Concurrent Constraint-based Memory Machines: A framework for Java Memory Models (Draft Preliminary Report, v 0.6)

Vijay Saraswat
IBM T.J. Watson Research Lab
PO Box 704
Yorktown Heights
NY 10598
(Please send comments to vijay@saraswat.org)

March 9, 2004

Abstract

A central problem in extending the von Neumann architecture to petaflop computers with millions of hardware threads and with a shared memory is defining the *memory model* [Lam79,AG95,APP99]. Such a model must specify the behavior of concurrent (conditional) reads and writes to the same memory locations. We present a simple, general framework for the specification of memory models based on an abstract machine that uses sets of (interdependent) *order* and *value* constraints to communicate between threads and main memory. The separation of order constraints allows a parametric treatment of different order consistency models such as *sequential consistency*, *location consistency*, *happens-before consistency* (HB-consistency), etc. The use of value constraints allows a simple formulation of the semantics of concurrent dependent reads and writes in the presence of look-ahead and speculative computation. Memory is permitted to specify exactly those linkings (mappings from read events to write events) which result in a unique solution for the constraints, and hence forces a unique patterns of bits to flow from write events to read events.

We show that this single principle accounts for almost all the "causality test cases" proposed for the Java memory model. To fix ideas, we present a structural operational semantics for a language using the HB order model. This operational semantics may be extended to develop a memory model for the Java programming language.

Contents

1 Introduction

The evolution of programming languages has been marked by increasing sophistication in application models, in machine architecture and in *programming models* (or abstract machines) that bridge the gap between the two.

For many years the von Neumann *Sequential Imperative Programming Machine* has defined an elegant and simple framework for computation. Abstractly, a programmer could think of the state of the computation as a *store* (a mapping from variables to values) together with instructions to extend the store (create new variables) and modify the store. Programs could be understood in terms of the effect they had on the store; operationally the result of executing a program was to obtain a series of stores, each related to the previous through a *transition*. Programs were designed so that the result of interest could be read off from the final store in the sequence, i.e. on termination.

The central value of a programming model is that it provides a mutually convenient shared abstraction between the application developer and the implementer (compiler-designer, operating system designer, hardware designer ... henceforth called *system implementer*). A properly designed programming model successfully resolves the tension between the demands for simplicity and generality arising from the application developer and the demands for simplicity and efficiency arising from the system implementer. It provides for a *decoupling* between the concerns of the developer (organizing of programming abstractions, type systems, program transformations, reusable libraries, reasoning tools, semantic models) and the system implementer (exploiting hardware and systems technology to provide a cheaper/faster/smaller/scalable/more powerful/more energy efficient implementation of the programming model). The system implementer may cut whatever corners he wishes, use implementation techniques limited only by Physics, his Imagination and the Golden Principle of Compilers: *Dont get caught*. Dont violate the programming model.

We are at a crucial stage in the development of the von Neumann model. The challenges thrown up by the demands for *parallel excution* have been around since at least the 1970s. Leslie Lamport [Lam79] first identified a simple set of rules (*Sequential Consistency*) which provide a programming model for machines with multiple hardware threads that can access the store simultaneously. Intuitively, the model states that an ensemble of parallel programs may be thought of as executing one step at a time. In each step, only one program may access memory, completing its operation before allowing the system to take the next step. In many ways this is a natural generalization of sequential execution to an *interleaving* model of concurrency. A parallel computation may be thought of as resulting in a sequence of states which is some interleaving of the sequence of states of each of the parallel constituents.

However, as computer designers scale their systems to millions of hardware threads directed at achieving petaflops of computational power, the cost of satisfying sequential consistency are becoming apparent. Parallelism in computation creates a demand for parallelism in the access to memory. Fundamentally, at any given instant such threads may have millions of memory operations outstanding,

in various stages of completion across intevening buses, bypass buffers, caches, memory banks etc. If these operations access different regions of memory, no changes to the programming model are needed because the Golden Principle is not being violated. The difficulty arises when multiple threads wish to access the same memory location at the same time. Therefore to properly account for this phenomena, one is forced to abandon the interleaving model of concurrency and move to a model where there may be simultaneous reads and writes to the *same* memory location by multiple threads.

The *memory model* problem is thus the problem of defining the semantics of concurrent reads and writes to shared memory locations. It must take into account that a thread may submit a *conditional write*, specifying a condition under which the write operation should be performed. The condition may arise from *speculative execution*: statements under a conditional are being executed speculatively without their condition being discharged. Or it may arise from *out of order* execution: a statement after a conditional may be submitted ahead of time under the assumption that the conditional will not succeed. The thread may submit *data-dependent* writes, specifying the value to be written as a function of other values to be read. The thread may also submit ordering constraints on events specifying, e.g. that a write must (appear to) execute before a read.

The job of the memory model then becomes the job of specifying which of these reads can be answered by which of these writes (thereby determining what value is returned by the read), while respecting the logical properties of conditional execution and the ordering constraints between events.

We consider the development of a framework for solving the memory model problem a central challenge in the further development of the von Neumann model [AG95,APP99]. It is an important problem because resolving it successfully will pave the way for the design of high-level imperative, concurrent programming languages that can be implemented portably and efficiently on state of the art multiprocessors. It is a problem large in scope: one is seeking a complete revision of the foundations of imperative concurrency, which will end up having an impact on denotational semantics, compiler transformations, proof techniques, hardware implementation techniques, programming language design etc. It is a hard problem: one seeks a uniform framework that is not sensitive to the details of a particular consistency model, since there are a variety to choose from.

1.1 JAVA Memory Model

The thorny nature of this challenge is highlighted by the issues in developing the JAVA *Memory Model* (JMM). The 1996 First Edition of [GJSB00, Chapter 17] defined the semantics of threads in JAVA through an abstract machine. Multiple threads communicate with each other through a shared main memory, as depicted in Figure 1. Each thread maintains its own call stack, thread-local variables and a *working memory* of objects in the main memory. It performs a *use* operations to read the value of a working memory variable into the execution engine, and an *assign* operation to write the value into working memory variable

from the execution engine. Main memory uses a *read* action to read the contents of the master copy of a variable into a thread's working memory. A thread performs a *load* action to place the value read from main memory into working memory. A thread performs a *store* action to transmit a value from working memory to main memory. Main memory uses a *write* action to write this value into the variable in main memory. *Lock* and *unlock* operations are performed jointly by a thread and main memory.

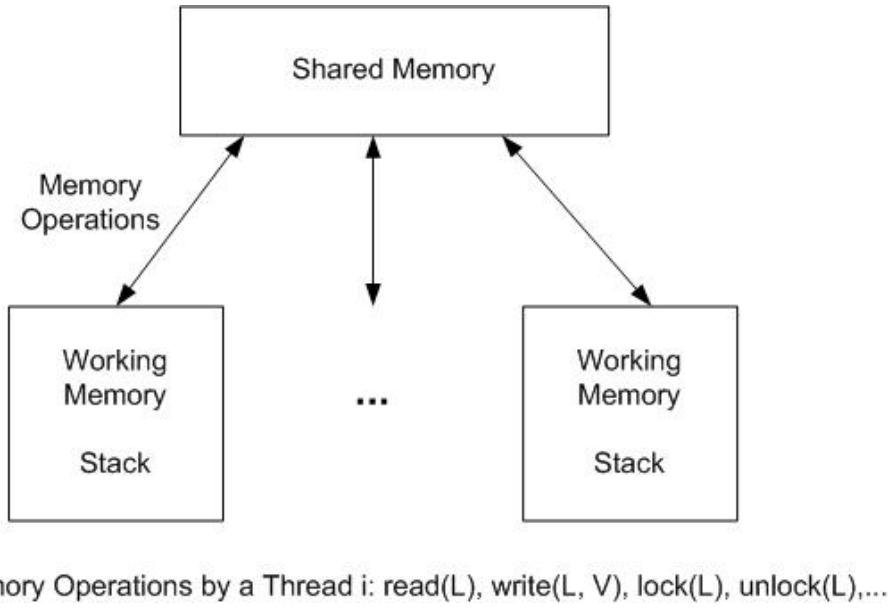


Figure 1: A schematic abstraction for shared memory multi-processors

Rules are provided for the order in which events may occur. The basic intention appears to have been that correctly synchronized programs[1] should appear to execute in a sequentially consistent manner [Lam79,AG95]. However, as a concession to then current multiprocessor architectures, the abstract machine allowed *prescient reads*: a read operation on a variable may return a value that is yet to be written to the variable, as long as synchronization conditions are not violated. Unfortunately, this means that programs that are not correctly synchronized may exhibit startlingly unexpected results. Therefore a cleanly specified and understandable memory model became critical.

While [GJSB00, Chapter 17] appears to provide a solution for this problem, it suffers from several major drawbacks. First, the set of rules specified are quite complex and not at all well-motivated for programmers. In the terminology introduced below, the specification is an "incomplete step semantics" (Section 1.3). The consequences of these rules were not worked out. For instance,

[1] A correctly synchronized program has no *data race*. A program has a data race if there is an execution sequence in which one thread is writing to a location and another reading/writing to it without any mutual synchronization.

it was later discovered that these rules imply *memory coherence* (see [GS97], [Pug04c]; informally, coherence is the property that all threads see operations on a shared variable in the same order), a property which is known to require significant overhead on shared memory multiprocessors. Second, the description was confusing enough that several existing implementations were found not to conform to the intended interpretation. Common compiler optimizations were not valid for the intended semantics [Pug04c].

1.1.1 JMM requirements

Since then, a Java Specification Request [Pug04a] has been comissioned to specify a *Java Memory Model* (JMM). While many details are contentious, the following requirements on the model seem to be commonly accepted:[2]

M1: Programs that do not exhibit data races in sequentially consistent executions must behave as if they are sequentially consistent. ("Correctly synchronized programs cannot go wrong.")

M2: Each read of a variable must see a value written by a write to that variable. ("No thin air reads.")

M3: The model must support efficient implementation on current multiprocessor architectures based on processors such as the PowerPC, IA64, IA32, and SPARC.

M4: Removing useless synchronization should be semantically valid.

We propose some additional requirements:

M5: The model should be simple to understand for programmers. It should be framed as an *abstract machine*, that is, in the form of configurations and transitions over those configurations, in a manner familiar from operational semantics.

M6: The model should be *generative*. Given a program, it should be possible to use the model to generate all possible execution sequences of the program.

From the semantics it should be possible to ascertain unambiguously whether a proposed compiler optimization is in fact sound (preserves the semantics of the program) or not.

1.2 Some motivating examples

The test cases discussed below are taken from [Pug04a]. Below we use the syntax of a mini programming language described in Section 3. The details should be familiar to most readers.

[2]The list below is our distillation of the requirements stated in [Pug01].

Figure 1 from [Pug04a]

Program	*Transformation*

`init A=0;`
`init B=0;`
`thread {`
`  r2=A;`
`  B=1;`
`} |`
`thread {`
`  r1=B;`
`  A=2;`
`}`
	`init A=0;`
	`init B=0;`
	`thread {`
	`  B=1;`
	`  r2=A;`
	`} |`
	`thread {`
	`  A=2;`
	`  r1=B;`
	`}`

Behavior:	`r1==r2==1`
Decision:	ok.

Table 1: Statement reordering may cause unexpected interactions

Example 1.1 (Out of order writes) Consider the program in Table 1. It is easy to see that a sequentially consistent execution cannot yield `r2==2` and `r1==1`.

Suppose we extend the model so that it may execute several instructions at the same time, restricted only by the requirement that for local variables it must always read the value that it last wrote into that variable. Now the machine for Thread 1 may issue a read for variable `A` from memory, and without waiting for it to complete, may go ahead and issue a write for variable `B`. Similarly the machine for Thread 2 may issue the read for `B` and simultaneously the write for `A`. The memory now sees these pair of "simultaneous" (i.e. unordered) requests arriving from Thread 1 and Thread 2. Its job is to match up these requests, responding to a read from a location with a value that was written into that location. Now it is easy to see how the values `r2==2` and `r1==1` are generated. □

Example 1.2 (Promoting conditional writes, Test 6) Consider Test 6 in [Pug04b], reproduced in Table 2. A sequentially consistent execution must yield `0` for either `r1` (Thread 1 goes first) or `r2` (Thread 2 goes first). However, as argued in Table 2, there is a plausible execution which can lead to the desired result. It hinges crucially on the implementation being able to examine a set of actions by Thread 1 and Thread 2 at the same time, and being able to reason conditionally about the consequences of these actions. □

Example 1.3 (No thin-air reads.) Consider Test 4 reproduced in Table 3. Two threads are copying values from one variable to the other in parallel. The

Test 6 in [Pug04b]

Program

```
init x=0;
init y=0;
thread {
 r1=x;
 if (r1==1) {
   y=1;
 } |
thread {
 r2=y;
 if (r2==1) {
    x=1;
 }
 if (r2==0) {
    x=1;
 }
}
```

Behavior:	`r1==r2==1`
Decision:	ok.

One may argue for this behavior as follows. Thread 1 communicates to main memory an unconditional write `r1=x`*, and a conditional write* `y=1` *provided that* `r1==1`*. At the same time, Thread 2 communicates an unconditional write of* `y` *to* `r2`*, and a conditional write of* `1` *on* `x` *if* `r2` *is* `0` *or* `1`*. Now main memory may choose to answer the read of* `x` *on line 1 by reading it from the write on line 6 if* `r2=1`*, and from the write on line 6 if* `r2=0`*. The read of* `y` *on Line 4 may be answered by the write on Line 3 (if* `r1=1`*) and by the initial value otherwise. Adding this linking to the constraints communicated from the threads results in a single solution, namely* `1` *for* `r1` *and* `r2`*.*

Table 2: Speculative execution may cause unexpected interactions

Test 4 in [Pug04b]

Program

`init x=0;`
`init y=0;`
`thread {`
`  r1=x;`
`  y=r1;`
`} |`
`thread {`
`  r2=y;`
`  x=r2;`
`}`

Behavior:	`r1==r2==1`
Decision:	Forbidden.

It should not be possible for values to be generated from thin air even in the presence of race conditions.

Table 3: Thin-air reads are not allowed.

variables are initialized to 0. Consider now that memory has simulatenously received all four read/write events. It may satisfy the read for `x` from Thread 1 by answering it with the write for `x` by Thread 2, and the read for `y` in Thread 2 with the write in Thread 1. This imposes no conditions on what values are transferred. So, theoretically, it might be possible to say that any value could be transfered, e.g. `1`. However, the semantics should not allow this.

□

No Optimization without Semantics. Before moving on to a discussion of what such a model should look like, we wish to discuss another seemingly plausible (but we think ultimately misleading) way to motivate an alternative to sequential consistency, that appears to underlie the discussion on the Java Memory Model mailing list.

