A Memory Model for X10

Andreas Zwinkau
Karlsruhe Institute of Technology
zwinkau@kit.edu

Abstract

A programming language used for concurrent shared-memory programs must specify its memory model for programmers to reason about the behavior of a program. Java and C++ have plugged this hole in their specifications, but not X10. This paper proposes a memory model for X10. Additionally, this serves as a case study of how the design goals of a language map to requirements for its memory model.

Categories and Subject Descriptors B.3.2 [*Design Styles*]: Shared Memory; D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.0 [*Programming Languages*]: Standards; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Language Constructs and Features*]: Concurrent Programming Structures, Frameworks

Keywords X10 Programming Language, Memory Model, Memory Consistency, Sequential Consistency, Concurrency

1. Introduction

A memory model specifies what values parallel execution threads can observe in shared data in what order. This is important for the programmer, who reasons about the behavior of a program, and for the compiler, which must map it to the target hardware's memory model. Every implementation implicitly defines a model, but it becomes confusing if the programmer has a different model in mind, usually a simpler one.

The X10 specification [9] does not contain a memory model. This is especially perilous, since the compilation targets are Java and C++, which have different memory models [4, 8]. An X10 program's behavior should be the same, no matter which compiler backend is used. Thus, we must specify the behavior for the source language X10.

1.1 Overview

The contributions of this paper are

1. a memory model proposal for the X10 programming language
2. a case study how X10 design decisions influence the memory model compared to Java and C++

First in section 2, we collect our assumptions in a requirements analysis. Section 3 specifies the memory model. We discuss differences to Java and C++ in section 4 and also the issues with volatile, fences, constructors. The section also shows how threads map to activities, and why a global address space does not affect the memory model.

For an overview over memory models of programming languages, the popular short version is: "Sequential consistency (SC) for data race free programs". The formal version is:

> A program whose sequentially consistent executions have no data races must have *only* sequentially consistent executions.

To understand this, let us look into what SC and data race mean.

1.2 What is Sequential Consistency?

Memory models in general are about the order of memory *actions*, which are read, write, compare-and-swap, etc. Naturally, program code defines an order within a single thread. A compiler and a programmer can understand and work with this sequence. However, if two (or more) threads run in parallel, then many interleavings become possible and programs may become indeterministic. SC is originally defined by Lamport in 1971 [5]. Since terminology has changed a little since then, a modern definition is:

> Within each thread memory accesses follow program order and all threads immediately observe every access.

This implies a global total order of actions for a given execution.

For memory models we define the term *synchronization operations*, which explicitly synchronize global memory state for the thread that executes it. The simple SC memory model declares every memory access a synchronization operation. The problem is that this prohibits a lot of optimizations, which we want to allow compilers and CPUs to do. Thus, Java and C++ define a limited set of synchronization operations provided by locks, monitors, semaphores, atomics, and other special constructs. A library cannot provide such constructs [1], so they belong to the language specification.

With weaker memory models, one thread may observe a different order of actions than another thread. Obviously, this means additional complexity for a programmer, who tries to understand an execution in a debugger.

1.3 What is a Data Race?

If two memory actions are not synchronization operations, access the same data, and at least one writes, then they *conflict*. There may be a *happens-before* relation between the two actions, which means the first action is visible to (due to synchronization operations) and ordered before (due to program and execution order) the second. If two actions conflict and have no happens-before relation, then we have a *data race*.

Be careful not to confuse a data race with a race condition, which means that timing or ordering of events affects a program's behavior. Some only consider it a race condition, if the behavior

Let `flag` be `false` initially	
`flag = true`	`flag = true`

Figure 1. An example of a **data race without a race condition**. Two threads set the same flag to true. There is a data race, because those actions conflict and there is no happens-before relation. There is no race condition, because the outcome is deterministic. We do not care about the order in this case as the behavior is identical.

Assume at least 2 elements in `queue`	
`x = queue.pop()`	`y = queue.pop()`

Figure 2. An example of a **race condition without a data race**. Two threads try to take an element from a properly synchronized queue. Since `pop` is synchronized, there is no data race. There is a race condition, because it is not deterministic which thread gets which element.

is faulty/incorrect. We use a weaker definition here for brevity and simplicity. It does not matter for the memory model. Although data race and race condition often occur together, these are orthogonal concepts. Race conditions are the reason why we have so many different executions with parallel and concurrent programs. Figure 1 and figure 2 show examples of one, but not the other.

