic tests ARCHTEST (ROWO)[4]	87
`po`	Multiprocessor diagnostic tests ARCHTEST (PO) [4]	132

Table 2. Test Programs used in our Experiments

In this section, we report the experiments used to evaluate our checker. The multi-threaded programs used in our experiments are listed in Table 2. Out of these, `peterson`, and `tbarrier` are standard algorithms that work correctly under Sequential Consistency, but require more synchronizations to do so in the C# memory model. The tournament barrier algorithm (taken from Java Grande benchmarks) provides an application program level implementation of the concurrency primitive "barrier" (*different from our memory barriers which prevent reordering of operations*) which allows two or more threads to handshake at a certain program point.

The programs `rw-vol` and `dc` have been discussed recently in the context of developing the new Java memory model [1]. In particular, `dc` has been used in recent literature as a test program to discuss the possible semantics of volatile variables in the new Java memory model [6]; this program involves the lazy initialization of a shared object by several threads.

The other programs `rowo` and `po` are test programs taken from the ARCHTEST benchmark suite [4, 17]. ARCHTEST is a suite of test programs where the programs have been systematically constructed to check for violations of memory models (by generating violation of memory ordering rules imposed by the memory models). In particular, the program `rowo` checks for violation of ordering between multiple reads as well as multiple writes within a program thread;

the program `po` checks for violation of program order among all operations in a program thread. These programs are effective for evaluating whether our checker can insert memory barriers to avoid behaviors not observable under Sequential Consistency.

For all of the above benchmarks we employ our checker to find all reachable states under (a) Sequential Consistency and (b) .NET memory model. For the latter, recall that we allow each program thread to maintain a list of incomplete bytecodes so that bytecodes can be completed out of order. For our experiments we do not impose a bound on the size of this list of incomplete bytecodes. So in practice it is bounded only by the (finite) number of bytecodes in a program thread. This exposes *all* possible behaviors of a given program under the .NET memory model.

Benchmark	# states	# transitions	S.C.	.NET				
				Time (secs)				#
			(secs)	CE	FR	Mflow	Total	barriers
`peterson`	903	2794	0.09	0.05	1.06	0.04	1.10	3
`tbarrier`	1579	5812	0.21	1.57	3.99	0.05	4.04	3
`dc`	228	479	0.10	0.11	0.27	0.03	0.30	1
`rw-vol`	1646	5616	0.20	0.29	2.75	0.23	2.98	4
`rowo`	1831	4413	0.16	0.23	1.87	0.05	1.92	2
`po`	6143	22875	0.29	0.60	13.07	1.48	14.55	6

Table 3. Summary of Experimental Results. Column 4 shows the time to perform full reachability analysis under Sequential Consistency. Under the heading .NET, the CE column shows time to find the first counter-example, while FR shows time for full reachability analysis under .NET memory model. The column Mflow indicates the time to run the Maxflow algorithm for inserting barriers. The *Total* column denotes time for full reachability and barrier insertion, that is, $Total = FR + Mflow$.

Our checker for C# programs is itself implemented in C#. It takes the binaries of the benchmarks, disassembles them and checks the bytecode against a given invariant property via state space exploration. For each of our benchmarks in Table 2 we provide a program invariant for the reachability analysis to proceed and report violations. For the Peterson's algorithm (`peterson`) this invariant is the mutually exclusive access of shared resource. The invariant for `tbarrier` follows from the definition of the concurrency primitive "barrier". For the Double checked Locking pattern (`dc`) this invariant states that whenever the shared object's data is read, it has been initialized. The invariant for `rw-vol` benchmark is obtained from [19]. For the ARCHTEST programs `rowo` and `po`, this invariant is obtained from the rules of read/write order and program order respectively (see [4, 17]).

Our checker performs reachability analysis to explore the reachable states under Sequential Consistency and the .NET memory model. Clearly, the reachability analysis under the .NET memory model takes more time since it involves exploring a superset of the states reachable under Sequential Consistency. In

Table 3 we report the running time of our checker for obtaining the first counter-example (column *CE*) and for performing full reachability analysis (column *FR*). The time taken to find the first counter-example is not high; so if the user is only interested in detecting a violation our checker can produce one in a short time. The time to perform full reachability analysis (thereby finding all counter-example traces) is tolerable, but much larger than the time to find one counter-example. All experiments were conducted on a 2.2 Ghz machine with 2.5 GB of main memory.

After exploring all reachable states under the .NET memory model, our checker can insert barriers via a maxflow-mincut algorithm (we used the "Modified Mincut Formulation" presented in Section 6). The time to run the maxflow algorithm is small as shown in column *Mflow* of Table 3. The results of the barrier insertion step are shown in the *# barriers* column of Table 3. This column shows the total number of barriers inserted by our tool into the program so that any program execution trace which (a) violates the invariant being verified and (b) is disallowed by Sequential Consistency, — is not observed even when the program is run under the relaxed .NET memory model.

