Computer Science Technical Report

Toward a programmer-friendly formal specification of the UPC memory model

by William Kuchera and Charles Wallace

Michigan Technological University
Computer Science Technical Report
CS-TR-03-01
February 25, 2003

Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu

Toward a programmer-friendly formal specification of the UPC memory model*

William Kuchera and Charles Wallace
Michigan Technological University
{wrkucher,wallace}@mtu.edu

February 25, 2003

1 Introduction

As part of our efforts to elucidate the UPC memory model, we have closely examined the memory model definition given in the official UPC specification [3] (hereafter, "the UPC spec"). In this document, §5.1.2.3 (hereafter, "the memory model spec") gives most of the relevant details about the memory model. After examining this material for some time, however, we feel that there are several issues that need to be addressed with regard to the memory model's definition. Our position is *not* that the memory model as envisioned by the language's designers is somehow inappropriate or impractical; rather, we feel that its *definition* is inadequate and fails to capture what the language designers intend. Our list of concerns appears in §2. We have come up with an alternative specification of the memory model, using the operational semantics formalism of *Abstract State Machines (ASMs)* [1]. This alternative specification, which appears in §2, avoids the problems we find with the original specification; furthermore, we feel that it is simpler and more intuitive for the average programmer struggling with the intricacies of UPC.

2 Issues with the existing UPC specification

2.1 What does "observe" mean?

Item 3 of the memory model spec uses the notion of a thread "observing" an access (read or write) of a reference. The set of accesses that a thread observes plays a significant role in determining which values are visible to a thread in a given execution. But "observing" is never defined in either the UPC spec or the ISO specification of C [4]. We believe it needs to be explained, because it is not obvious what the term means. In particular, there is no association at all between what a thread *does* (in terms of detectable behavior during an execution) and what it "observes". The notion of observing lies on a plane totally divorced from actual behavior. As a result, any *witness* — any programmer, tester, or user who executes a given UPC program and sees the results — has no means to determine whether the visible behavior is even legal according to the memory model.

Consider an execution with the following sequences of operations:

*Financial support for this work has been provided by Hewlett Packard.

```
Thread 0:    write(x,1); fence; write(x,2)
Thread 1:    read(x,2); read(x,1)
```

Here, write($addr, val$) siginifes a write operation on address $addr$ that stores the value val; read($addr, val$) signifies a read operation on address $addr$ that returns the value val. We assume that Thread 0's writes are relaxed and the variable x is initialized to 0. Furthermore, Thread 1 writes the values it reads to an output device, so its reads (and, as a result, the operations it observes) do "affect... its input and output dynamics", as stated in item 3 of the memory model spec. Personal communication with Bill Carlson has confirmed our hypothesis that this should be an *illegal* execution. The fence should prevent Thread 1 from reading the value 1 at its second read.

Let us now put ourselves in the position of a witness of this program execution. We wish to check whether the behavior we have witnessed is in keeping with the restrictions of the UPC memory model. The only evidence available to us is what the program actually outputs. In this case, this includes the two values read for x. Armed with this evidence, we could reason as follows:

1. Since Thread 1 read the value 2 the first time, it must have observed Thread 0's second write.
2. If Thread 1 reads the value 1 the second time, it must have observed Thread 0's first write.
3. This implies that Thread 1 must observe Thread 0's fence, since it turns out that this fence does indeed "affect... its input and output dynamics". The reasoning here would go as follows: if Thread 1 does *not* observe the fence, it can read 1 the second time, but if it *does* observe the fence, it can *not* read 1. Since there is a possible difference in what is read (and output), Thread 1 had better observe the fence.
4. Then since the fence is really a strict reference, Thread 1 must observe Thread 0's first write before it observes the fence, and it must observe Thread 0's second write after it observes the fence.
5. This would seem to indicate that if Thread 1 reads the value 2 first, it has already observed Thread 0's first write, but the value of that write has been "overwritten" (in Thread 1's view) by Thread 0's second write. Therefore, Thread 1 cannot get 1 for its second read.