2. Requirements Analysis

Before we present the actual memory model, we analyze the requirements and derive design decisions.

The X10 compiler targets Java (called managed) and C++ (called native), and there is also an inofficial assembly backend [3]. This means whatever memory model we design, it must be possible to map it to the Java memory model (JMM) [8], the C++ memory model (CMM) [4], and hardware memory models.

Another aspect is that X10 targets High-Performance Computing (HPC), so performance is important.

2.1 Data Races are Undefined Behavior

Programmers strive for data race free code, so CMM considers data races as undefined behavior. "There are no benign data races" in C++ [4]. In contrast, Java must define the semantics of data races, otherwise a data race could be exploited to, e.g., circumvent the security manager. Via data races the current JMM fails [7] to prevent an execution from reading values "out of thin air", which were never written according to the program. Must X10 care about the semantics of data races and Thin Air Reads? No, because X10 provides no isolation mechanism within the language, which would have to be secure even with data races. Neither does X10 need to use a memory model framework [10], which supports this complexity.

If X10 defines semantics for data races, then the compiler must maintain this semantics in C++ and must not generate code with undefined behavior. Thus, the compiler must litter the code with additional synchronizing operations like memory fences, which degrades performance. This is not acceptable for HPC programs, thus X10 cannot provide a semantics for data races. While undefined behavior is a source of agony, the advantage is a simplified memory model.

2.2 Termination can be Assumed

With a similar argument, an X10 compiler can assume that all loops terminate. C++ [4, §1.10.27] allows to remove empty loops even if they might not terminate. Thus, the compiler can remove an empty loop in X10, if it is compiled directly to a C++ loop. Therefore, X10 must use an equivalently weak semantics or the compiler must ensure not to generate empty loops by inserting dummy statements. Since the upside of stronger semantics is not clear, we assume that empty loops can be removed and thus assume termination. We see an example in figure 3. This is the behavior of the current X10 version 2.5.

```
def foo(y:int,n:int):void {
  var x:int = 0;
  while (x < y) { x += n; }
}
```

Figure 3. We store the local variables `x`, `y`, `n` in registers, so there is no memory access within the loop. During the execution there is no "action" (see below) with respect to the memory model, so we consider the loop empty. Additionally, `x` is not used after the loop, so we do not care about its value. We cannot guarantee termination, since `n` might be zero. Still, the compiler can remove the loop.

3. X10 Memory Model

This sections provides a complete memory model for X10. The structure of this section mostly matches the Java memory model [8] §7, with the necessary parts from §5 and §9 merged in. Where it made sense, we copied the text verbatim for better comparison, so a lot of credit goes to the authors of the JMM. However, we changed many details in the adaption to X10.

In X10, an *activity* is the concept to model a thread of execution. A *place* is a shared memory domain. Activities within the same place use the same heap. Activities in different places cannot communicate via shared memory. Instead, the programmers must use the `at` construct to transfer an activity to another place, which implicitly copies context data.

3.1 Actions and Executions

An action a is described by a tuple $\langle t, k, v, u \rangle$, comprising:

- t the activity performing the action.
- k the kind of action.

  Most kinds are synchronization operations: activity creation (within the spawning activity), start and end of an activity, global termination of finish block, lock, unlock, library and external actions.

  Two kinds are not: read and write.
- v the variable or lock involved in the action. Variables and locks on different places cannot overlap.
- u an arbitrary unique identifier for the action.

As a notation for the variable or lock v of an action a, we use the notation $a.v$ in the following.

An execution E is described by a tuple $\langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V \rangle$, comprising:

- P a program.
- A a set of actions.
- $\xrightarrow{po}$ program order, which for each activity t is a total order over all actions performed by $t \in A$.
- $\xrightarrow{so}$ synchronization order, which is a total order over all synchronization actions in A. For $a_0 \xrightarrow{so} a_1$, we say a_1 is *subsequent* to a_0.
- W a write-seen function, which for each read $r \in A$, gives $W(r)$, the write action seen by r in E.