One may observe that commonly compilers are free to reorder code as long as they observe The Golden Principle ("Dont get caught"). Now for the program in Table 1, the original code for Thread 1 is identical to the rewritten code for Thread 1 as far as Sequential Execution semantics are concerned: they both compute the same function from stores to stores. Thus the compiler should be free to rewrite the code in the way shown in the table. The rewritten code, when executed by a Sequentially Consistent Machine will show the given behavior.

There is a single global sequence of steps.

The reason we think this is a misleading way to define a programming model is that unfortunately the range and nature of these compiler optimizations cannot be fixed beforehand. Thus a programmer has no effective way of knowing *how* a compiler might reorder code, and therefore how his program might behave. Second, this notion of a program model being defined in terms of "all legal rewrites of the code" appears very unnatural and counter-intuitive to most programmers. We assert that programmers prefer a "machine oriented" view: a view in which the rules for a machine executing the code are directly specified, e.g. as transformations of machine configurations (in the usual way of operational semantics). Such an operational model should define the framework within which the compiler-writer must operate. Clearly, this machine will be more useful than the Sequentially Consistent Machine only if it permits more of the aggressive "compiler optimizations" designed to improve performance. Once the model is defined we can *check* that these compiler optimizatons are valid, i.e. they preserve the semantics of the program. We prefer this approach to the approach of taking the collection of desired optimizations to *define* the semantics of the program, and summarize it with the slogan "No optimization without semantics".

1.3 Our proposal: Concurrent Constraint-based Memory Machines

Instead of reasoning about possible compiler optimizations, we desire to formulate a family of simple shared memory abstract machines that can be used to generate all possible behaviors of a given program. We call such an abstract machine a *concurrent constraint-based memory machine* (CCM Machine).

The standard way to extend the Sequential Imperative Programming Model to the current setting as follows. A configuration would be the state of the shared main memory and the program being executed at each thread. In one step, a thread executes an action, possibly updating memory, and determines the next instruction for it to execute. Thus the steps taken by a parallel composition of statements are obtained as an interleaving of the steps taken by each component separately.

If each step *must complete* the action (and all its attendant updates to memory), then, as we have discussed above, this technique can only give rise to sequentially consistent execution sequences. Therefore to accomplish the task at hand there are two basic choices.

Incomplete step semantics. The first choice is to permit a single step to initiate an action but not complete it. For instance, an assignment step may be modeled by adding events (corresponding to the reads and writes in the step) to the store, without requiring that the effects of these events be reflected in the new store. Over time, such a store would autonomously pick up "incomplete" events, and add additional information (e.g. ordering information) to "link" it

into the rest of the store. Thus a step in the computation may be initiated by a thread in the program or by memory, and a memory operation may take several steps to complete.

Such an approach is feasible. Indeed this is the approach taken in [GJSB00, Chapter 17] and by various papers on consistency models oriented towards architecture [GLL+90]. Unfortunately the book-keeping required to model memory operations (particularly synchronization operations) in various stages of completion can be intricate. A lot of details about mechanisms needs to be formalized – detail which is arguably at the wrong level of abstraction. Seemingly arbitrary concrete decisions need to be made – discussions which arguably belong in an implementation, not in semantics. The central task at hand is to give a simple unambiguous account of the semantics of concurrent memory operations that can be used as a compact between the programmer, the compiler writer and the system architect, and these details are at the wrong level of abstraction.

Concurrent step semantics. Instead we turn to another approach, an *invariant-oriented* approach. We model the store as a collection of events and associated pieces of information that represents a set of *completed* memory operations. As discussed earlier, to exploit the power of modern hardware, this set should allow speculative execution, conditional look-ahead execution etc. Such a store must be *valid*: it must satisfy a set of invariants arising from the memory model we are trying to formalize. A transition takes a valid store to a valid store. To correctly model concurrent operations on memory, a transition must represent a *set* of memory operations (e.g. read, write, synchronize), possibly arising from different threads, that are to be thought of as executing "at the same time". The new store obtained on completion of the transition must reflect the successful completion of *all* of these operations. In contrast with the incomplete step semantics, such an approach does not allow for autonomous memory transitions.

We shall call the set of events and associated pieces of information communicated by one or more threads to memory an *action set*. (Detailed definitions are given in Section 2.) As discussed earlier, an action set may specify an ordering on events, conditional introduction of events, and dependent writes.

Given such an action set, what must memory do? First, it should add the action set to the store. Second, it may need to add ordering information between events in the action set, arising from different threads, so that the invariants associated with a valid store may be satisfied. For instance, if the action set contains two `lock` events from different threads targeted at the same location, it must ensure that these events are ordered. It may also need to add ordering information between events in the old store and events in the new store (e.g. to record that the events in the action set occur after the events in the store, which represents a record of past events). Third, it must find a *linking*, which matches each read event in the input action set to a write event on the same location. The write event may already have occurred in the past (i.e. may be in the store) or may be supplied in the input action set. Crucially, the write event

must be *visible* to the read event according to the ordering rule between different events specified by the memory model. Further, the linking must unambiguously specify the *value* (i.e. pattern of bits) that is read from a write event. Thus in the new store every read will have been satisfied by a write on the same location that is visible to the read, and all necessary ordering relations between synchronization events will have been established.

Let us make explicit certain operations that memory must *not* do. It must not add ordering information between events in the action set supplied by the same thread (either now or in the past); such information can only be supplied by the thread. It must not add ordering information between events that have already occurred in the past; for that would be tantamount to changing the past. Note also that most operations are targeted at a single memory location (e.g. reads and writes, but not *fences*) and to implement them ememory should only add ordering information between events targeted to that location. Finally, memory should not introduce "magic writes" – i.e. answer reads by pulling values (patterns of bits) out of thin air. It must merely *link* reads to reads, not create values.

This should complete one basic step of execution. Execution should begin in an "empty store" (reflecting no past interactions), and proceed by accepting an action set from threads, computing the new store and repeating until done.

Constraints. The central question that remains is how these action sets are to be specified. Since stores are just an "integral" of action sets over time, this will also determine how stores are specified.

We shall use the technique of *constraints* to specify the linkages between reads and writes. The use of constraints for communication and control in concurrent programming was discussed at length in [Sar93] (see specifically Section 1.4.2). Constraints allow the partial specification of information (e.g. the linkage between reads and writes, without requiring that the value be known). Constraint imposition is commutative, associative and idempotent (i.e. the order of constraint imposition is irrelevant). Because of this, constraints support concurrency: they may simultaneously (and asynchronously) be added to a shared store by multiple threads. Constraints are *additive* – a collection of constraints may imply more than what is implied by each one separately. Constraints are *declarative* – the notion of a solution of a constraint (and of one constraint entailing another) can be understood purely in logical terms, independently of the details of algorithmic manipulation performed by the constraint solver.

Thus, we shall take an action set (and hence the store) to be a constraint. We shall require the invariant that the store uniquely specify the value of all read and write operations; that is, it has a *unique solution* for program variables. Additional invariants are associated with the particular choice of *Order Model* (see below). Typically, all Order Models require that the the ordering on events be acyclic, and that all events targeted at a particular location by a particular thread are totally ordered.

Order models. Our approach is parameteric on the order relation between events, and the closely related notion of *write-visibility* (which writes can be seen by which reads, based on the order relation). Given a particular choice of order, we shall obtain a specific CCM Machine.

We identify several machines of interest. The *Sequential Consistency* (SC) Machine establishes the requirement that all events in an action set arising from a single thread are totally ordered, and that all events in a store are totally ordered. Given input action sets from multiple threads and a store, the only work that needs to be done to obtain the new store is to embed the set of total orders (one for each thread) into a single total order, i.e. to interleave the events from each thread.

The *Location Consistency* (LC) Machine is based on the very weak notion of location consistency of Gao and Sarkar [GS00]. In such a machine two data accesses (non-synchronization accesses) are considered to be ordered only if they are made by the same thread to the same variable.

The *Happens Before* (HB) Machine is based on the happens before relation, particularly as developed in [Pug04a], which specifies a total order on all events emanating from a thread, and an edge between a synchronized read event and a write event that answers it.

We show that the LC and HB machines agree on the results of the Test Cases posed in [Pug04b]. We discuss some desirable properties that are satisfied by these machines in Section 2.6.

Other Machines are possible, particularly those based on processor consistency, weak consistency and release consistency. Out of considerations of space a detailed development of these machines and their properties is reserved for future work.

1.4 Examples revisited

Consider the examples discussed earlier in the context of the LC machine. For the sake of simplicity, we elide the description of the underlying events, and their ordering relations, and rely on the naming convention of the variables to match them up to the underlying events. (A more detailed example is presented in Section 2.5.)

Example 1.4 (Table 1) The first thread sends two write events and a read event, with the associated constraints `w(R2)=r(A1r), w(B1w)=1`. These constraints may be read as saying that the value written into the register `R2` is the value read by Thread 1 from location `A`, and the value written into location `B` by Thread 1 is `1`.

The second thread similarly sends `w(R1)=r(B2r), w(A2w)=2`.

Now main memory can choose to answer each read from the corresponding write received from the other thread, through the linking `r(A1r)=w(A2w), r(B2r)=w(B1w)`. This estabishes `r(R1)=2, r(R2)=1`. □

The next example shows that memory can respond to conditional writes by making conditional linkages.

Example 1.5 (Test 6) Thread 1 can be seen as communicating

```
w(R1w)=r(X1r), (r(R1r)=1 -> w(Y1w)=1)
```

and Thread 2 can be seen as communicating

```
w(R2w)=r(Y2r), (r(R2r)=1 -> w(X2w1)=1), (r(R2r)=0 -> w(X2w2)=1)
```

Given these action sets the store can establish the linking

```
r(X1r)=(r(R2r)=1)?w(X2w1):((r(R2r)=0)?w(X2w2):w(Xi)),
r(Y2r)=(r(R1r)=1)?w(Y1w):w(Yi),
r(R1r)=w(R1w),
r(R2r)=w(R2w),
```

This linking states that `X1r` should receive the value from `X2w1` if that write action executes successfully, and if not, from `X2w2` (if that action suceeds), and if not, from `Xi`. Similarly for `Y2r`. This linking together with the other constraints establishes `r(R2r)=1` and `r(R1r)=1`. No other solutions are possible.

□

Example 1.6 (Test 4) The two threads may be seen to communicate

$$\mathtt{w(R1w)} = \mathtt{r(X1r)}, \mathtt{w(Y1w)} = \mathtt{r(R1r)}$$

and

$$\mathtt{w(R2w)} = \mathtt{r(Y2r)}, \mathtt{w(X2w)} = \mathtt{r(R2r)}$$

respectively. However, no linkage can be used to establish the desired result. Consider for example the linkage

```
r(X1r)=w(X2w),r(Y2r)=w(Y1w)}
r(R1r)=w(R1w), r(R2r)=w(R2w)
```

This succeeds in establishing `r(R1r)=r(R2r)` (all the variables are equated to each other). But this is vacuous – every valuation is a solution. Hence this linkage must be rejected. □

1.5 Comparison with related work

The use of constraints distinguishes our approach from the models of Manson/Pugh [MP04] and Sarita Adve [Adv04]. We believe that ideas in this paper can be used to considerably simplify their models.

At a high-level, both proposals are motivated by the desire to specify a semantics which will permit "reasonable compiler transformations". They choose to tackle the problem of out-of-order writes by requiring that a proposed execution trace be given first. (For example the proposed trace may contain out of order executions of program statements.) This trace is then examined for out

of order writes and explicit additional conditions on traces are given to permit certain kinds of writes and rule out others.

For instance, the Manson/Pugh model requires that there be a total order over actions called a *justification order*. The value returned by a read on a location must be the value written into that location by a prior write event (in the justification order). The conditions on justification orders require another complex notion, i.e. the set of forbidden prefixes. Adve's proposal (the SC-model) requires that the out of order writes be justified by another execution of the system which must be "similar" to the current execution in many ways but which should have enough information content to justify the out of order writes.

We believe that the details of these two models are very brittle: seemingly simple variations in the definitions can lead to different results. These ideas appear to be in need of some underlying systematic theory.

In contrast, we believe that constraint-based communication, with the Unique Solutions Criterion, offers a simple, operationally-motivated approach that will appeal to programmers. Both the Manson/Pugh model and the Adve model are built on the happens before relation and thus may be fruitfully compared to the HB Machine. A more detailed technical comparison between our approach and these models must await completion of the technical description of these models (which are still being worked on by their authors).

We believe that an attraction of our conceptual framework is that it places LC and HB on the same footing, making their differences and similarities quite clear.

CRF. Arvind and his colleagues have developed a mechanism-oriented memory model, the "Commit-Reconcile and Fence" approach, designed for architects and compiler writers [SAR99,MAS00]. CRF provides a small language (whose semantics is defined using term-rewriting) into which higher-level parallel languages are intended to be translated. CRF exposes both data replication and instruction reordering. CRF commits to memory coherence. The semantics of JAVA's synchronization constructs is described by translation into CRF.

The CRF approach is similar in spirit to an incomplete step operational semantics discussed in Section 1.3. We believe that it will be very useful to develop a relationship between such an incomplete-step abstract machine and the invariant-oriented abstract machines discussed in this paper. We believe our use of constraints rather than term-rewriting significantly simplifies the technical development. We look forward to a more detailed technical comparison between the two approaches in future work.

Uniform Memory Model. [YGL04] presents an approach to defining memory models using an abstract machine coupled with a transition table for executing instructions. A local instruction buffer at each thread to is used to store pending instructions in "program order". The specification of memory system is defined in terms of a transition system specified using guarded commands.

Entries in a bypass table govern instruction reordering at runtime.

This approach also appears to correspond to an incomplete step semantics. Data structures are explicitly maintained and modified by the operation of the abstract machine.

We believe that our formalization in terms of memory as a store of constraints permits a simpler and more elegant treatement of similar intuitions. Our approach permits multiple memory instructions to be executed in "one step", thus obtaining the effect of instruction reordering.

1.5.1 Rest of this paper

We discuss the design of the Concurrent Constraint-based Memory Machine framework in detail. We instantiate the framework with the LC and HB partial orders to obtain two concrete machines. We discuss properties of these machines. We outline the results obtained with these models for all the test cases of [Pug04b]. We discuss future work.

The presentation in this paper is deliberately informal but rigorous for the most part. The reader wishing to get a basic understanding of this approach should read Section 2 and Appendix A and examine the test cases in detail. The theoretician interested in how action sets get generated from the source program may wish to look at the operational semantics in Section 3.

This is a preliminary draft paper. The full version of this paper will contain statement and proofs of theorems framing key properties of these machines.

2 Concurrent Constraint-based Memory Machines

This section describes the CCM Machine framework in detail. Using the material in this section, Section 3 presents a structural operational semantics for a mini-language sufficient to describe the examples in [Pug04a].

We note that this description is intended for *specification purposes* only. There is no requirement that an actual implementation behave in this way; only that all the behaviors that it generates must be understandable in terms of the operation of such an abstract machine.