Interestingly, the reader may notice that our checker inserts only one barrier for the Double Checked Locking pattern (same as the solution in [16], [3] and [2]) while the solution using "explicit memory barriers" given in [6] suggests putting two barriers. Both solutions are correct, because they work for different memory models. Our checker only inserts those barriers that enforce the program's correctness under *a .NET memory model compliant implementation*. It will not insert barriers to disable behaviors which are not even allowed by the .NET memory model (the additional barrier in [6] is needed if the memory model allows reorderings which do not respect program dependencies).

More details about our checker (including its source code) and the test programs are available from `http://www.comp.nus.edu.sg/~release/mmchecker`

8 Discussion

In this paper, we have presented an invariant checker which works on the bytecode representation of multi-threaded C# programs. The main novelties of our work are (a) we can expose non sequentially consistent execution traces of a program which are allowed by the .NET memory model, and (b) after inspecting the counter-example traces violating a given invariant, we can automatically insert barriers to disallow such executions.

We are now in the process of integrating partial order reduction with dynamic escape analysis [5] into our checker. This will allow us to safely reduce the set of explored program states during invariant checking.

Acknowledgments This work was supported partly by a grant from Microsoft under the Shared Source CLI (SSCLI aka Rotor) programme, partly by a Public Sector grant from A*STAR (Singapore) and partly by an internal grant from NUS (Singapore). Tulika Mitra and Weng-Fai Wong gave us useful feedback during the project.

References

1. Java Specification Request (JSR) 133. Java Memory Model and Thread Specification revision, 2004.
2. B. Abrams. http://blogs.msdn.com/brada/archive/2004/05/12/130935.aspx.
3. C. Brumme. Weblog: Memory model. http://blogs.msdn.com/cbrumme/archive/2003/05/17/51445.aspx.
4. W. W. Collier. *Reasoning about Parallel Architectures*. Prentice Hall, 1992.
5. M. B. Dwyer et al. Using static and dynamic escape analysis to enable model reductions in model-checking concurrent object-oriented programs. Technical report, Kansas State Univ., 2003.
6. D. Bacon et al. The "Double-checked Locking is Broken" declaration. http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.
7. L.R. Ford and D.R. Fulkerson. Maximum flow through a network. In *Canad. J. Math*, volume 8, pages 399–404, 1956.
8. JGF. The Java Grande Forum Multi-threaded Benchmarks, 2001. http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/threads.html.
9. JPF. The Java Path Finder model checking tool, 2005. http://javapathfinder.sourceforge.net/.
10. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), 1979.
11. D. Lea. The JSR-133 cookbook for compiler writers. http://gee.cs.oswego.edu/dl/jmm/cookbook.html.
12. J. Manson and W. Pugh. The Java Memory Model Simulator. In *Workshop on Formal Techniques for Java-like Programs, in association with ECOOP*, 2002.
13. J. Manson, W. Pugh, and S. Adve. The Java Memory Model. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2005.
14. Microsoft. Standard ECMA-335 C# Specification, 2005. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.
15. Microsoft. Standard ECMA-335 Common Language Infrastructure (CLI), 2005. http://www.ecma-international.org/publications/standards/Ecma-335.htm.
16. V. Morrison. Dotnet discussion: The DOTNET Memory Model. http://discuss.develop.com/archives/wa.exe?A2=ind0203B&L=DOTNET&P=R375.
17. R. Nalumusu et al. The "test model checking" approach to the verification of memory models of multiprocessors. In *Computer Aided Verification (CAV)*, 1998.
18. T. Nipkow et al. Special issue on Java bytecode verification. *Journal of Automated Reasoning (JAR)*, 30(3–4), 2003.
19. W. Pugh. Test for sequential consistency of volatiles. http://www.cs.umd.edu/~pugh/java/memoryModel/ReadAfterWrite.java.
20. M. Raynal. *Algorithms for mutual exclusion*. MIT Press, 1986.
21. A. Roychoudhury and T. Mitra. Specifying multithreaded Java semantics for program verification. In *ACM Intl. Conf. on Software Engineering (ICSE)*, 2002.
22. D. Schmidt and T. Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *3rd annual Pattern Languages of Program Design conference*, 1996.
23. R.F. Stark and E. Borger. An ASM specification of C# threads and the .NET memory model. In *ASM Workshop, LNCS 3065*, 2004.
24. Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Memory model sensitive data race analysis. In *Intl. Conf. on Formal Engg. Methods (ICFEM)*, 2004.