V a value-written function, which for each write $w \in A$, gives $V(w)$, the value written by w in E.

An *external action* is an action that may be observable outside of an execution, and may have a result based on an environment external to the execution. An external action tuple contains an additional component, which contains the results of the external action as perceived by the activity performing the action. This may be information about the success or failure of the action, and any values read by the action. Parameters to the external action (e.g., which bytes are written to which socket) are not part of the external action tuple, since it does not concern the memory model.

A *library action* is an action from the standard library, which provides additional synchronization mechanisms as shown in section 3.5.

3.2 Synchronizes-with and happens-before

Two additional relations are uniquely determined from an execution. Since we never reason about multiple executions at the same time, we do not annotate E with those relations explicitly.

$\xrightarrow{sw}$ *synchronizes-with*, a partial order over synchronization actions determined by synchronization order according to the rules below. We call the source of synchronizes-with *release* and the target *acquire*.

The total synchronization order of an execution determines a partial order synchronizes-with according to the following rules:

1. An unlock action on lock l synchronizes-with all subsequent lock actions on l.
2. An action that creates an activity synchronizes-with the start action of the created activity.
3. The write of the default value to each variable synchronizes-with the first action in every activity (Conceptually, every object is created at the start of the program). The default value of non-static val fields is their initialization value.
4. The end action of an activity synchronizes-with the end of the surrounding finish block.
5. The last action of an `atomic` or `when` block synchronizes-with the first action of subsequent `atomic` blocks and `when` conditions.
6. Further actions as specified in parts of the standard library in section 3.5.

A set of synchronizes-with relations is *sufficient* if it is the minimal set such that you can take the transitive closure of those relations with program order relations, and determine all the happens-before relations in the execution. This set is unique.

$\xrightarrow{hb}$ happens-before, a partial order over actions is the transitive closure of synchronizes-with and program order.

3.3 Well-formed executions

We only consider well-formed executions. An execution $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V \rangle$ is well-formed if the following conditions are true:

1. Each read of a variable x sees a write to x. For all reads $r \in A$, we have $W(r) \in A$ and $W(r).v = r.v$.
2. Synchronization order is consistent with program order and mutual exclusion. Having synchronization order consistent with program order implies that the happens-before order is a valid partial order: reflexive, transitive, and antisymmetric. Having synchronization order consistent with mutual exclusion mean that on each lock, the lock and unlock actions are correctly nested.
3. The execution obeys intra-activity consistency. For each activity t, the actions performed by t in A are the same as that activity t would generate in program order in isolation, with each write w writing the value $V(w)$, given that each read r sees/returns the value $V(W(r))$. The memory model determines the values seen by each read. The program order must reflect the program order in which the actions would be performed according to the intra-activity semantics of P, as specified by the parts of the X10 specification that do not deal with the memory model.
4. The execution obeys happens-before consistency. Consider all reads $r \in A$. It is not the case that $r \xrightarrow{hb} W(r)$. Additionally, there must be no write w such that $w.v = r.v$ and $W(r) \xrightarrow{hb} w \xrightarrow{hb} r$.

3.4 Termination

Like C++ [4, §1.10.27], any X10 activity will eventually "terminate, make a call to a library IO function, access or modify an atomic object, or perform a synchronization operation". This means in contrast to Java, we do not need to consider infinite executions. No "hang" action is necessary.

3.5 Constructs in the Standard Library

3.5.1 Atomics

The X10 runtime comes with atomic boolean, double, float, integer, long, and reference types in `x10.util.concurrent`. All method invocations of these constructs synchronize-with subsequent invocations on the same object.

3.5.2 Clock

For `x10.lang.Clock` the synchronizing methods are advance, advanceAll, drop, resume, resumeAll. All its method invocations synchronize-with subsequent invocations on the same clock object. In the case of advanceAll and resumeAll this affects all clocks the activity is registered at.

3.5.3 Condition

All method invocations of `x10.util.concurrent.Condition` synchronize-with subsequent invocations on the same object.