Indeed, there is no requirement even that the steps in one execution of the machine must all be taken in hardware or performed at run-time. A compiler is free to perform whatever transformations on the input program it wishes as long as it does not violate the Golden Principle.

Another obvious point worth reiterating is that the abstract machine describes *all* possible behaviors. A concrete implementation is interested only in realizing *one* particular behavior and will thus usually make additional implementational commitments.

In particular we do not anticipate that any realistic implementation of CCM

machines will perform any kind of constraint-based reasoning at run-time.[3] CCM Machines are intended to be used as an abstract model of computation that describes what is accomplished by current hardware designs.

2.1 Constraint system

We describe a constraint system [Sar93,Sar92] that can be used to formalize most of the structure needed for [Pug04a]. This constraint system may be extended in a fairly routine way to accomodate more events, as necessary, for instance, to accomodate more concurrent constructs (e.g. atomic sections, fences etc). It may help the reader to think that the constraint system formalizes that part of the semantics of the Java Virtual Machine that corresponds to interaction between threads and main memory. Section 3 describes a framework for operational semantics for a small programming language, using the constraint system discussed in this section.

We assume some underlying undefined (infinite) sets of *threads* $\mathbf{T}$, *locations* $\mathbf{L}$ and *events* $\mathbf{E}$. $\mathbf{T}$ has no special structure other than a special constant $\mathbf{0} \in \mathbf{T}$ which interpretes the literal `0` that is used as the designation of the "main" thread (see Section 3).

All locations are thought of as existing in shared memory, even those accessed only by a single thread. Locations may be structured. A location may correspond to an object and may have fields (which are other locations). An object value is represented by a *literal* $\{f_1 = k_1, \ldots, f_n = k_n\}$ which should be thought of as saying that field f_i of the object has the value k_i (see Table 5). A location may correspond to an array, whose elements (corresponding to other locations) are accessed through an index. $\mathbf{L}$ is classified into *normal* or *volatile* locations by the predicates $n(\_)$ and $v(\_)$.

We assume that the constraint system respects the natural properties for arrays and objects, namely:

$$X = \{f_1 = e_1, \ldots, f_k = e_n\}.f_i \vdash X = e_i \tag{1}$$

$$X = [e_1, \ldots, e_k][i] \vdash X = e_i \tag{2}$$

$\mathbf{E}$ has a strict partial order (i.e. a binary relation that is transitive, asymmetric and irreflexive) which is the interpretation of the predicate $\_ \ll \_$. Each event is associated with exactly one thread T and may be associated with zero or more locations. $\mathbf{E}$ supports the following kinds of events:

read such events E satisfy $r(E, T, L)$. The value returned by the read may be accessed through the term $r(E)$.

write such events E satisfy $w(E, T, L)$. The value written into the location may be accessed through the term $w(E)$.

[3]Note, however, that in the future work section we describe some extensions to the underlying programming language that can fruitfully exploit the architecture of CCM Machines. Such extensions may well require some representation of run-time constraint-solving.

(Literals)	`k`	::=	`0` | `1` | ...	(Integer literal)
			`null`	(Null reference)
			{`f=k, ..., f=k`}	(Object literal)
			`[k, ...,k]`	(Array literal)
(Event Constraints)	n	::=	`r(X,T,L)`	(Read)
			`w(X,T,L)`	(Write)
			`l(X,T,L)`	(Lock)
			`u(X,T,L)`	(Unlock)
			`o(X,T)`	(Object creation)
			`a(X,T)`	(Array creation)
			`i(X,T)`	(Thread Create)
			`x(X,T)`	(Thread Destroy)
(Ordering Constraints)	o	::=	`E` $\ll$ `E`	
(Read Terms)	r	::=	`r(E)`	
			`k`	(Literals)
			`r` + `r` | `r` $\times$ `r` | ...	(Arithmetic terms)
(Write Terms)	w	::=	`w(E)` | `d?w:w`	
(Write Constraints)	q	::=	`w(E)=r`	
(Linking Constraints)	l	::=	`r(E)=w`	
(Pending Conditions)	d	::=	`r` $>$ `r` | `r` $<$ `r`	
			`r!=r` | ...	(Comparisons)
			!(`d`)	(Negation)
			`d` && `d`	(Conjunction)
			`d` || `d`	(Disjunction)
(Loc Constraints)	x	::=	`n(L)`	(Normal Location)
			`v(L)`	(Volatile Location)
			`l(E)=l(E)`	(Location Equality)
(Action set)	a	::=	x | n | o	
			q | z,z	
			$d \rightarrow a$	(Conditional)
(Store)	s,z	::=	x | n | o	
			l | q	
			d	
			$d \rightarrow s$	(Conditional)
			z,z	(Conjunction)
(Field Selector)	f	::=	⟨ident⟩	
(Event Vars)	X,E	::=	⟨ident⟩	
(Thread Vars)	T	::=	`0` | ⟨ident⟩	
(Locations)	l	::=	⟨ident⟩	(Static location)
			r(E)	(Dynamic location)
			l.f	(Object field)
			l[r]	(Array location)

Table 4: The constraint system C_J for JAVA

lock such events E satisfy $l(E, T, L)$. There is no value associated with this event.

unlock such events E satisfy $u(E, T, L)$. There is no value associated with this event.

array creation such events E satisfy $a(E, T)$. The array created by this event may be accessed through the term $r(E)$.

object creation such events E satisfy $o(E, T)$. The object created by this event may be accessed through the term $r(E)$.

thread creation such events E satisfy $i(E, T)$. There is no value associated with this event. T should be a new variable, and may now be used as a Thread Var.

thread destruction such events E satisfy $x(E, T)$. There is no value associated with this event. There should be no further events (in the ordering) generated by thread T.

For each event E, the associated thread may be accessed through the term $t(E)$, and the associated location, if any, through the term $l(E)$. All the classifiers $r(E, T, L), w(E, T, L), \ldots$ enumerated above classify disjoint sets of events. Thus, for example, the conjunction of assertions $w(E, T, L), r(E, T', L')$ is inconsistent, the conjunction of assertions $w(E, T_1, L_1), w(E, T_2, L_2)$ implies $T_1 = T_2$ and $L_1 = L_2$ etc.

An event associated with thread T and a single location L is said to be targeted at the *memory line* $m(T, L)$. Some events (e.g. lock and unlock on normal variables, reads and writes on volatile variables) are considered *synchronization* events (and are treated specially by the underlying Order Model).

The set of terms $r(E)$ for E a read event, closed up under the given primitive operations, is called the set of *read terms*. We shall often say that $r(E)$ is a *read variable* and $w(E)$ is a *write variable* and call such variables *program variables*.

Arithmetic operators are interpreted in the standard way. The interpretation of equality is also standard. The conditional expression is used to model conditional assigments. The expression $d\,?\,t_1 : t_2$ stands for the expression t_1 if d has the value `true` and for t_2 otherwise.

Logical connectives are interpreted as in standard classical logic. One may give a semantics to constraints in terms of sets of valuations that *realize* the constraint.

Definition 2.1 (Valuation, Satisfaction) A *valuation* is a mapping of variables to values in the underlying domain of interpretation (i.e. **E** for events, **L** for locations, the integers for integer program variables etc). A valuation *realizes* a constraint if the constraint is true under this mapping. A set of constraints s *entails* a constraint d, written $s \vdash d$, if every valuation that realizes s realizes d. □

An *inconsistent* constraint is one which cannot be realized by any valuation. As is standard in logic, an inconsistent constraint entails every constraint.

2.2 Configuration

The state of a CM machine is defined by $k \geq 0$ *threads*, and a *store*.

2.2.1 Thread

A thread contains all the private state necessary to execute a thread of control. In particular it has space for private local variables and for a stack. We assume that a thread is capable of performing arithmetic operations, branching, method invocations etc, much as a thread of the current Java Virtual Machine (JVM).

The thread communicates with the shared store through a collection of reads and (conditional) write operations, via an *action set* as described below. Conceptually, the result of communicating an action set to memory is that the store is changed (because writes are committed), and values are obtained (through read operations) into private variables local to the thread. Based on these values, the thread may perform certain expression evaluations, decide the value of predicates, take some branches, throw an exception, invoke a method etc. Subsequently it may again interact with memory by communicating another action set, wait for (some or all of) the results before continuing etc.

We will distinguish a set of variables used in an action set for communication between thread and main memory. These variables are single-assignment variables (sometimes called *logical variables*). These variables will be created in the private store of a thread but references to them may be passed to main memory. As we shall see below, main memory will assign values to these variables by imposing constraints on them which reflect how read events are matched to write events. These values are communicated to the thread and used by it to direct its operations. In what follows, we shall use the term *variable* for logical variables, and use the term *locations* for normal (Java) program variables.

In addition to this, a thread may have several assignable registers for its own internal book-keeping. These do not participate in communications with main memory and are hence not relevant to our model.

Strictly speaking the CCM Machine framework does not require that a thread be implemented sequentially, and hence the activity of communicating with the store be sequenced with the activity of making local decisions. Indeed a thread may be implemented as a collection of much finer-grained activities, some of which are responsible for manipulating the stack, some for communicating with main memory etc. But all these are issues of implementation. Here we are concerned with describing the overall logical picture.

2.2.2 Store

The set of events defined by a particular CM Machine depends on the underlying Order Model (Section 2.4).

Definition 2.2 (Action set) An *action set* is a set a of constraints in the constraint system of Table 4 satisfying the Ordering Invariants for action sets of the underlying Order Model. □

Store. A store is an action set with additional *linking* constraints. Before defining these constraints, we need to introduce the notion of a *write term*.

Given that an action set may contain conditional writes, it must be possible for the memory to specify a linking between reads and writes that is also conditional. For instance, suppose the store contains two write events for a location x, with associated write variables `Y` and `Z`. Suppose the first is a conditional write, with constraint `c`. Then it should be possible for the store to specify that a read should be answered from location `Y` if the condition `c` is true, and from location `Z` otherwise. This motivates the following definition.

Definition 2.3 (Write term) A *write term for location* L is the term $w(E)$ for any event E on location L or the term $d\,?\,w_1 : w_2$, where d is a pending condition and w_1, w_2 are write terms for location L. □

Definition 2.4 (Store) A *store* is a set of constraints containing an action set and *linking* constraints, which are of the form $r(E) = w$ for a read event E for location L and a write term w for L. □

A linking constraint is added to the store by Main Memory after it has made a particular choice of write events to use to respond to a read. Typically a linking constraint relates a read variable to a single write variable (thus establishing a direct linking).

Since we allow conditional events, we must have a way of determining when an event is *active*. Intuitively, an event is active if its associated condition (if any) is satisfied.

Definition 2.5 (Active event) We say that a read event E is *active* in a store s if $s \vdash r(E, T, L)$ for some T, L. Similarly for write events. □

By the remark above, every event is active in an inconsistent store.

Definition 2.6 (Extension of a store) An *extension* of a store s is any set augmenting s with equations of the form $r(E) = k$, for k a literal. An extension is *total* if it forces every active read event to have a unique value. □

Example 2.1 Let s be the set of constraints:

$$\begin{array}{l}\{\ w(Ix, \mathbf{0}, x), w(Ix) = 0, r(X1r, 1, x), Ix \ll X1r, \\ \quad r(X1r) > 0 \ \rightarrow\ (w(Y1w, 1, y), w(Y1w) = 1, X1r \ll Y1w) \\ \quad r(X1r) \leq 0 \ \rightarrow\ (w(Y1w1, 1, y), w(Y1w1) = 0, X1r \ll Y1w1)\}\end{array}$$

Intuitively it represents the events generated by the program:

```
init x=0;
if (x>0) {
 y=1;
}
if (x=<0) {
 y=0;
}
```

The event $Y1w$ is not active in s because s does not entail $w(Y1w, 1, y)$. However, the extension $s' = s \cup \{r(X1r) = 1\}$ of s does entail $w(Y1w, 1, y)$ (as can be established through standard logical reasoning). But s' does not entail $w(Y1w1, 1, y)$. □

For any store s, let $o(s)$ be the set of all event constraints and order constraints entailed by s. We can use the underlying Order Model to answer questions about visibility of a write event at a read event in $o(s)$. However, we must deal with conditional events, and hence need to extend the notion of visibility to write terms.

Definition 2.7 (Visibility of a write term) Given a store s, a read event E and write term w for location L, w is said to be *visible* to E

- if $w \equiv w(E')$, and for every consistent extension s' of s, E' is visible to E in $o(s')$ (according to the underlying Order Model).
- if $w \equiv d\,?\,w_1 : w_2$, and w_1 is visible to E in every consistent extension of s that entails d and w_2 is visible to E in every consistent extension of s that entails $!(d)$.

□

We say that a set of linkings l is *valid* in a store s if for every link $r(E) = w \in l$, w is visible to E in s.

Proposition 2.1 *Let s be a store and l a set of linkings valid in s. Then l is valid in $s \cup l$.*

Let $r(E) = w \in l$. We have to show that w is visible to E in $s \cup l$. Let $w \equiv w(E')$. Let s' be an extension of $s \cup l$. Then there must be an extension s'' of s such that $o(s') = o(s'')$. But $w(E')$ is visible to E in $o(s')$. Hence it is visible to E in $o(s'')$. It follows that $w(E')$ is visible to E in $s \cup l$. A similar argument works in the other case.

The definition above has been chosen carefully. As we see below (Definition 2.10), memory is required to add linkings that are valid in the union of the store and action set (together with forced synchronization and ordering constraints). This condition ensures that each constraint `r(E)=w` will link `E` to a write event which is visible to it, *regardless* of other linkings introduced for other events. Section A.5.1 discusses a situation in which a linking valid in the new store is in fact not valid in the union of the store and the action set, and is hence ruled out by Definition 2.10, as desired.

Definition 2.8 (Events associated with a store) Let s be a store. The set $e(s)$ of *events associated with* s is defined as the set of all events E such that some consistent extension of s entails an event constraint for E. □

Example 2.2 For the store s of Example 2.1, $e(s) = \{Ix, X1r, Y1, Y1w\}$. □

Definition 2.9 (Valid store) A store s is said to be *valid* if

1. the store satisfies *the Ordering Invariant for stores* of the underlying Order Model,
2. every linking in s is valid in s, and
3. s forces $r(E)$ to have a unique value for every active read event E in s (*Unique Solution Criterion*).
4. if $s \vdash w(E) = r$ and some read term $r(E)$ occurs in r, then $s \vdash r(E, T, L)$ for some T, L (*Read Closure*).
5. for every condition d and event n, if $s \vdash d \rightarrow n$ then for every term $r(E)$ occurring in d and every event E' in n, $s \vdash E \ll E'$.

□

Recall that every inconsistent store satisfied every valuation. Therefore for non-empty valid stores (with at least one program variable), the Unique Solution Criterion requires that the order constraints in s are satisfiable, i.e. there are no cycles etc.

Proposition 2.2 (Empty store) *The empty set is a valid store.*

Execution of a CCM Machine begins in the empty store.