Notice that this reasoning makes a couple of important assumptions about what it means to observe an operation. The first assumption comes in steps 1 and 2: from seeing a given value a for address x written to output, we conclude that Thread 1 must have observed a write of a to x. The second assumption comes in step 5: once we establish that Thread 1 has observed writes of 1 and 2 to x in order, we conclude that the "older" value 1 has been "overwritten" by the "newer" value 2, and hence 1 is no longer visible to Thread 1.

As it is, nothing is said about what "observing" means – so its meaning is left up to the reader's imagination. This could easily lead people to invalid conclusions. Consider the following code fragments:

```
Thread 0:    x = -1; x = 1;
Thread 1:    if (x==1) y = 2;
Thread 2:    temp1 = y; temp2 = x;
```

Here, Thread 0's writes are strict, all others are relaxed, and x and y are initialized to 0. Say these fragments are executed concurrently, and Thread 2 gets the value 2 for y. It certainly seems

natural for a programmer to reason, "Since Thread 2 obviously observed Thread 1's write, and that write was contingent on x having the value 1, Thread 2 must also have observed Thread 0's second write." The programmer would then conclude that Thread 2 cannot get -1 when it reads x. This uses a more sophisticated, "causal" notion of observation, which seems perfectly reasonable. However, discussion with UPC experts has confirmed that this is not the intended interpretation; Thread 2 may indeed read the value -1.

2.2 How ordered does each "actual order" need to be?

For each thread t, the memory model spec defines a partial order "actual-order(t)", representing the order in which t observes accesses. This order is defined quite loosely — so loosely, in fact, that it permits certain behaviors of doubtful legality. Consider the following execution:

Thread 0: write(x,1); read(x,2); read(x,1)
Thread 1: write(x,2)

Is this a valid execution? It all depends on actual-order(0). If the two writes are ordered in actual-order(0), then Thread 0's first read would indicate that Thread 0's write is ordered before Thread 1's write. Following the "overwriting" assumption of the previous section, the value 1 would not be possible for the second read.

But actual-order(0) is simply defined as a partial order, so there does not seem to be any reason to assume that the writes are ordered with respect to each other. If the writes are not ordered in actual-order(0), then the second read could legally return 1. So should this be a legal result? Communication with Bill Carlson indicates that this is not intended to be legal.

If the intent is to disallow this result, something more about actual-order(0) must be stated in the memory model spec – perhaps, writes to a single location are linearly ordered in each thread's actual order.

2.3 The "least requirements" are undecidable

In item 3 of the memory model spec, the "least requirements on a conforming implementation" are given, with a final disclaimer, "UNLESS such a restriction has no effect on either the data written into files at program termination OR the input and output dynamics requirements". Briefly stated, the constraints of the least requirements do not apply if they have no effect on externally visible behavior. Interestingly, the condition in this disclaimer is undecidable. Consider the following code fragments:

Thread 0: x = 1; x = 2
Thread 1: f(x); flag = 1

We assume that calls to f never produce output. In an execution of this code, must Thread 0's first write precede its second write in Thread 1's actual order? According to the "UNLESS" clause, only if it has an effect on files or input/output dynamics. If function f halts on both input values 1 and 2, or if it fails to halt on both 1 and 2, there is no difference, so there is no need to order the writes. On the other hand, if f halts on one input but not the other, there will be a difference (in the value of flag), so the writes do need to be ordered. But determining whether an arbitrary f halts on a given input is, of course, undecidable.

It is not clear whether the presence of an undecidable predicate in the memory model specification is undesirable. It is certainly an interesting oddity.

2.4 The definition of "abstract order" is circular

The memory model spec relies on the notion of an "abstract order" on accesses. In item 3, we are asked to think of the accesses of each thread as labeled with integers according to their order of execution; these integers "monotonically [increase] as the evaluation of the program proceeds from startup through termination." In the abstract order, accesses by a single thread are ordered linearly according to their integer labels. However, there is a circularity here: the accesses that a thread performs, and the order in which it performs them, may depend crucially on the memory model. Consider the following code fragment:

Thread 0: `if (x == 1) y = 2; else z = 3;`

Assume that threads other than Thread 0 also access x. Let us establish the abstract order for the accesses by Thread 0. First of all, what are the accesses that Thread 0 performs? Clearly, a read of x — but then, it performs either a write to y or a write to z, depending on the value it reads for x. This in turn depends on the definition of the memory model, which is based on the abstract order.