3.5.4 Lock

The `x10.util.concurrent.Lock` class provides (surprise!) a lock. All its method invocations synchronize-with subsequent invocations on the same object if the operation is successful. Other synchronization primitives Monitor, SimpleIntLatch, SimpleLatch, IntLatch, Latch inherit from `Lock`, so they need no special treatment.

For clarification, the behavior of the trylock method is equivalent in Java, C++, and X10. The method acts like `lock()` if a lock is not taken already and returns a boolean whether it has taken the lock. C++ explicitly gives trylock the freedom for "spurious failure" [2], which means it might fail to lock even if nobody else held it at the time. Java also provides this implicitly. The JavaDoc for `Lock` says

> Unsuccessful locking and unlocking operations, and reentrant locking/unlocking operations, do not require any memory synchronization effects.

If trylock should be synchronizing even if it fails, it requires a slower tryLock implementation with an additional fence instruction. There is a motivating example in figure 4, which C++ uses to motivate spurious failure.

```
x = 42          | while(l.tryLock())
l.lock()        |     l.unlock()
                | assert(x == 42)
```

Figure 4. Undesirable use of trylock from [2]. The second thread waits for someone else to take the lock[2]. The assert may fail, because there is no happens-before relation with the assignment according to JMM or CMM.

If the `tryLock`() succeeds in taking the lock, it would synchronize-with any previous locking operation. Via Lock semantics we know there cannot be a previous locking operation as we enter the loop. Thus, if we enter the loop, there is no synchronize-with relation between the threads.

If (or when) the trylock fails it has no synchronization effects. Thus, there is no synchronize-with relation to the first thread. We assume that `tryLock`() failing is an "unsuccessful locking operation".

Therefore, in either case there is no happens-before relation between assignment and assert. Hence, we have a data race and the value of x might or might not be 42. A compiler is free to move the assignment into the critical section, because nobody outside of the critical section can observe the assignment.

Even while there is nothing explicit about Javas `tryLock` being "spurious", this example demonstrates spurious behavior in Java. The JavaDoc of `tryLock`() says:

> Acquires the lock if it is available and returns immediately with the value true.

This is not wrong, but misleading in our example since the observed behavior looks like a spurious failure. There is no happens-before relation between assignment and assert, even if `tryLock`() correctly observed the lock as taken.

4. Discussion

4.1 Differences between X10 and Java

Compared to Java, X10 lacks three “features”, which simplify the memory model (and complexity of the language in general).

1. There is no `Thread` or `Activity` object in X10[3], so one cannot interrupt or join an activity. There is only the finish block, which waits for the (global) termination of all activities within.
2. X10 has no (user-defined) finalizers for objects. This also simplifies the model. See the JMM [8, §16] for the details. In general, finalizers are avoided, because execution is non-deterministic.
3. X10 does not provide reflection in a modifying way. The issues of modifying final fields in Java at any time do not exist. Corresponding to Javas final fields are val fields in X10.

4.2 Differences between X10 and C++

The CMM distinguishes between acquire and release synchronization, because this is relevant to support “relaxed” operations. However, such mechanisms are not provided by the X10 standard library nor supported by the compiler in any way. For high performance applications it might become worthwhile to provide this at some point, e.g., to implement non-blocking data structures. Then we must adapt the memory model.

[2] Advice for programmer: Convert `x` into an `AtomicInteger`, remove the lock, and spin on `x` directly.

[3] They exist hidden in the runtime, but not accessible to the programmer.

C++ has bitfields, where the compiler can compact fields to a certain amount of bits. This affects the memory model, since writes to bitfields are usually also writes to neighboring bitfields and might introduce data races. X10 has no such feature, which also simplifies the memory model.

4.3 Volatile Fields

X10 has no keyword volatile, but an annotation `@Volatile`. Useful documentation is missing. The C++ backend inserts the volatile keyword and the Java backend ignores it in version 2.3 and 2.4. The intended semantics seems to be that of C++, but not from Java. Namely, the C standard [4, §7.1.5.1/8]:

> `volatile` is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation.