2.3 Transition

In this section, we informally review the transitions made by a CCM machine. Appendix 3 provides an operational semantics that formalizes these ideas.

The CM Machine moves from one state to another as the result of two kinds of moves.

2.3.1 Silent transitions

Based on its control state, and data received into private (logical) variables from the shared memory, a thread may make a private transition (affecting only its local state) to another state. For instance it may perform arithmetic operations, take a branch, thrown an exception, allocate a new object on the heap etc. This change of state is entirely unobservable by the shared store or by any other thread.

2.3.2 Shared transitions

The LC machine may also make a step based on a simultaneous interaction between $k \geq 1$ threads and the store s.

In such a step, each thread T_i participating in that step communicates an action set S_i to the store obtained from the next set of statements to be executed

by the thread. The size of the action set is entirely up to the thread. The action set may contain a single read event or a single write event or dozens of such events. Roughly speaking, the size of this set is correlated to the amount of prescient computation permitted by the thread.

Similarly it is not *necessary* that more than one thread participate in a shared transition; it is merely *permitted* for them to do so. Indeed a system execution in which only one thread participates in a shared transition, and performs only one event per transition is completely legal and corresponds to a sequentially consistent execution of the program.

We now define how to obtain in one step a new store from an existing store and an input action set.

Definition 2.10 (New store) Assume a fixed underlying Order Model O. Let s be a valid store (for O) and z an action set such that $e(s)$ and $e(z)$ are disjoint. We say that s' is a *new store for s on input a* (for Order Model O) and write $s, a \rhd_O s'$ provided that $s' = s \cup a \cup o_h \cup o_i \cup l$ is a valid store (for O) where:

1. o_h is a set of constraints $E_1 \ll E_2$ where $E_1 \in e(s)$ and $E_2 \in e(a)$ are defined on the same thread (*sequencing constraints*),
2. o_i is a set of constraints $E_1 \ll E_2$ for E_1, E_2 synchronization events in $e(a)$ on different threads, (*synchronization constraints*)
3. l is a valid linking in $s \cup a \cup o_h \cup o_i$ (*linking constraints*).

Above, the sets o_h and o_i should be chosen to be *minimal*, that is, they should add only those order relations that need to be added in order to obtain a valid store. (That is, we require that there be no subsets o'_h of o_h and o'_i of o_i such that $s \cup a \cup o'_h \cup o'_i \cup l$ satisfies all the conditions above.)

We say $s \longrightarrow s'$ if there is some action set a such that $s, z \rhd s'$. □

For the three kinds of machines we consider o_h is determined uniquely by s and z – there is no choice. The synchronization constraints capture ordering constraints introduced to resolve competing synchronized accesses to shared variables; as such there can be zero, one or more choices of o_i for a given s, a and o_h.[4]

With s, a, o_h, o_i fixed, there may still be many choices of linkings l – but only if there are data races. See Section 2.6 where we argue informally that if there are no data races then in fact the choice of linking is forced and corresponds to what would be returned by a sequentially consistent execution respecting the given order constraints.

These values are communicated to the appropriate threads, which may now resume execution, and repeat the cycle.

This concludes a description of the action of the basic abstract machine.

[4]There may be zero choices if a offers events that cannot be performed in the current state, e.g. a thread T may offer a *lock* operation on a variable x, but in s the lock is held by some other thread T' and a does not contain an *unlock* event on $m(T, x)$. It is up to the underlying Order Model to specify conditions on input action sets such that if these conditions are satisfied then it is always possible to obtain a new store from the current store.

Propositions. The following propositions are true for all CCMs.

Proposition 2.3 (Accumulativity of store) *Suppose s, s' are valid stores such that $s \longrightarrow s'$. Then $s' \vdash s$.*

Thus the store is accumulative. That is, given an execution sequence $s_0 \longrightarrow s_1 \ldots \longrightarrow s_n$, we have $s_n \vdash s_i$ for all i.

Proposition 2.4 (Conservativity of order relations.) *Suppose s, s' are valid stores such that $s \longrightarrow s'$. Then $s' \vdash E \ll E'$ for $E, E' \in e(s)$ iff $s \vdash E \ll E'$.*

The proposition says that the new store is conservative over old order relations: it induces no new order relation between old event variables. This follows from the fact that the new order relations in s' relate to the new events in s' (o_i) or relate an old event to a new event (o_h). But these cannot cause a new order relation to be induced on the old variables. Finally, because of USC all the conditions in s are discharged so there is no possibility of new information on program variables (as could potentially be induced by the new linking) discharging an old conditional, thereby establishing a new order relation between old variables.

Proposition 2.5 (Conservativity of linkings.) *Suppose l is a valid linking in s and $s \longrightarrow s'$. Then l is a valid linking in s'.*

This follows from Proposition 2.4. The new information cannot add any new order relation on old variables, hence cannot affect the visibility of write terms to read events.

At each step, the store resolves enough of the non-determinism in the events offered by participating threads to determine the order of synchronization operations, and specify the precise value returned by each read operation.

2.4 Order Models

We now turn to defining a few representative order models. Each of these order models O gives rise to a specific abstract machine CCM(O).

The following definitions will be useful below.

Definition 2.11 (*Event(L)*, etc.) Let L be any location and T any thread. We define the following sets:

- *Event(L)* is the set of all event constraints on L.
- *Synch(L)* is the set of all synchronization event constraints on L.
- *Thread(T)* is the set of all event constraints whose thread is T.
- $m(T, L)$ is the set of all event constraints whose thread is T and location is L. Note that $m(T, L) =$ *Thread(T)* $\cap$ *Event(L)*.

Let s be a set constraints, and O a set of event constraints. Then the *restriction of s to O*, written $s \downarrow O$ is defined to be the set

$$(s \cap O) \cup \{s \vdash E \ll E' \mid E, E' \in s \cap O\}$$

□

Definition 2.12 (Totally ordered) Let s be a store. s is said to be *totally ordered* if for every consistent total extension s' of s, and every pair of events $E, E' \in e(s')$ either $s' \vdash E \ll E'$ or $s' \vdash E' \ll E$. □

(See Definition 2.6.)

Definition 2.13 (Lock condition for action sets) An action set s satisfies *the lock condition* for a location L if for all threads T, and extensions s' of s, $o(s') \downarrow (Synch(L) \cap Thread(T))$ is a total order in which every *lock* (*unlock*) event is followed (if at all) by an *unlock* (*lock*) event E', and for every *lock* event E followed by an *unlock* event E', $t(E) = t(E')$. □

Definition 2.14 (Lock condition for stores) A valid store s satisfies *the lock condition* for a location L if $o(s) \downarrow Synch(L)$ is a total order, whose minimal event (if any) is a *lock* event, and in which every *lock* (*unlock*) event is followed (if at all) by an *unlock* (*lock*) event E', and for every *lock* event E followed by an *unlock* event E', $t(E) = t(E')$. □

Definition 2.15 (Proper Initialization) A valid store s is said to be *properly initialized* if for every location L such that s entails an event constraint on L, s entails a minimal event constraint on L, and this event constraint is a write event. □

2.4.1 The Sequential Consistency Model

Definition 2.16 (Ordering Invariant for Action sets) An action set s must satisfy the conditions:

SC-A1: for all threads T and total extensions s' of s, $s' \downarrow Thread(T)$ is totally ordered, and,

SC-A2: the lock condition for action sets for all normal locations L.

□

Definition 2.17 (Ordering Invariant for Stores) A store s must satisfy the conditions:

SC-S1: it is totally ordered,

SC-S2: it satisfies the locking condition for all normal locations, and,

SC-S3: it is properly initialized.

□

No conditions are necessary for volatile variables. Given a store s, we say that thread T *holds the lock on location* L if the maximal event in $s \downarrow$ *Synch(L)* is a *lock* event with thread index T.

For SC, write visibility can be defined directly: a read event sees only the last write event on that location. However, we take this opportunity to present a much more general definition, which will work for other order models as well.

Definition 2.18 (Write visibility) In a store s, each read event r on a line $m(l,t)$ can see a write event w on a line $m(l,t')$ (t may or may not equal t') if w is a maximal element in the set of all write events that occur $\ll$-before r in s or w is not $\ll$-related to r. □

Because the set of all events is totally ordered, this definition gives the same results as the one above.

We now consider the work that needs to be done to move to a new store from a given store s and an action set a. First, o_h must be chosen to force for every thread T every event of T in a to be ordered after every event of T in s. Second, o_i must be chosen to force a total ordering on all events in $s \cup a$ merging the orderings for each thread in a and satisfying the lock condition. Note that the lock condition may not always be satisfiable. Given $s \cup a \cup o_h \cup o_i$ it is not hard to see that there can be at most one valid linking l such that $s \cup a \cup o_h \cup o_i \cup l$ is a valid store.

2.4.2 The LC Order Model

Location consistency requires all events on a memory line are (conditionally) totally ordered. Additionally, *lock* and *unlock* events introduce a total order across memory lines for the same location.

Definition 2.19 (Ordering Invariants for Action Sets) An action set s must satisfy the conditions:

LC-A1: for every memory line $m(T,L)$ and total extensions s' of s, $s' \downarrow m(T,L)$ is totally ordered, and,

LC-A2: the lock condition for action sets for all normal locations L.

□

Definition 2.20 (Ordering Criterion for Stores) An store s must satisfy the conditions:

LC-S1: $s \downarrow m(T,L)$ is totally ordered, for every memory line $m(T,L)$,

LC-S2: it satisfies the locking condition for all normal locations, and,

LC-S3: it is properly initialized.

□

The definition of Write Visibility is the same as for SC.

We now consider the work that needs to be done to move to a new store from a given store s and an action set z. First, o_h must be chosen to force for every memory line $m(T, L)$ every event in z to be ordered after every event of T in s. Second, o_i must be chosen to satisfy the lock condition. Note that the lock condition may not always be satisfiable. Also, because of race conditions and the lack of a total order more than one write event may be visible at a read event. Therefore multiple linkings may be possible.

2.4.3 The HB Order Model

This definition is drawn from [Pug04a, P.17].

Definition 2.21 (Ordering Invariants for Action Sets) An action set s must satisfy the conditions:

HB-A1: for all threads T and total extensions s' of s, $s' \downarrow$ *Thread(T)* is totally ordered, and,

HB-A2: the lock condition for action sets for all normal locations L.

□

Definition 2.22 (Ordering Criterion for Stores) An store s must satisfy the conditions:

HB-S1: $s \downarrow$ *Thread(T)* is totally ordered, for every thread T,

HB-S2: it satisfies the locking condition for all normal locations,

HB-S2v: for all volatile locations V, if $s \vdash r(E') = w(E)$ for two events E, E' for V, then $s \vdash E \ll E'$

HB-S3: it is properly initialized.

□

The condition for write visibility is the same as for LC.

From this the following proposition is easy to establish:

Proposition 2.6 *Let s be a store satisfying the HB Order Model and containing no synchronization events. Then s satisfies the LC Order Model. Further, for every read event in s the set of visible write events is the same according to both Order Models.*

We now consider the work that needs to be done to move to a new store from a given store s and an action set z. First, o_h must be chosen as for sequential consistency. Second, o_i must be chosen to satisfy the lock condition. Note that the lock condition may not always be satisfiable. Third, an order relation must be introduced between every write on a volatile variable and a read which is answered by that write.

2.5 Examples

Example 2.3 (Table 1 program) Consider a run of the program in Table 1 for the LC machine. Initially the program consists of Thread 1, Thread 2 and an initial store z_0 with memory lines

$$m(1, r2), m(1, a), m(1, b), m(2, r1), m(2, a), m(2, b)$$

and with initial events

$$\{\mathtt{w(Ai, 0, a)}, \mathtt{w(Ai) = 0}, \mathtt{w(Bi, 0, b)}, \mathtt{w(Bi) = 0}, \mathtt{n(a)}, \mathtt{n(b)}\} \tag{3}$$

We read the set thus. `a` and `b` are normal locations. There is a write event `w(Ai,0,a)` on memory line `m(0, a)`, with associated variable `Ai`. The constraint associated with the write is `Ai=0`. Similarly for the other events. Note that there is no order relation between the events, and the linking is empty (there are no read events).

In one step, Thread 1 may communicate the following action set z_1 to the store:

$$\{\mathtt{r(A1r, 1, a)}, \mathtt{w(R21w, 1, r2)}, \mathtt{w(B1w, 1, b)}, \mathtt{w(R21w) = r(A1r)}, \mathtt{w(B1w) = 1}, \mathtt{n(R2)}\} \tag{4}$$

Since the action set does not contain two events on the same memory line, no ordering information is needed.

At the same time, Thread 2 may communicate the following action set z_2:

$$\{\mathtt{r(B2r, 2, b)}, \mathtt{w(R12w, 2, r1)}, \mathtt{w(A2w, 2, a)}, \mathtt{w(R12w) = r(B2r)}, \mathtt{w(A2w) = 2}, \mathtt{n(R1)}\} \tag{5}$$

As above, this set contains no ordering information.

The new store will contain all these events. Additionally, memory will add ordering constraints o_h to ensure that all actions by each thread are after the actions already recorded on that memory line.

$$\{\mathtt{Ai} \ll \mathtt{A1r}, \mathtt{Bi} \ll \mathtt{B1w}, \mathtt{Bi} \ll \mathtt{B2r}, \mathtt{Ai} \ll \mathtt{A2w}\} \tag{6}$$

Since there are no synchronization events, no synchronization constraints need to be added.

Finally, the store must add a linking to answer the reads communicated to it. For instance one linking l_1 is:

$$\{\mathtt{r(A1r) = w(A2w)}, \mathtt{r(B2r) = w(B1w)}\} \tag{7}$$

As can be verified, the set of constraints $\sigma_1 = z_0 \cup z_1 \cup z_2 \cup o_h \cup l_1$ forces a unique solution for program variables: every read variable has a concrete value, and only one value is possible. In this store, the "current value" of the memory line $\mathtt{m}(\mathtt{1}, \mathtt{r2})$ is $\mathtt{2}$ and of $\mathtt{m}(\mathtt{2}, \mathtt{r1})$ is $\mathtt{1}$.

Another possible linking l_2 is:

$$\{\mathtt{r}(\mathtt{A1r}) = \mathtt{w}(\mathtt{A2w}), \mathtt{r}(\mathtt{B2r}) = \mathtt{w}(\mathtt{B2i})\} \tag{8}$$

Again, the set of constraints $\sigma_2 = z_0 \cup z_1 \cup z_2 \cup o_h \cup s_2$ is uniquely satisfiable, and provides current values of $\mathtt{2}$ for $\mathtt{m}(\mathtt{1}, \mathtt{r2})$ and $\mathtt{0}$ for $\mathtt{m}(\mathtt{2}, \mathtt{r1})$.

Two other stores are possible, corresponding to the two other linkings for the two reads.

Figure 2 depicts the stores S_0, σ_1 and σ_2 using an evident graphical notation. □

Appendix A lists all the test cases in [Pug04b] and [Pug04a] and discusses their results for CCM(HB).