The problem here is that the memory model rests on a well defined notion of "program order" of accesses, but "program order" is not well defined without a memory model. Note that many other memory-model definitions suffer from an *a priori* notion of "program order". Rudolph, Arvind and Shen comment on this in their original CRF paper [5].

3 An operational semantics approach

We wish to alleviate at least some of the problems with the current UPC spec by providing a precise memory consistency model using *Abstract State Machines (ASMs)* [1]. Our view of a UPC program execution is quite high-level, in keeping with the definition in the memory model spec: we conceive of each thread producing a stream of accesses (reads and writes), as well as barrier statements (fence, notify, and wait). At this point, we do not take into consideration the possibility of reordering these instructions, though we are currently working on this. Thus, our semantics are based at the architecture level rather than at the compiler level.

We begin by providing a translation from the official memory model spec to our semantics. This requires adding some material to the official spec: in particular, we provide a link between visible behavior and the memory model spec's notion of "observing". These axioms are based on our understanding of the language designers' intent, based on conversations with them.

If a thread performs a read at an address a and gets back a certain value v (as made visible to a witness through a write to a file or output device), what can we say about what it must have observed? It seems reasonable to infer that it observed a write of v to a. Our first axiom states that threads read values provided by earlier write accesses, rather than "out of thin air". Our second axiom ties together the notions of reading and observing: reading is only possible for *observed* write accesses. Finally, our third axiom asserts the assumption about write accesses "overwriting" one another.

- The "not out of thin air" axiom: If thread t performs a read at address a that returns value v, then t must read a write $w(a, v)$.
- The "observe what you read" axiom: If t reads a write w, then t must have observed w.

- The "overwriting" axiom: Let $w(a)$ and $w'(a)$ be writes observed by t. If $w < w'$ in actual-order(t), then t can no longer read w.

Universes

We now present our operational semantics. For brevity, we do not provide an introduction to the ASM formalism, but we do annotate the ASM definitions with comments which we hope will be illustrative. We begin by defining *universes*, which can be thought of as the basic data types of the ASM.

A distributed ASM consists of agents which execute rules of the ASM concurrently. Our ASM consists of thread agents of the universe Thread that execute instructions of the universe Instruction. A universe Action contains the type of actions that an instruction will perform.

In describing the memory model of UPC we are concerned with the ordering of shared memory "accesses". We define a universe Event containing all executions of write, read, and fence instruction in the system history. Read and write instructions operate on addresses, which form the universe Address. An address may take any value from the universe Value.

To facilitate global synchronization, UPC uses two so-called "split-phase" barrier statements, upc_notify and upc_wait. A thread cannot proceed past a upc_wait until all threads have executed the previous upc_notify. Additionally, an integer label is associated with each upc_wait or upc_notify statement. The label of any upc_wait must match the label of all upc_notify statements in the same phase. The universe BarrierLabel is the universe of barrier statement labels.

A synchronization phase is defined in §6.5.1 of the UPC spec as the collection of statements between upc_notify statements. An alternative definition of synchronization phase has been suggested in discussion on the UPC mailing list [2]; in the alternative, a phase is the collection of statements between upc_wait statements. We use the latter definition with one modification: in our version, a synchronization phase contains a upc_wait and all the preceding statements up to but not including the previous upc_wait. We introduce the universe Phase of synchronization phases.

Functions

We now define the functions used in the ASM to relate elements of the various universes. The interpretations of these functions, taken together, form the "current state" of the ASM.

Each thread agent needs to keep track of the instruction to be executed. The function currInstr maps each thread to its current instruction. Another function nextInstr maps an instruction to the next instruction to be executed. For any instruction we need the ability to extract information about its attributes: the instruction type, the address being read/written (in the case of a read or write), the value being written (in the case of a write), the barrier label (in the case of a notify or wait), the consistency mode (in the case of a read or write). The functions type, value, addr, label, and mode map an instruction to these various attributes. Of course, some of these attributes will be undefined for any given instruction.