The C++ semantics makes *no* guarantee about optimization by the hardware. Thus, `@Volatile` has no influence on the memory model. The Java backend of the X10 compiler cannot implement this semantics since Java provides no equivalent structure.

In version 2.5 the Java backend also inserts the volatile keyword, which has different semantics than in C++ and would have implications for the memory model. Java-volatile does make guarantees, like all reads and writes to such variables being atomic. The JMM specifically declares volatile accesses as synchronizing. Java-volatile fields seem to be unnecessary in X10, since the programmer could use constructs like `AtomicReference` instead. These also provide additional features, like a `weakCompareAndSet` method, but require slightly more code. The compiler should be able generate equivalent code. To implement Java-volatile behavior in the C++ backend, the X10 compiler would have to insert `std::atomic` implicitly, which might degrade performance unnecessarily.

Since neither Java nor C++ semantics for `@Volatile` are desirable, it should be removed from X10.

4.4 Fences

The `x10.util.concurrent.Fence` utility class provides four kind of barriers: load-load-barrier, load-store-barrier, store-load-barrier, and store-store-barrier. X10’s C++ backend only implements them for the PowerPC architecture. For the Java backend the implementation is broken. For these reasons we exclude them from the memory model.

Figure 5 shows the current Java implementation. The underlying idea comes naturally by looking at a table from the JMM Cookbook [6]. For example: If a volatile load is followed by a volatile store, the compiler must insert a LoadStore barrier. In the implementation of `loadStoreBarrier`, you can see a volatile load from `v1` followed by a volatile store to `v2`.

The issue is that the JMM defines a synchronizes-with relation for concurrent volatile accesses to the *same* variable. If the JVM can guarantee that a volatile variable is never accessed concurrently, then a compiler can remove the barrier. This seems feasible for figure 5. Additionally, this implementation probably adds a lot of overhead due to the actual memory accesses. The only correct solution would be that Java itself provides such barriers, but that is not the case. Currently, the pragmatic solution is to ignore `Fences` and use atomics instead.

4.5 StoreStore Barrier After Constructor

Our proposed X10 memory model, and also JMM and CMM, specify no relation between references and the referenced location. This includes for example the this-pointer of an object and its fields. If you assign to a field and share the object reference with another activity, reading the field might not yield the assigned value. This

```
static volatile int v1;
static volatile int v2;
static int d1;
static int d2;
public static void loadStoreBarrier() {
  v2 = v1;
}
public static void storeLoadBarrier() {
  v2 = d1;
  d2 = v1;
}
public static void loadLoadBarrier() {
  d1 = v1;
  d2 = v2;
}
public static void storeStoreBarrier() {
  v1 = d1;
  v2 = d2;
}
```

Figure 5. Broken implementation of fences for Java.

is counterintuitive, if the assignment is within the constructor to an immutable `val` field. Even if the object sharing is synchronized, the field accesses are not.

In Java, the final field semantics cover this even with a data race. In C++, this is undefined behavior. The implementation advice from the JMM authors [8, journal version] is a store-store-barrier at the end of a constructor. This is `NOP` on x86, which probably explains why nobody is hurt in practice. On ARMv8 the barrier is necessary, for example.

4.6 Global Address Space and the Memory Model

X10 models an asynchronous partitioned global address space (APGAS), but the memory model does not address this explicitly. It is not necessary, because an activity can only access its own part of the address space [9, sec. 2.4]:

> With remote reference, an activity can access objects at a remote place (remote objects) when the activity has moved to the remote place.

Moving an activity requires `at` and is already covered by program order. The implicit deep copy of context before and after `at` does not concern the memory model, apart from the normal read operations, which might or might not be properly synchronized.

It is possible to introduce arbitrary additional features using native code within the runtime, but also in user code. This extensibility is desirable and nobody wants to restrict that. In this case the documentation must include information about the synchronization. One example would be `Rail.asyncCopy`, which in version 2.5.4 specifies with respect to synchronization:

> The activity created to do the copying will be registered with the dynamically enclosing finish.

This implies the only way to synchronize is `finish` at the sending place. Thus, synchronization with the receiving place requires an additional `at`.