2.6 Properties satisfied by the LC model

No thin air reads. This property requires that all reads of a variable must return a value that was communicated by some thread to memory in a write event on that variable, and this write event must be visible to the read event, per the ordering rules.

This property is structurally guaranteed by CCMs because they can only establish "variable-variable" value constraints, linking a read event to a write event to the same location. Such linkings may be thought of as introducing "flows" in a graph, not supplying sources or sinks. The "reasoning" performed by the constraint system to flow values throug the graph is merely reasoning that captures the logical properties of implications and conjunction.

All reads are answered by writes to the same location. Because a linking must satisfy the Unique Solutions Criterion, each read returns a concrete value.

In particular, this implies that Type Safety is preserved.

Adding threads does not invalidate executions. More threads can only lead to more writes and hence more ways of answering reads. Previous ways of answering reads continue to be valid.

Specifically, CCM Machines satisfy *linking monotonicity*. Suppose a CCM machine produces a set of linking constraints l in a valid store s in response to valid input action sets $s_0, \ldots, s_{k-1}$ received from threads $z_0, \ldots, z_{k-1}$. Let s_k be a new action set received from thread z_k such that $s_0, \ldots, s_{k-1}, s_k$ is a valid input action set for s. Then there must be a new store for $s, s_0, \ldots, s_{k-1}, s_k$ that contains l.

Instrumentation reads are benign. Any linking established by memory cannot be invalidated because of the presence of more reads. Therefore if the

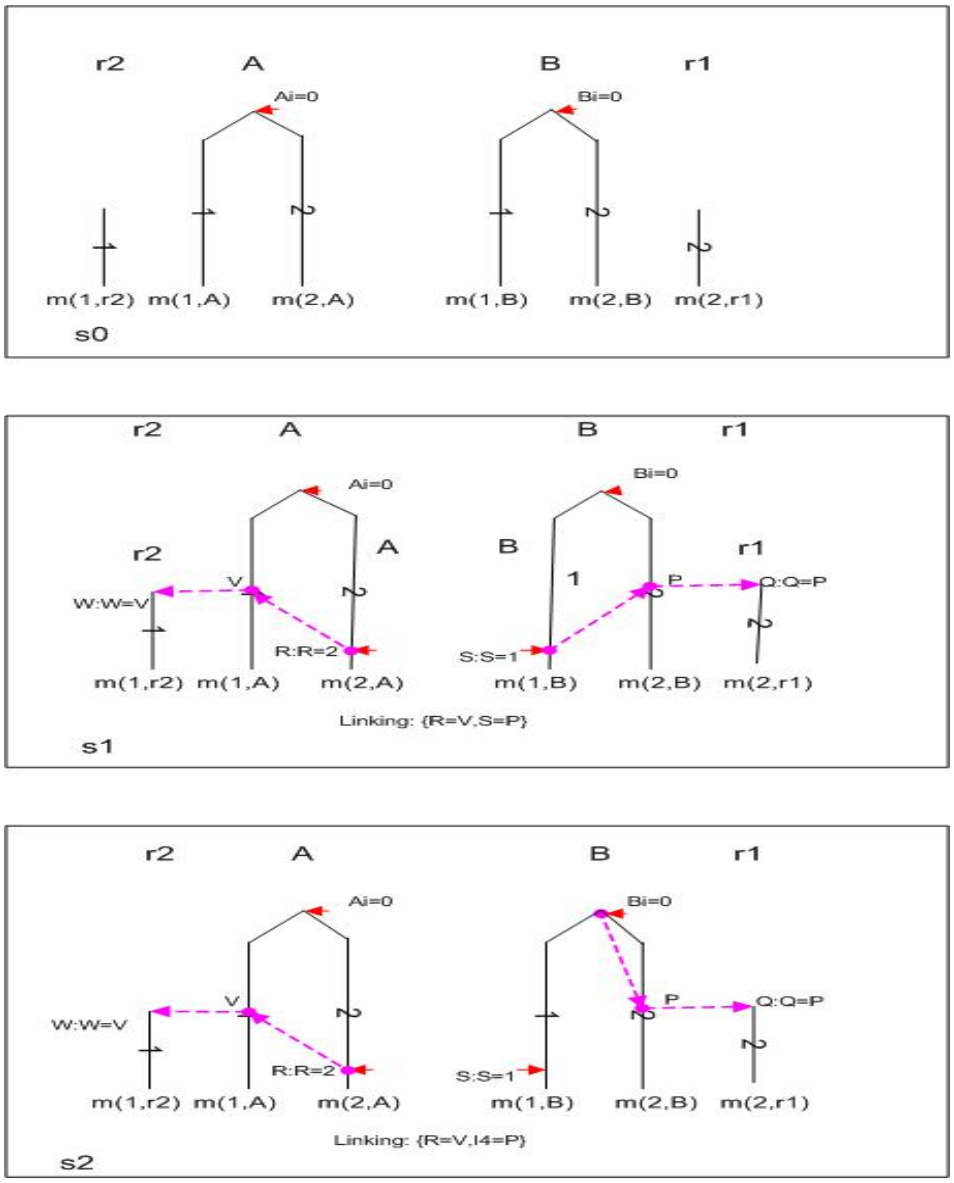


Figure 2: The evolution of the store for the program in Table 1

program is modified only by adding more reads all previous executions continue to be valid. This is true even if an originally properly-synchronized program now contains data-races.

Properly synchronized programs have sequentially consistent executions. The outline of the proof of this proposition for CCM(HB) is as follows. Similar considerations apply to other machines.

Let s be a valid store reached at some stage of the execution, a an input action set, and o_h and o_i sequencing and synchronization constraints respectively. Let l be a valid linking such that for $s' = s \cup a \cup o_h \cup o_i \cup l$, we have $s, a \rhd s'$. We observe that if there are no data races, then all valid linkings are equivalent, that is, for any other store $s'' \supseteq s \cup a \cup o_h \cup o_i$ such that $s, a \rhd s''$, we have $s' \vdash r(E) = k$ iff $s'' \vdash r(E) = k$. Further, we observe that if s forces a read event E to be answered by a write event E' then $s \vdash E' \ll E$. (Why? Because proper initialization ensures that there is always such a dominated write that can be used to answer a read in every valid store. If there are no data races then there are no other writes that can be visible at the read in s', which has resolved all the conditionals.) Next we observe that any scheduling (= total order) of the the events in the program that respects the $\ll$ relation in s' will return the values for the reads determined by s' and corresponds to a sequentially consistent execution of the program.

Compiler optimizations. We consider some compiler optimizations that are sound for CCMs. In particular we consider CCM(LC).

Two reads to the same mutable global variable by the same thread with no intervening writes to that variable may always be replaced by a single read into a register and reuse of that register.

3 Operational Semantics

The reader interested in examples may wish to skip this section (which is more formal in nature) on the first reading and proceed to Appendix A.

In this section we present a framework for the operational semantics of a simple, toy language CJ, parametrized by the underlying Order Model. A choice of an Order Model yields an operational semantics for CJ. The language is intended to illustrate how to use CCMs in a concrete context and is not intended for serious programming.

The syntax of the language is given in Table 5. It deals with constructs necessary for JSR 133. To deal with a fuller language, one needs to add classes, methods, dynamic object creation, method invocation, exceptions etc. These are concerned primarily with local thread execution and are hence not central to the concerns of this paper.

(Expressions)	e	*::=*	L	*(Variables)*
			k	*(Literals)*
			e + e | e × e | ...	*(Arithmetic terms)*
			e.f	*(Field selection)*
			e[e]	*(Array selection)*
(Conditions)	c	*::=*	e > e | e < e | e! =e | ...	*(Comparisons)*
			!(c)	*(Negation)*
			c && c	*(Conjunction)*
			c || c	*(Disjunction)*
(Statements)	S	*::=*	init L=k;	*(Normal Var Declaration)*
			volatile L=k;	*(Volatile Var Declaration)*
			L=e;	*(Assignment)*
			lock(L);	*(Locking)*
			unlock(L);	*(Unlocking)*
			S1 S2	*(Sequencing)*
			if (c) {S} [else {S}]	*(Conditionals)*
			d → S [: S]	*(Pending Conditionals)*
			while (c) {S}	*(Iteration)*
			do {S} while (c)	*(Iteration)*
			P;	*(Parallelism)*
(Command)	P	*::=*	thread {S}	*(Thread)*
			thread(T) {S}	*(Active Thread)*
			P | P	*(Parallel Composition)*
(Loc Vars)	L	*::=*	⟨*ident*⟩	*(Static Location)*
			L.f	*(Object field)*
			L[e]	*(Array element)*

The categories *Literals* (k), *Pending Conditions* (d), *Locations*(l) and *Thread Vars*(T) are defined in Table 4.
Input programs may not have any occurrences of pending conditionals, active threads or use the thread variable 0. These are generated by the operational semantics.

Table 5: Syntax for a mini-language, CJ

3.1 Informal semantics

Variables in this programming language denote locations in shared memory. Locations may be declared normal or volatile. Accesses to volatile locations are considered synchronized, and follow the rules of the underlying Order Model. Access to normal variables is unsynchronized. Each location is associated with a *lock* which permits locking and unlocking operations for synchronization.

Locations may contain integers, or references to objects and arrays. Objects have one or more fields, each of which is a location. Arrays are indexed using arithmetic expressions and are intended to be of fixed size, as determined by the literal used to initialize the location. Object fields and array locations may be assigned values.

The language is untyped, for simplicity of exposition.

A few exemplar control constructs are provided: assignment, sequencing, conditionals (single and two-armed), while loops.

Threads may be created dynamically and run in parallel with each other.

The syntax permits two control constructs, pending conditionals and active threads, which are not allowed to occur in source programs. These constructs are introduced by the operational semantics as part of the elaboration of conditionals and threads respectively.

3.2 Formalization of operational semantics

The semantics is defined in the usual structural operational semantics style due to Plotkin. We take a *configuration* to be an ordered pair consisting of a statement to be executed and the store in which it is to be executed, or just a store (the statement has terminated). The transition relation $\longrightarrow$ is a binary relation on configuration which specifies how one configuration evolves into another.

Definition 3.1 ($\mathbf{var}(s)$) For s a set of constraints, let $\mathbf{var}(s)$ stand for the set of variables occurring in a constraint in s. □

The operational semantics we present is parameterized by two relations, $\lhd$ and $\rhd$. These are defined differently based on the underlying Order Model. We now defined $\lhd$:

Definition 3.2 (Ordered union of constraints) Let s and s' be two sets of constraints. The *ordered union* of two sets of constraints is defined thus:

$$s \lhd_{SC} s' \stackrel{def}{=} s \cup s' \cup \{E \ll E' \mid E \in e(s), E' \in e(s')\} \tag{9}$$

$$s \lhd_{HB} s' \stackrel{def}{=} s \lhd_{SC} s' \tag{10}$$

$$s \lhd_{LC} s' \stackrel{def}{=} s \cup s' \cup \{l(E) = l(E') \rightarrow E \ll E' \mid E \in e(s), E' \in e(s')\} \tag{11}$$

□

Note that the equality of two locations may depend on runtime information (e.g. the location denoted by `A[r]` depends on the runtime value of `r`); hence for LC we must generate constraints that check the equality of locations at runtime.

The inference rules for $\longrightarrow$ uses a few auxiliary relations which we define first.

3.3 Expression, condition and location elaboration.

We define the relation "Thread T, in the presence of the store z, can elaborate the expression e into the read term r, extending the action set s to s'", written as:

$$T, z \vdash \langle e, s\rangle \Longrightarrow \langle r, s'\rangle$$

In this rule and the others to follow, we use the generic ordered union, $\lhd$. We should understand this to mean that to obtain the corresponding transition for the O-machine (for O equals HB, SC or LC), we should take $\lhd_O$ for $\lhd$.

$$\frac{E \notin \mathbf{var}(s)}{T, z \vdash \langle x, s\rangle \Longrightarrow \langle r(E), s \lhd \{r(E, T, x)\}\rangle} \tag{12}$$

$$\frac{\begin{array}{c} T, z \vdash \langle e_1, s\rangle \Longrightarrow \langle r_1, s_1\rangle \\ T, z \vdash \langle e_2, s_1\rangle \Longrightarrow \langle r_2, s_2\rangle \end{array}}{T, z \vdash \langle e_1\, op\, e_2, s\rangle \Longrightarrow \langle r_1\, op\, r_2, s_2\rangle} \tag{13}$$

$$\frac{T, z \vdash \langle e, s\rangle \Longrightarrow \langle r, s_1\rangle \quad X \notin \mathbf{var}(s)}{T, z \vdash \langle e.f, s\rangle \Longrightarrow \langle r(X), s_1 \lhd \{r(X, T, r.f)\}\rangle} \tag{14}$$

$$\frac{T, z \vdash \langle e, s_1\rangle \Longrightarrow \langle r_1, s_1\rangle \quad T, z \vdash \langle e_2, s_1\rangle \Longrightarrow \langle r_2, s_2\rangle \quad X \notin \mathbf{var}(s)}{T, z \vdash \langle e_1[e_2], s\rangle \Longrightarrow \langle r(X), s_2 \lhd \{r(X, T, r_1[r_2])\}\rangle} \tag{15}$$

We now consider the elaboration of literals. Scalar literals (constants, `null`) generate themselves:

$$\frac{k \text{ a scalar constant}}{T, z \vdash \langle k, s\rangle \Longrightarrow \langle k, s\rangle} \tag{16}$$

The object literal notation leads to the generation of events that create a new object and that initialize the fields of the object:

$$\frac{\begin{array}{l} T, z \vdash \langle e_1, s \lhd \{o(O, T)\}\rangle \Longrightarrow \langle r_1, s_1\rangle \\ T, z \vdash \langle e_2, s_1 \lhd \{w(O_1, T, r(O).f_1), w(O_1) = r_1\}\rangle \Longrightarrow \langle r_2, s_2\rangle \\ \ldots \\ T, z \vdash \langle e_n, s_{n-1} \lhd \{w(O_{n-1}, T, r(O).f_{n-1}), w(O_{n-1}) = r_{n-1}\}\rangle \Longrightarrow \langle r_n, s_n\rangle \\ s_{n+1} = s_n \lhd \{w(O_n, T, r(O).f_n), w(O_n) = r_n\}) \\ \forall i \leq n : O_i \notin \mathbf{var}(s_i)\ O \notin \mathbf{var}(s) \end{array}}{T, z \vdash \langle \{f_1 = e_1, \ldots, f_n = e_n, s\rangle \Longrightarrow \langle r(O), s_{n+1}\rangle} \tag{17}$$

Similarly for an array literal:

$$\frac{\begin{array}{l} T, z \vdash \langle e_1, s \lhd \{a(A,T)\}\rangle \Longrightarrow \langle r_1, s_1\rangle \\ T, z \vdash \langle e_2, s_1 \lhd \{w(A_1, T, r(A)[1]), w(A_1) = r_1\}\rangle \Longrightarrow \langle r_2, s_2\rangle \\ \ldots \\ T, z \vdash \langle e_n, s_{n-1} \lhd \{w(A_{n-1}, T, r(O)[n-1]), w(A_{n-1}) = r_{n-1}\}\rangle \Longrightarrow \langle r_n, s_n\rangle \\ s_{n+1} = s_n \lhd \{w(A_n, T, r(O)[n]), w(A_n) = r_n\} \\ \forall i \leq n : A_i \notin \mathbf{var}(s_i)\ A \notin \mathbf{var}(s) \end{array}}{T, z \vdash \langle [e_1, \ldots, e_n], s\rangle \Longrightarrow \langle r(O), s_{n+1}\rangle} \tag{18}$$

The rules for condition ($\Longrightarrow_c$) elaboration are similar, except that they are defined for conditions rather than expressions.