When a thread issues a write, certain attributes of the write must be established. The functions thr, addr, and val map events to threads, addresses, and values, respectively. The function type maps events to their respective types: write, read, or fence. A thread perfoming a read uses these attributes when selecting a valid write.

In UPC, a thread is more restricted when reading a write issued by itself than reading a write from another thread. Only the last write to a location can be read by the thread that issued the write, although there may be many writes from other threads which can be read for that location. In the case of a single thread this mimics the behavior of sequential, non-parallel computations. To

keep track of the latest write we use the function maxLocal, which maps a thread and an address to a write.

A memory consistency model can be described in terms of two characteristics: the precedence relation between shared accesses and the values that can be returned from a read. The relation $\prec$ specifies how shared accesses are ordered across all threads. We use a partially ordered precedence relation to describe the hybrid memory model of UPC. It is helpful to think of each thread maintaining a history of shared acesses. All accesses created by relaxed instructions are ordered after the most recent strict access. A strict access is ordered after the most recent strict access in the history. The function maxStrict maps a thread to its latest strict access.

If a thread reads a write from another thread, it must obey the restrictions imposed by the memory model spec. To determine which writes it can read, a thread agent must keep track of the maximal write it has read from a remote thread. The function maxRemote maps the reading thread and a remote thread to the maximal event read from the remote thread.

To perform synchronization operations, thread agents must store information regarding phases and barrier labels. The function phase maps a thread to its current synchronization phase. Another function nextPhase maps a phase to the next sychronization phase. Each phase contains at most one notify statement and notify label. Once a thread reaches a upc_wait it must compare the label of the upc_wait to the label of all the upc_notifys issued in the same phase as the upc_wait. This comparison necessitates the function notifyLabel that maps a given thread and phase to a barrier label. If a thread has not yet reached the upc_notify in a certain phase the function is undefined. When a thread has reached a upc_notify, all waiting threads are brought up to date with regard to the thread's most recent event(s) before the upc_notify. The function phaseMax maps a thread and a phase to the most recent event before a notify.

Rules

We now give rules describing the dynamics of any UPC memory system. We view each thread as operating on a stream of instructions. With the exception of a wait instruction, we conceive of each instruction type as taking a single step.

module Thread:
let i = Self.instr
  case i.type **of**
  write: *Write* i.val *to* i.addr *in mode* i.mode
  read: *Read to* i.addr *in mode* i.mode
  fence: *Fence*
  notify: *Notify* i.label
  wait: *Wait* i.label
endcase

rule *Proceed*:
Self.instr := Self.instr.nextInstr

We begin by focusing on the actions associated with a write. When a write event is issued, the three attributes associated with it are updated. The write is also added to the thread's shared access history. How it is ordered with respect to other events in the history depends on the consistency mode of the write (strict or relaxed).

rule *Write v to a in mode m*:
extend WriteEvent **with** w
    w.thr := Self w.addr := a w.val := v
    Self.maxLocal(a) := w
    if m = strict **then** *Order new strict event w*
    else *Order new relaxed event w*
    endif
endextend
Proceed

When a strict access is created, it must be ordered in a thread's shared access history. The new strict access is ordered after the latest strict access and all relaxed accesses that follow the latest strict access. The new strict access then becomes the latest strict access. In contrast, a relaxed write only needs to be ordered after the latest strict access.

rule *Order new strict event e*:
do-forall d: Event: Self.maxLocalStrict $\preceq d$
    $d \prec e$
    Self.maxLocalStrict := e
enddo

rule *Order new relaxed write event w*:
Self.maxLocalStrict $\prec w$:= true

We now describe the dynamics behind a read access. To read from a shared address, a thread has two options: read a local write or a remote write. If a thread reads a write from itself, then it must choose the latest write to the corressponding location. When a thread reads a write from another thread, there may be multiple legal choices. For any pair of threads t_1 and t_2, there is a maximal event that t_1 has observed from t_2. A write is (remotely) readable if there is no intervening write between it and the maximal element read by the reading thread. As a result, the semantics of reading relaxed writes between two strict accesses is very relaxed indeed.