4.7 Threads and Activities

The JMM talks about threads, while we have activities in X10. For the memory model this does not matter much, since both just model parallel execution. Figure 6 shows an example of activity creation.

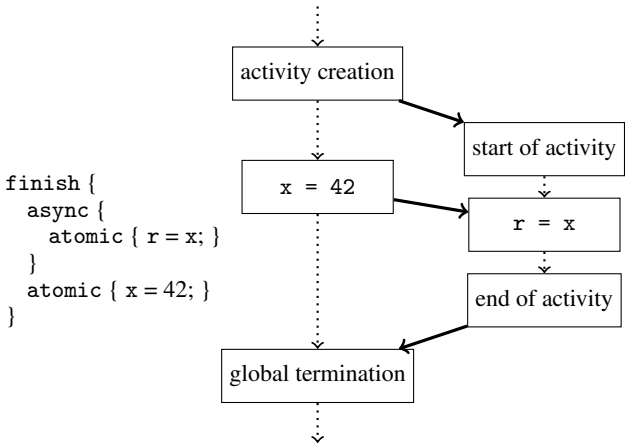

Figure 6. An example execution of an activity life cycle as written in the code on the left. Each box is an action. The thick arrows are synchronize-with edges and the dotted arrows show program order. The figure demonstrates the difference between "activity creation" and "start of activity" actions.

The JMM explicitly mentions the problem of thread inlining [8, fig. 12], which puts code of one thread into another one and removes parallelism. This must be forbidden in Java, because it introduces an additional happens-before relation and a compiler might optimize the program in an undesirable way. X10 uses lightweight activities, so the programmer is encouraged to create more of them than a Java programmer would create threads. Consequently, activity-inlining would be even more desirable than thread-inlining. The Java example relies on a data race and is thus undefined in X10. So, activity-inlining should be possible in X10.

5. Conclusions

We presented a possible memory model for X10 together with a rationale for its design. While we started with the JMM, due to X10's roots in Java, the outcome is closer to the CMM. One essential assumption of this proposal is the use case of high performance computing. If X10 is changing towards Java cloud computing and orchestration[4], then Java interop might get priority over performance. In this case, a semantics for data races might become necessary and we must adapt the memory model towards Java.

During the specification process, we uncovered minor issues in the X10 standard library. The `@Volatile` annotation has unclear semantics and might be unnecessary. The `Fence` utility class is broken and you should not use it. We have reported these issues as 3547, 3548, and 3549 in the X10 bugtracker[5].

A possible critique of the style of the JMM and this work is the distance to language semantics. Before inclusion in the language specification it would be worthwhile to express the memory model closer to the feature descriptions and language semantics.

Acknowledgments

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89). Thanks to Joachim Breitner, Manuel Mohr, Martin Hecker, Martin Mohr, Sebastian Buchwald, and the anonymous reviewers for valuable feedback and discussion.

[4] rumors via personal communication

[5] `https://xtenlang.atlassian.net/projects/XTENLANG`

References

[1] H.-J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Not.*, 40(6):261–268, June 2005. ISSN 0362-1340. doi:10.1145/1064978.1065042.

[2] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. *SIGPLAN Not.*, 43(6):68–78, June 2008. ISSN 0362-1340. doi:10.1145/1379022.1375591.

[3] M. Braun, S. Buchwald, M. Mohr, and A. Zwinkau. An X10 compiler for invasive architectures. Technical Report 9, Karlsruhe Institute of Technology, 2012.

[4] ISO/IEC 14882:2014(E). Programming Language C++. Standard, International Organization for Standardization, Geneva, CH, Nov. 2014.

[5] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979. ISSN 0018-9340. doi:10.1109/TC.1979.1675439.

[6] D. Lea. The JSR-133 cookbook for compiler writers. URL `http://g.oswego.edu/dl/jmm/cookbook.html`.

[7] A. Lochbihler. Making the Java memory model safe. *ACM Transactions on Programming Languages and Systems*, 35(4):12:1–12:65, 2014. doi:10.1145/2518191.

[8] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi:10.1145/1040305.1040336.

[9] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 language specification. Technical report, IBM, February 2014.

[10] V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 161–172, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-602-8. doi:10.1145/1229428.1229469.