The rules for location elaboration are straightforward. Loc Vars are elaborated to locations, possibly with events.

$$\frac{E \notin \mathbf{var}(s)}{T, z \vdash \langle x, s\rangle \Longrightarrow_l \langle r(E), s \lhd \{r(E, T, x)\rangle} \tag{19}$$

$$\frac{T, z \vdash \langle L, s\rangle \Longrightarrow_l \langle l, s_1\rangle}{T, z \vdash \langle L.f, s\rangle \Longrightarrow_l \langle l.f, s\rangle} \tag{20}$$

$$\frac{T, z \vdash \langle L, s\rangle \Longrightarrow_l \langle l, s_1\rangle \quad T, z \vdash \langle e, s_1\rangle \Longrightarrow \langle r, s_2\rangle}{T, z \vdash \langle L[e], s\rangle \Longrightarrow_l \langle l[r], s_2\rangle} \tag{21}$$

3.4 Rules for statements

We now define the relation "Thread T, in the presence of the store z, can elaborate statement S into statement S', producing the action set s", written as:

$$T, z \vdash S \xrightarrow{s} S'$$

We omit s if it is empty. For an action set s and a pending condition d, we will use the notation $d \rightarrow s$ for the set $\{d \rightarrow u \mid u \in s\}$. Also, we use the notation $z \downarrow_{T,x}$ to indicate that the memory line $m(T, x)$ in z, i.e. the set of events in z generated by thread T on location x.

We treat the declaration of "normal" and "volatile" variables identically in the transition system using the markers `v(x)` and `n(x)` in the store to distinguish them. The Order Model will treat read and write accesses to volatile variables differently (in the definition of $\rhd$).

$$\frac{z \downarrow_{T,x} = \emptyset \quad s = \{\texttt{n(x)}\}}{T, z \vdash \texttt{init x = e} \xrightarrow{s} x = e} \qquad \frac{z \downarrow_{T,x} = \emptyset \quad s = \{\texttt{v(x)}\}}{T, z \vdash \texttt{volatile x = e} \xrightarrow{s} x = e} \tag{22}$$

Lock/unlock. Lock and unlock actions are elaborated into events immediately. Note that the transition for the entire configuration will depend on whether the various lock/unlock events offered by different threads can be reconciled (see Rule 37).

$$\frac{s = \{l(E,T,x)\}}{T, z \vdash \texttt{lock(x)} \xrightarrow{s} \epsilon} \quad \frac{s = \{u(E,T,x)\}}{T, z \vdash \texttt{unlock}(x) \xrightarrow{s} \epsilon} \tag{23}$$

Assignment. The rule for assignment statements uses the auxiliary relation introduced above to generate a set of events from the assignment statement.

$$\frac{T, z \vdash \langle e, \emptyset \rangle \Longrightarrow \langle r, s \rangle \quad s' = s \lhd \{w(E,T,x), w(E) = r\}}{T, z \vdash x = e \xrightarrow{s'} \epsilon} \tag{24}$$

Conditionals. Conditionals are immediately rewritten into pending conditionals, with events generated corresponding to the condition c:

$$\frac{T, z \vdash \langle c, \emptyset \rangle \Longrightarrow \langle d, s \rangle}{\begin{array}{l} T, z \vdash \texttt{if}(c)\{S\} \xrightarrow{s} d \ \rightarrow \ S \\ T, z \vdash \texttt{if}(c)\{S\}\texttt{else}\{S'\} \xrightarrow{s} d \ \rightarrow \ S : S' \end{array}} \tag{25}$$

A pending conditional may use the information in the store to discharge the conditional. This is the standard rule for conditionals. As usual for constraint systems, we use the notation $z \vdash d$ to indicate that the constraint d is entailed by the store z.

$$\frac{z \vdash d}{\begin{array}{l} T, z \vdash (d \ \rightarrow \ S) \longrightarrow S \\ T, z \vdash (d \ \rightarrow \ S : S') \longrightarrow S \end{array}} \quad \frac{z \vdash !d}{\begin{array}{l} T, z \vdash (d \ \rightarrow \ S) \longrightarrow \epsilon \\ T, z \vdash (d \ \rightarrow \ S' : S) \longrightarrow S \end{array}} \tag{26}$$

However, we also wish to allow speculative execution. This can be done by allowing a statement in a conditional to perform without the associated condition d being discharged. All events generated in this way must be made conditional on d:

$$\frac{T, z \vdash S \xrightarrow{s} S' \mid \epsilon}{\begin{array}{l} T, z \vdash (d \ \rightarrow \ S) \xrightarrow{d \rightarrow s} d \ \rightarrow \ S' \mid \epsilon \\ T, z \vdash (d \ \rightarrow \ S : S_1) \xrightarrow{d \rightarrow s} d \ \rightarrow \ S' : S_1 \mid (!d) \ \rightarrow \ S_1 \\ T, z \vdash (d \ \rightarrow \ S_1 : S) \xrightarrow{!d \rightarrow s} d \ \rightarrow \ S_1 : S' \mid d \ \rightarrow \ S_1 \end{array}} \tag{27}$$

Sequencing. First we present the standard transition rule for sequential execution. The sequential composition of two statements $(S_1 \ S_2)$ can take a step if the first statement S_1 can take a step.

$$\frac{T, z \vdash S_1 \xrightarrow{s} S_1' \mid \epsilon}{T, z \vdash S_1 \ S_2 \xrightarrow{s} S_1' \ S_2 \mid S_2} \tag{28}$$

The next transition models out of order execution. Suppose under condition d, S_1 can terminate, without generating any events. (For instance, S_1 may be

a pending conditional, and d may entail the negation of its condition.) Then we allow S_2 to execute, and generate events. However, these events can be communicated to memory only under the condition that d holds.

$$\frac{T, z \cup \{d\} \vdash S_1 \longrightarrow \epsilon \quad T, z \cup \{d\} \vdash S_2 \xrightarrow{s} S_2' \mid \epsilon \quad s_2' = d \ \rightarrow \ s_2}{T, z \vdash S_1 \ S_2 \xrightarrow{s} S_1 \ S_2' \mid \epsilon} \tag{29}$$

Iteration Iteration is performed in the standard way by translation into conditionals and sequencing. No special rules are needed for out of order or speculative execution, because the corresponding rules for sequencing and conditionals are automatically available.

$$T, z \vdash \texttt{while}(c)\{S\} \longrightarrow \texttt{if}(c)\{S \ \texttt{while}(c)\{S\}\} \tag{30}$$

$$T, z \vdash \texttt{do}\{S\}\texttt{while}(c) \longrightarrow S \ \texttt{if}(c)\{\texttt{do}\{S\}\texttt{while}(c)\} \tag{31}$$

Parallel Commands. A thread can be activated by generating a thread initiation event. This associates the thread with a thread identifier that must be distinct from the current thread, and from any previously created threads. (The conditions for combining action sets will ensure that this thread id is different from the thread id of any other thread created simultaneously within another parallel component.)

$$\frac{s = \{i(X, T)\} \quad T \notin \mathbf{var}(z) \cup \{T'\}}{T', z \vdash \texttt{thread}\{S\} \xrightarrow{s} \texttt{thread}(T)\{S\}} \tag{32}$$

$$\frac{\dfrac{T, z \vdash S \xrightarrow{s} S'}{T', z \vdash \texttt{thread}(T)\{S\} \xrightarrow{s} \texttt{thread}(T)\{S'\}} \quad \dfrac{T, z \vdash \ S \xrightarrow{s} \epsilon \quad s' = s \lhd \{x(X, T)\} \quad X \notin \mathbf{var}(s)}{T', z \vdash \texttt{thread}(T)\{S\} \xrightarrow{s'} \epsilon}}{} \tag{33}$$

We now state the rule for obtaining an action set for a parallel composition of statements.[5]

The first rule allows a single component of a parallel execution to contribute to the action set:

$$\frac{T, z \vdash P_1 \xrightarrow{s} P_1' \mid \epsilon}{\begin{array}{c} T, z \vdash P_1 \mid P_2 \xrightarrow{s} (P_1' \mid P_2) \mid P_2 \\ T, z \vdash P_2 \mid P_1 \xrightarrow{s} (P_2 \mid P_1') \mid P_2 \end{array}} \tag{34}$$

The second rule allows both components to contribute. Below, the requirement that the variables associated with s_1 and s_2 be distinct enforces that new

[5]Please note that " | " is being used in two different senses here – first as syntax for parallel composition in the object language, and second as syntax for alternatives in the meta-language. We trust the reader will use contextual information to distinguish the two kinds of uses.

threads created in s_1 and s_2 have distinct thread ids. It also enforces that the events of the two action sets are different. We use an evident, commonly used shorthand to collapse four separate inference rules (one for each choice of the two assumptions) into one figure:

$$\frac{T, z \vdash P_1 \xrightarrow{s_1} P_1' \mid \epsilon \quad T, z \vdash P_2 \xrightarrow{s_2} P_2' \mid \epsilon \quad s = s_1 \cup s_2 \quad \mathbf{var}(s_1) \cap \mathbf{var}(s_2) = \emptyset}{\begin{array}{c} T, z \vdash P_1 \mid P_2 \xrightarrow{s} P_1' \mid P_2' \\ T, z \vdash P_1 \mid P_2 \xrightarrow{s} P_1' \\ T, z \vdash P_1 \mid P_2 \xrightarrow{s} P_2' \\ T, z \vdash P_1 \mid P_2 \xrightarrow{s} \epsilon \end{array}} \tag{35}$$

This completes the specification of the rules for each kind of statement.

We allow multiple elaboration steps to be considered as a single elaboration step. This allows arbitrarily many statements to be elaborated into a single action set.

$$\frac{T, z \vdash S \xrightarrow{s_1} S_1 \quad T, z \vdash S_1 \xrightarrow{s_2} S_2}{T, z \vdash S \xrightarrow{s_1 \lhd s_2} S_2} \tag{36}$$

3.5 State transitions.

We now define the central relation of interest to us

$$\langle S, z \rangle \longrightarrow \langle S', z' \rangle \mid z'$$

We read $\langle S, z \rangle \longrightarrow \langle S', z' \rangle$ as "Statement S can be executed in store z to obtain statement S' and store z'". We read $\langle S, z \rangle \longrightarrow z'$ as "Statement S terminates in store z' to yield store z'.

$$\frac{\mathtt{0}, z \vdash S \xrightarrow{s} S' \mid \epsilon \quad z, s \rhd z'}{\langle S, z \rangle \longrightarrow \langle S', z' \rangle \mid z'} \tag{37}$$

Here we use the thread constant `0` to indicate the initial thread.

Proposition 3.1 *Suppose $T, z \vdash S \xrightarrow{s} S'$. Then z is a valid action set for CCM(HB) and $e(s) \cap e(z) = \emptyset$.*

Many other propositions may be proven, and will be the subject of future work.

4 Conclusion and future work

We have presented a simple framework for concurrent access to shared memory, the Concurrent Constraint-based Memory Model. This framework is parametrized with an underlying Order Model. Different event ordering proposals, such as Location Consistency and Happens Before ordering are consistent with this framework. Threads communicate with shared memory using

ordered sets of read and (conditional) write, lock and unlock events, together with constraints on variables associated with these events. The model supports a notion of volatile variables, locking/unlocking etc. We believe it can be extended to account for newer concurrency constructs such as atomic sections. Memory responds to these requests by linking reads with the writes in such a way that the associated constraints are uniquely satisfied, and the rules of the underlying Order Model are preserved. Each transition moves a valid store to a valid store and corresponds to the completion of the set of events communicated by threads in that step. This approach provides significant flexibility to the implementers of the memory module since it is consistent with a variety of advanced implementation techniques.

Based on these ideas we have presented a simple operational semantic framework (parametrized by Order Models) which supports out of order instruction execution, speculative execution and look-ahead execution while preserving the property that correctly synchronized programs cannot go wrong.

We believe the semantic framework is naturally motivated and easy to reason with informally. We have worked out test cases in [Pug04b] and [Pug04a]. Of the 30-odd cases, our proposed theory agrees with all but one of the test cases. For the one test case (Test 18), we believe there are sound semantic reasons for the desired result of the test case to be changed (Section A.1.5).

An implementation of CCMs in a constraint language on top of Prolog, using Constraint-Handling Rules is currently under development [SD04].

Future work. In future work we hope to develop a set of compiler transformations that are sound for CCM semantics, and understand which "commonly used" sequential program transformations are not valid. We expect to develop a denotational semantics and reasoning system for CCM Machines, exploiting past work on the semantics of concurrent constraint programming [SRP91].

We believe that it will be fruitful to develop "mechanism-oriented" interleaving abstract machines (as discussed in Section 1.3), as implementations of CCM Machines. Such machines would be easier to relate to implementations and would be useful in establishing their correctness.

We also intend to explore the definition of higher-level synchronization operations, such as atomic sections, on CCM Machines. Atomic sections specify a set of events that must be performed atomically, without specifying which locks to obtain on which variables.

We believe that the structure of CCM Machines may be usefully pulled back into programming language design, in the form of appropriate type systems and *assertional* systems. For instance, it may make sense for the programmer to directly assert ordering constraints on events and value constraints on locations in the program syntax. Conceptually, these constraints would be transmitted to Main Memory at run-time. For instance, a programmer may specify that a location can only take on the value `0` and `42`. The compiler may use such assertions to introduce early writes that could permit certain behaviors that may not have been permissible without these assertions. In effect, certain linkings

that would have been ruled out by Memory because they do not uniquely specify values for reads may now be permitted because the additional constraints force a unique valuation.

We hope to examine several current and proposed high-performance parallel machines and develop CCM Machine models for their operation.

Finally we note that CCM Machines may be simulated in any (concurrent) constraint programming language supporting the constraint system discussed in this paper. We are currently investigating such an implementation with collaborators.