If t_1 reads a write from t_2 that is ordered after the current maximal element, the write is updated as the new maximal element. Keeping track of maxRemote allows us to limit which writes can be read by one thread from another. If a read is strict, it is ordered as a new strict access.

rule *Read to a for r locally*:
r.val := Self.maxLocal(a).val

term w.overwritten?:
($\exists w'$: WriteEvent: w'.addr = w.addr) $w \prec^+ w' \prec^*$ Self.maxRemote(w.thr)

rule *Read to a for r remotely*:
choose w: WriteEvent: w.thr $\neq$ Self and w.addr = a and not w.overwritten?
    r.val := w.val
    if Self.maxRemote(w.thr) $\prec^+ w$ **then** Self.maxRemote(w.thr) := w
    endif
endchoose

rule *Read to a in mode m*:
extend ReadEvent **with** r
    r.thr := Self r.addr := a
    choose among
        Read to a for r locally
        Read to a for r remotely
    endchoose
    if m = strict **then** *Order new strict event r*
    endif
endextend
Proceed

A fence is defined as a null strict reference, so it only affects the ordering of shared accesses. The fence is ordered as a new strict access.

rule *Fence*:
extend FenceEvent **with** f
    Order new strict event f
endextend
Proceed

We now turn to the dynamics of notify and wait. A thread that issues a notify in a given phase must keep track of the associated barrier label. A notify instruction also records the maximal strict event issued by the thread. Other threads will be brought up to date at least up to this maximal event when they reach the corresponding wait.

rule *Notify* ℓ:
Self.phaseMax(Self.phase) := Self.maxLocalStrict
Self.notifyLabel(Self.phase) := ℓ
Proceed

Three possible situations arise when a thread reaches a wait instruction. First, some thread may have issued a notify with a label different from the label of the current wait. This is illegal behavior, according to the UPC spec; all threads must share the same label for a given phase. A mismatch in label values is detected by labelMismatch? and, according to the UPC spec, results in a "runtime error". Both the UPC and ISO C specs are silent on the meaning of "runtime error",

and no answer has been forthcoming on the UPC mailing list. For the time being, we leave this portion of the ASM undefined.

The second situation arises if another thread has not reached the corresponding notify; in this case, the waiting thread continues to wait. The final situation is when all other threads have executed their respective notify instructions and there is no label mismatch. In this case, a thread proceeds and updates its current phase. It must also update maxRemote for itself across all other threads, bringing itself up to date with respect to accesses issued by the other threads.

term stopWaiting?(ℓ):
($\forall t$: Thread) Self.phase $\leq t$.phase and t.notifyLabel(Self.phase) $= \ell$

term labelMismatch?(ℓ):
($\exists t$: Thread) t.notifyLabel(Self.phase) $\neq \ell$

rule *Wait* ℓ:
if stopWaiting? **then**
    do-forall t: Thread: $t \neq$ Self
        Self.maxRemote(t) := t.phaseMax(Self.phase)
    enddo
    Self.phase := Self.phase.nextPhase
    Proceed
elseif labelMismatch?(ℓ) **then** *"Runtime error"*
endif

References

[1] Abstract State Machines home page. `http://www.eecs.umich.edu/gasm/`.

[2] Archives of `upc@hermes.gwu.edu`. `http://hermes.gwu.edu/archives/upc.html`.

[3] T. El-Ghazawi, W. Carlson, and J. Draper. UPC language specifications, v1.0. Technical report, Center for Computing Sciences, 2001. Available at `http://www.gwu.edu/~upc/doc/upc_specs.pdf`.

[4] Programming languages — C. ISO/SEC 9899, 2000.

[5] X. Shen, Arvind, and L. Rudolph. Commit-Reconcile & Fences (CRF): A new memory model for architects and compiler writers. In *Proc. ISCA*, pages 150–161, 1999.