Acknowledgements. My thanks to Doug Lea for keeping on pushing this work in productive directions. His focus on getting convergence with the other models forced me to look for ways to express the intuition behind compiler reasoning within the constraint-based framework. His insistence on understanding where LC semantics differed from an ordering based on the "Happens Before" relation led to the realization that the techniques of this paper worked also with the "Happens Before" relationship. This led to the formulation of the HB-machine. Martin Rinard suggested that the framework be applied to other consistency models. Hans Boehm suggested that "thin air reads" be formalized more carefully.

My thanks also to Vivek Sarkar, Guang Gao and Zhang Yuan for detailed discussions that helped move the subject of this paper forward.

Thanks to Robert O'Callahan, Perry Cheng, Julian Dolby and Kemal Ebcioglu of the PERCS Programming Model project at IBM TJ Watson Research for discussions on related topics and to Tom Schrijvers for a careful reading of the paper.

My thanks to Bill Pugh and the participants of the Java Memory Model mailing list for their contributions to the definition of the problem, especially the identification of test cases.

References

[Adv04] S. Adve. Sc-. Technical report, 2004. `http://www.cs.umd.edu/ pugh/java/memoryModel/`.

[AG95] S. Adve and K. Gharachorloo. Shared Memory Consistency Models: A tutorial. Technical report, Digital Western Research Laboratory, 1995.

[APP99] S. Adve, V. S. Pai, and P.Ranganthan. Recent Advances in Memory Consistency Models for Hardware Shared-Memory Systems. *Proceedings of the IEEE*, 87(3):445–455, March 1999.

[GJSB00] J. Gosling, W. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2000.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual ACM Symposium on Computer Architecture*, pages 15–25, June 1990.

[GS97] A. Gontmakher and A. Schuster. Java consistency: Non-operational characterization for the Java Memory Behavior. Technical report, Department of Computer Science, Technion, 1997.

[GS00] G. Gao and V. Sarkar. Location Consistency – A New Memory Model and Cache Consistency Protocol. *IEEE Transactions on Computers*, 49(8):798–813, August 2000.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), 1979.

[MAS00] J.-W. Maessen, Arvind, and X. Shen. Improving the Java Memory Model Using CRF. In *OOPSLA*, 2000.

[MP04] J. Manson and W. Pugh. The Manson/Pugh model. Technical report, U Maryland, 2004. `http://www.cs.umd.edu/ pugh/java/memoryModel/`.

[Pug01] W. Pugh. Proposal for Java Memory Model and Thread Specification Revision, 2001. JSR 133, http://www.jcp.org/en/jsr/detail?id=133.

[Pug04a] W. Pugh. Java Memory Model and Thread Specification Revision, 2004. JSR 133, http://www.jcp.org/en/jsr/detail?id=133.

[Pug04b] W. Pugh. Java Memory Model Causality Test Cases. Technical report, U Maryland, 2004. `http://www.cs.umd.edu/ pugh/java/memoryModel/`.

[Pug04c] W. Pugh. The Java Memory Model is Fatally Flawed. *Concurrency: Practice and Experience*, 2004? To appear.

[Sar92] V. Saraswat. The Category of Constraint Systems is Cartesian Closed. In *LICS '92*, pages 341–345, 1992.

[Sar93] V. Saraswat. *Concurrent Constraint Programming*. Doctoral Dissertation Award and Logic Programming. MIT Press, 1993.

[SAR99] X. Shen, Arvind, and L. Rudolph. Commit-reconcile and fences (crf): a new memory model for architects and compiler writers. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 150–161, 1999.

[SD04] T. Schrijvers and B. Demoen. JMMSolve: a generative reference implementation of CCM Machines. Technical Report Report CW 379, Katholieke Universiteit Leuven, January 2004.

[SRP91] V. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, 1991.

[YGL04] Y. Yang, G. Gopalakrishna, and G. Lindstrom. A Generic Operatonal Memory Model Specification Framework for Multithreaded Program Verification. Technical report, School of Computing, U. of Utah, 2004.

A Test cases

In this section we discuss the Causality Test cases of [Pug04b]. The results are summarized in Table 6. Cases involving volatile variables and thread creation ordering conditions are not discussed.

Each test case shows an Action Set that can be generated from the corresponding program using the operational semantics. We do not show all the order relations that are generated by the operational semantics, just enough to entail all of them.

We use an evident shorthand to indicate chains of ordering relations between events.

In some of the programs below, we present a "Simplified Action Set" for ease of exposition. Such an action set omits the underlying event and order constraints. The treatment of variables (such as `R1,R2`) which are used by only one thread is simplified since their linking is always forced. In particular, these variables are assigned to once and are read at most once; we know that the value returned in such cases is the value written, so the constraints can be simplified by using the same variable for the reads and writes. Finally, we replace read (write) terms `r(X)` (`w(X)`) with the underlying variable `X` (since the event variable is not used for any other purpose). All these simplifications are sound and considerably simplify the presentation of constraints from a pedagogical point of view.

For readers not accustomed to constraint reasoning, some of the cases below might appear "magical". Constraint reasoning can typically be understood *declaratively* (one reasons in terms of valuations) and *operationally* (one reasons in terms of the behavior of an underlying constraint solver). If the constraint solver is complete (as we assume) then the two forms of reasoning are equivalent. Often the first is useful to understand the result, and the second to understand how the result was obtained.

For ease of exposition, we will often describe each test case from three points of view, the two mentioned above, as well as the "compiler's" point of view discussed in [Pug04b]. We caution that in our opinion the "compiler's" point of view is suggestive but not definitive. It does help, though, to show how "standard" compiler reasoning fits into the constraint-based reasoning framework.

A.1 Promotion of conditional events based on case analysis

The example below shows that a condition may be discharged if it holds for each value that can be generated for a variable by the given linking. Discharging the condition causes events guarded by that condition to become unconditional, typically enabling other flows of information.

bf Test Case	**"Desired" result**	**HB result**	**Reference**	**Comments**
Test 1	Allowed	ok	Page 47	
Test 2	Allowed	ok	Page 53	
Figure 2	Allowed	ok	Page 55	
Test 3	Allowed	ok	Page 54	
Test 4	Forbidden	ok	Page 59	
Test 5	Forbidden	ok	Page 60	
Test 6	Allowed	ok	Page 49	
Figure 15	Allowed	ok	Page 50	
Test 7	Allowed	ok	Page 67	
Test 8	Allowed	ok	Page 56	
Test 9	Allowed	ok	Page 57	
Test 9a	Allowed	ok	Page 58	
Test 10	Forbidden	ok	Page 61	
Test 11	Allowed	ok	Page 68	
Test 12	Allowed	ok	Page 65	
Test 13	Forbidden	ok	Page 62	
Figure 11	Allowed	ok	Page 63	
Figure 19	Forbidden	ok	Page 64	
Test 14	Forbidden	ok	Page 71	Volatile
Test 15	Forbidden	ok	Page 73	Volatile
Test 16	Allowed	ok	Page 69	
Figure 10	Allowed	ok	Page 70	
Test 17	Allowed	ok	Page 51	
Test 18	Allowed	**Wrong**	Page 52	
Test 19	Allowed			(Thread joins)
Test 20	Allowed			(Thread joins)

HB-semantics agrees with LC-semantics for all these cases. The Test cases are from [Pug04b] and the Figures are from [Pug04a].

Table 6: The result of Test Cases, per HB-semantics

A.1.1 Test 1

Program	Action set

`init x=0;`
`init y=0;`
`thread {`
`r1=x;`
`if (r1>=0) {`
`y=1;`
`}`
`} |`
`thread {`
`r2=y`
`x=r2`
`}`
	`w(Xi,0,x), w(Xi)=0, w(Yi,0,y), w(Yi)=0,`
	`i(T1c, T1),`
	`r(X1r,T1,x), w(R1w,T1,r1),`
	`w(R1w)=r(X1r), r(R1r,1,r1),`
	`r(R1r)>=0 ->(w(Y1w,T1,y), w(Y1w)=1),`
	`x(T1d,T1),`
	`Xi<<Yi<<T1c<<X1r<<R1w<<R1r<<Y1w<<T1d`
	
	`i(T2c,T2),`
	`r(Y2r,T2,y), w(R2w,T2,r2), w(R2w)=r(Y2r),`
	`r(R2r,T2,r2), w(X2w,T2,x), w(X2w)=r(R2r),`
	`x(T2d,T2),`
	`T2c<<oY2r<<R2w<<R2r<<X2w<<T2d`

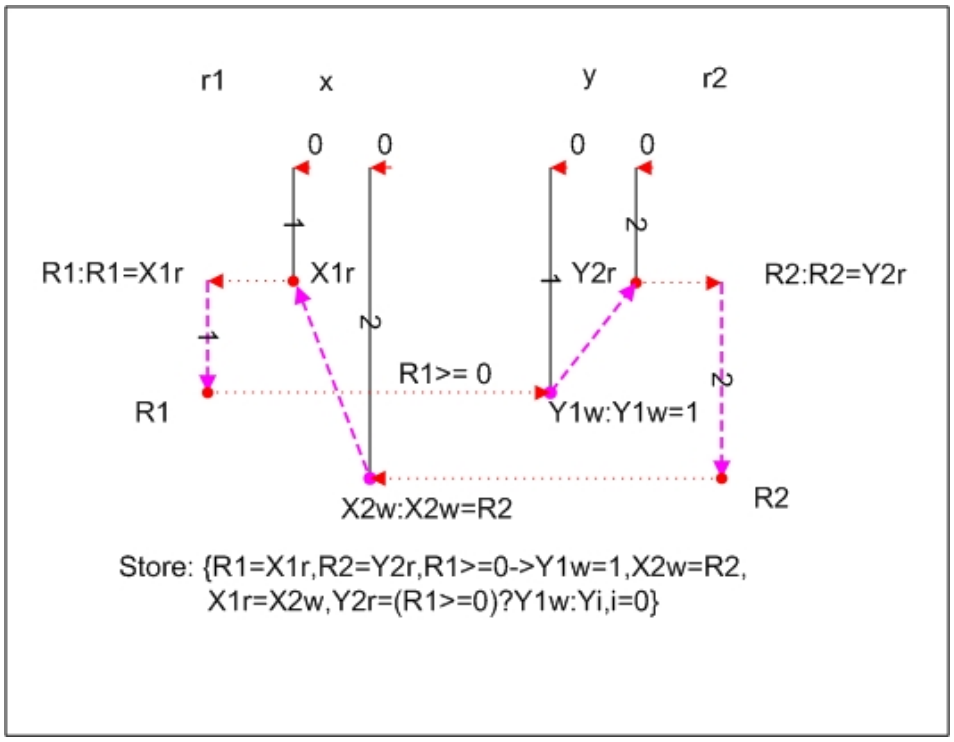
Behavior:	`r1==r2==1`
Decision:	*Allowed.*
HB semantics:	*ok*
Linking:	{`r(X1r)=w(X2w),r(R1r)=w(R1w), r(R2r)=w(R2w),` `(r(Y2r)=(r(R1r)>=0)?w(Y1w):w(Y2i))`}

Note that the linkings force `r(R1r)=r(R2r)`. The set of constraints boils down to `R=(R>=0)?1:0`, for `R=r(R1r)=r(R2r)`. This has only one solution, `R=1`. See Figure 3.

Operationally, one may understand the constraints as follows. The linking forces `Y2r` to receive its value from one of two places. For both of these places it can be shown that the condition can be discharged. Hence the condition can be discharged. But this forces only one of the two values to be feasible `1`. Hence the system has a unique solution.

The compiler-based point of view from [Pug04b] is:

> *Decision: Allowed, since interthread compiler analysis could determine that x and y are always non-negative, allowing simplification of r1 ¿= 0 to true, and allowing write y = 1 to be moved early.*



An LC representation of a run for Test Case 1. The dotted arrows represent write constraints, and the dashed arrows represent linkings. Vertical lines represent memory lines and hence implicitly represent ordering relations – in LC programs without synchronization only two events from the same thread targeted to the same location can be ordered.

Figure 3: A run for Test Case 1 showing desired behavior

A.1.2 Test 6

Program	Action Set

init x=0;
init y=0;
thread {
r1=x;
if (r1==1) {
y=1;
}
} |
thread {
r2=y;
if (r2==1) {
x=1;
}
if (r2==0) {
x=1;
}
}
	w(Xi,0,x), w(Xi)=0,
	w(Yi,0,y), w(Yi)=0,
	i(T1c, T1),
	r(X1r,1,x),
	w(R1w,1,r1), w(R1w)=r(X1r),
	r(R1r,1,r1),
	(r(R1r)==1) -> (w(Y1w,1,y), w(Y1w)=0),
	x(T1d,T1),
	Xi<<Yi<<T1c<<X1r<R1w<<R1r<<Y1w<<T1d,
	
	i(T2c,T2),
	r(Y2r,T2,y),
	w(R2w,T2,r2), w(R2w)=r(Y2r),
	r(R2r1,T2,r2),
	r(R2r1)==1) -> (w(X2w1,T2,x), w(X2w1)=1),
	r(R2r2,T2,r2),
	r(R2r2)==0) -> (w(X2w2,T2,x), w(X2w2)=1),
	x(T2d,T2),
	Yi<<T2c<<Y2r<<R2w<<R2r1<<X2w1,
	X2w1<<R2r2<<X2w2<<T2d

Behavior:	r1==r2==1
Decision:	*Allowed.*

HB semantics:	ok
Linkage:	{r(X1r)=(r(R2r1)=1)?w(X2w1):(r(R2r2)=0)?w(X2w2):w(Xi), r(R1r)=w(R1w), r(R2r1)=w(R2w), r(R2r2)=w(R2w), r(Y2r)=(r(R1r)==1)?w(Y1w):w(Yi)}

The constraints simplify to

```
r(R1r)=(r(R2r1)==1)?1:(r(R2r2)==0)?1:0,
r(R2r1)=(r(R1r)==1)?1:0}
```

This has only one solution.

Operationally, based on the linkage for y, it can be established that y can take on only the values 0 or 1. For both of these cases, the linkage for x establishes that x=1. In turn, through Thread 1, this establishes that y=1.

From [Pug04b]:

> *Decision: Allowed. Intrathread analysis could determine that*

thread 2 always writes 1 to x and hoist the write to the beginning of thread 2.

A.1.3 Figure 15

Program	Action Set

`init x=0;`
`init y=0;`
`thread {`
`r1=x;`
`if (r1==1) {`
`y=1;`
`}`
`} |`
`thread {`
`r2=y;`
`if (r2==1) {`
`x=1;`
`} else {`
`x=1;`
`}`
`}`
	`w(Xi,0,x), w(Xi)=0,`
	`w(Yi,0,y), w(Yi)=0,`
	`i(T1c, T1),`
	`r(X1r,1,x),`
	`w(R1w,1,r1), w(R1w)=r(X1r),`
	`r(R1r,1,r1),`
	`(r(R1r)==1)`
	`-> (w(Y1w,1,y), w(Y1w)=0),`
	`x(T1d,T1),`
	`Xi<<Yi<<T1c<<X1r<<R1w,`
	`R1w<<R1r<<Y1w<<T1d,`
	
	`i(T2c,T2),`
	`r(Y2r,T2,y),`
	`w(R2w,T2,r2), w(R2w)=r(Y2r),`
	`r(R2r,T2,r2),`
	`r(R2r)==1)`
	`-> (w(X2w1,T2,x), w(X2w1)=1)`
	`: (w(X2w2,T2,x), w(X2w2)=1),`
	`x(T2d,T2),`
	`Yi<<T2c<<Y2r<<R2w<<R2r<<T2d,`
	`r(R2r1)==1)`
	`-> (R2r1<<X2w1<<T2d)`
	`: (R2r1<<X2w2<<T2d)`

Behavior:	`r1==r2==1`
Decision:	*Allowed.*
HB semantics:	ok

Linkage:	`{r(X1r)=(r(R2r1)==1)?w(X2w1):w(X2w2),`
	`r(R1r)=w(R1w), r(R2r)=w(R2w),`
	`r(Y2r)=(r(R1r)==1)?w(Y1w):w(Yi)}`

The constraints simplify to

```
r(R1r)=(r(R2r)==1)?1:1,
r(R2r)=(r(R1r)==1)?1:0
```

This has only one solution.

Operationally, the linking for `x` can be used to established that `x=1` unconditionally. This propagates through and establishes `y=1`.

A.1.4 Test 17

Program	Simplified Action Set

`init x=0;`
`init y=0;`
`thread {`
`r1=x;`
`if (r1!=42) {`
`x=42;`
`}`
`r1=x;`
`y=r1;`
`} |`
`thread {`
`r2=y;`
`x=r2;`
`}`
	`Xi=0, Yi=0`
	`R3=X1r1,`
	`(R3!=42) -> X1w=42`
	`R1=X1r2,`
	`Y1w=R1`
	
	`R2=Y2r,`
	`X2w=R2`

Behavior:	`r1==r2==r3==42`
Decision:	*Allowed.*
HB semantics:	*ok*
Linking:	`X1r1=X2w, X1r2=(R3!=42)?X1w:X2w,Y2r=Y1w`

The constraint simplifies to `R3=(R3!=42)?42:R3` which has a unique solution.

Operationally, `X1r2` can be established to be `42`, given that `X1r1` is reading from `X2w`. This value propagates through.

From [Pug04b]:

> *Decision: Allowed. A compiler could determine that at r1 = x in thread 1, is must be legal for to read x and see the value 42. Changing r1 = x to r1 = 42 would allow y = r1 to be transformed to y = 42 and performed earlier, resulting in the behavior in question.*

A.1.5 Test 18

Program	Action Set

init x=0;
init y=0;
thread {
r3=x;
if (r3==0) {
x=42;
}
r1=x;
y=r1;
} |
thread {
r2=y;
x=r2;
}
	R3=X1r1,
	if (R3=0)
	X1w=42
	R1=X1r2,
	Y1w=R1
	
	R2=Y2r,
	X2w=R2

Behavior:	r1==r2==r3==42
Decision:	*Allowed.*
HB semantics:	*ok*
Linking:	X1r1=X2w, X1r2=(R3=0)?X1w:X2w,Y2r=Y1w

The constraint simplifies to R3=(R3=0)?42:R3 which has many solutions. Hence this linkage must be rejected.

Operationally, the linkage cannot be used to establish that X1r2 is 42, because of the nature of the condition.

This case is very similar to Test 17 which was allowed. So why a different solution? In Test 18 the conclusion R3=42 is forced simply by the conditions associated with the writes and the linkages. In Test 18, an *additional* piece of information needs to be added. For instance, arguably the memory may make a "dataflow" conservative analysis and conclude that (R3 in {0,42}. This constraint, in conjunction with R3=(R3=0)?42:R3 will force R3=42.

However, doing so would be unsound according to the semantics. The semantics forces all solutions to have R3=0. This example shows that a program transformation rule which adds runtime assertions based on conservative data flow analysis is unsound for the semantics. For these assertions may allow inferences to be made that could force a linking that was previously rejected because it did not force a unique solution to now be accepted. We leave open the question if there is a variation of this rule which is sound for this semantics. One candidate would be a rule which requires that an assertion R in S (which capatures the result of such a data flow analysis) can be added only if S contains a

"junk" value. (See also our proposal in Section 4 for allowing programmers to specify assertions.)

From [Pug04b]:

> *Decision: Allowed. A compiler could determine that the only legal values for x are 0 and 42. From that, the compiler could deduce that r3 != 0 implies r3 = 42. A compiler could then determine that at r1 = x in thread 1, is must be legal for to read x and see the value 42. Changing r1 = x to r1 = 42 would allow y = r1 to be transformed to y = 42 and performed earlier, resulting in the behavior in question.*

A.2 Forward substitution

A.2.1 Test 2

Program	Action Set

`init x=0;`
`init y=0;`
`thread {`
`r1=x;`
`r2=x;`
`if (r1==r2)`
`y=1;`
`} |`
`thread {`
`r3=y`
`x=r3`
`}`
	`w(Xi,0,x), w(Xi)=0,`
	`w(Yi,0,y), w(Yi)=0,`
	`i(T1c, T1),`
	`r(X1r1,1,x),`
	`w(R1w,1,r1), w(R1w)=r(X1r1),`
	`r(X1r2,1,x),`
	`w(R2w,1,r2), w(R2w)=r(X1r2),`
	`r(R1r,1,r1), r(R2r,1,r2),`
	`r(R1r)==r(R2r)`
	`-> (w(Y1w,1,y), w(Y1w)=1, R2r<<Y1w)`
	`x(T1d,T1),`
	`Xi<<Yi<<T1c<<X1r1<R1w<<X1r2<<R2w,`
	`R2w<<R1r<<R2r<<T1d,`
	`Y1w<<T1d`
	
	`i(T2c,T2),`
	`r(Y2r,2,y), w(R3w,2,r3), w(R3w)=r(Y2r),`
	`r(R3r,2,r3), w(X2w,2,x), w(X2w)=r(R3r),`
	`x(T2d,T2),`
	`Yi<<T2c<<Y2r<<R3w<<R3r<<X2w<<T2d`

Behavior:	`r1==r2==r3==1`
Decision:	*Allowed.*

HB semantics:	*Allowed.*

Linkage:	`{r(X1r1)=w(X2w), r(X1r2)=w(X2w),`
	`r(R1r)=w(R1w), r(R2r)=w(R2w), r(R3r)=w(R3w),`
	`r(Y2r0=(r(R1r)=r(R2r))?w(Y1w):w(Yi)}`

Note that the linkings force `r(R1)=r(R2)=r(R3)`. The set of constraints boils down to `R=(R=R)?1:0`, for `R=r(R1)`. This has only one solution, `R=1`.

Operationally, the system exploits the knowledge that the two reads of `x` by Thread 1 can be bound to the same write event, without knowing the value returned by that event. This enabled Thread 1 to establish `y=1` unconditionally, and the chain of connections forces the two reads of `x` to obtain `1`.

The compiler-based point of view from [Pug04b] is:

> *Decision: Allowed, since redundant read elimination could result in simplification of r1 == r2 to true, allowing y = 1 to be moved early.*
>
> *Notes: In SC executions, both reads of x always return the same value (i.e., zero), so that r1 == r2 is always true in SC executions.*

A.2.2 Test 3

Program

`init x=0;`
`init y=0;`
`thread {`
`  r1=x;`
`  r2=x;`
`  if (r1==r2) {`
`    y=1;`
`  }`
`} |`
`thread {`
`  r3=y;`
`  x=r3`
`} |`
`thread {`
`  x=2;`
`}`

Behavior:	`r1==r2==r3==1`
Decision:	*Allowed.*
HB semantics:	*Allowed.*

Test 3 has the same justification as Test 2. The extra thread is irrelevant, per the discussion on Linking Monotonicity in Section 2.6.

From [Pug04b]:

> *Decision: Allowed, since redundant read elimination could result in simplification of r1 == r2 to true, allowing y = 1 to be moved*

early.

Notes: Same as test case 2, except there are SC executions in which r1 != r2

A.2.3 Figure 2

Program	Action Set

`init p = {x=0};`
`init q=null;`
`q=p;`
`thread {`
`  m = p.x;`
`  n = q.x;`
`  o = p.x;`
`} |`
`thread {`
`  p.x = 3;`
`}`
	`o(O,0),`
	`w(Ox0w,0,r(O).x), w(Ox0w)=0,`
	`n(p),w(P0w,0,p), w(P0w)=r(O),`
	`n(q),w(Q0w,0,q), w(Q02)=null,`
	`r(P0r,0,p),`
	`w(Q0w,0,q), w(Q0w)=r(P0r),`
	`O<<Ox0w<<P0w<<Q0w<<P0r<<Q0w`
	
	`i(T1c,T1),`
	`r(P1r,T1,p),`
	`r(P1xr,T1,r(P1r).x),`
	`w(Mw,T1,m),w(Mw)=r(P1xr),`
	`r(Q1r,T1,q),`
	`r(Q1xr,T1,r(Q1r).x),`
	`w(Nw,T1,n),w(Nw)=r(Q1xr),`
	`r(P1r2,T1,p),`
	`r(P1r2xr,T1,r(P1r2).x),`
	`w(Ow,T1,o),w(Ow)=r(P1r2xr),`
	`x(T1d,T1),`
	`Q0w<<T1c<<P1r<<P1xr<<Mw<<Q1r<<Q1xr,`
	`Q1xr<<Nw<<P1r2<<P1r2xr<<Ow<<T1d`
	
	`i(T2c,T2),`
	`r(P2r,T1,p),`
	`w(P2xw,T1,r(P2r).x), w(P2xw)=3,`
	`x(T2d,T2),`
	`Q0w<<T2c<<P2r<<P2xw<<T2d`

Behavior:	`m == o == 0,n == 3`
Decision:	*Allowed.*

HB semantics:	*Allowed.*

Linkage:	{`r(P0r)=w(P0w), r(P1r)=w(P0w), r(P1r2)=w(P0w)`
	`r(P2r)=w(P0w), r(Q1r)=w(Q0w),`
	`r(P1xr)=w(Ox0w), r(P1r2xr)=w(Ox0w),`
	`r(Q1xr)=w(P2xw)`}

The three read events `P1xr, Q1xr, P1r2xr` see the two writes `Ox0w` and `P2xw` and may independently choose either value. The given choice produces the desired result.

A.3 Memory may perform arithmetic reasoning

A.3.1 Test 8

Program	Simplified Action set

`init x=0;`
`init y=0;`
`thread {`
`r1=x;`
`r2=1+r1*r1-r1;`
`y=r2;`
`} |`
`thread {`
`r3=y;`
`x=r3;`
`}`
	`R1=X1r,`
	`R2=1+R1*R1-R1,`
	`Y1w=R2`
	
	`R3=Y2r,`
	`X2w=R3`

Behavior:	`r1==r2==1`
Decision:	*Allowed.*
HB semantics:	*Allowed.*
Linking:	{`X1r=X2w, Y2r=Y1w`}

The obvious linking works. Note that the constraint `R1=1+R1*R1-R1` must be verified to have a unique solution. This particular case is easy because the technique of propagating values may be used to determine that `1` is the only value. In general such systems of constraints may be undecidable.

From [Pug04b]:

> *Decision: Allowed. Interthread analysis could determine that x and y are always either 0 or 1, and thus determine that r2 is always 1. Once this determination is made, the write of 1 to y could be moved early in thread 1.*

A.3.2 Test 9

Program	Simplified Action Set

`init x=0;`
`init y=0;`
`thread {`
`r1=x;`
`r2=1+r1*r1-r1;`
`y=r2;`
`} |`
`thread {`
`r3=y;`
`x=r3;`
`}`
`thread {`
`x=2;`
`}`
	`R1=r(X1r),`
	`R2=1+R1*R1-R1,`
	`w(Y1w)=R2,`
	
	`R3=r(Y2r),`
	`w(X2w)=R3`
	
	`w(X3w)=2`

Behavior:	`r1==r2==1`
Decision:	*Allowed.*
HB semantics:	*Allowed.*
Linking:	{ `r(X1r)=w(X2w), r(Y2r)=w(Y1w)`}
Comments:	Same as(8). Note that the presence of the third thread has no bearing on the result. (See Linking Monotonicity, Section 2.6.)

The extra thread can be ignored by Link Monotonicity.
From [Pug04b]:

> *Decision: Allowed. Similar to test case 8, except that the x is not always 0 or 1. However, a compiler might determine that the read of x by thread 2 will never see the write by thread 3 (perhaps because thread 3 will be scheduled after thread 1). Thus, the compiler can determine that r1 will always be 0 or 1.*

A.3.3 Test 9a

Program

`init x=2;`
`init y=0;`
`thread {`
`  r1=x;`
`  r2=1+r1*r1-r1;`
`  y=r2;`
`} |`
`thread {`
`  r3=y;`
`  x=r3;`
`} |`
`thread {`
`  x=0;`
`}`

Behavior:	`r1==r2==1`
Decision:	*Allowed.*
HB semantics:	*Allowed.*

Same as Test 9, except that the initial value of `x` and the value written by Thread3 are swapped. This has no bearing on the run establishing the question.

From [Pug04b]:

> *Decision: Allowed. Similar to test case 8, except that the x is not always 0 or 1. However, a compiler might determine that thread 3 will always execute before thread 1, and that therefore the initial value of 2 will not be visible to the read of x in thread 1. Thus, the compiler can determine that r1 will always be 0 or 1.*

A.4 Unresolvable mutual dependencies

This section covers some cases in which linkings are forbidden because they would correspond to generating concrete read values from "thin air", i.e. values that have not been communicated by a thread to memory, or because there are mutually unresolvable dependencies, i.e. values cannot be established unconditionally, via logical reasoning.

A.4.1 Test 4

Program	Action Set

init x=0;
init y=0;
thread {
r1=x;
y=r1;
} |
thread {
r2=y;
x=r2;
}
	w(Xi,0,x), w(Xi)=0,
	w(Yi,0,y), w(Yi)=0,
	i(T1c, T1),
	r(X1r,1,x),
	w(R1w,1,r1), w(R1w)=r(X1r),
	r(R1r,1,r1),
	w(Y1w,1,y), w(Y1w)=r(R1r),
	x(T1d,T1),
	Xi<<Yi<<T1c<<X1r<R1w<<R1r<<Y1w<<T1d,
	
	i(T2c,T2),
	r(Y2r,2,y),
	w(R2w,2,r2), w(R2w)=r(Y2r),
	r(R2r,2,r2),
	w(Y2w,2,y), w(Y2w)=r(R2r),
	x(T2d,T2),
	Yi<<T2c<<Y2r<<R2w<<R2r<<Y2w<<T2d

Behavior:	r1==r2==1
Decision:	*Forbidden.*
HB semantics:	ok
Linkage:	{r(X1r)=w(X2w), r(R1r)=w(R1w), r(R2r)=w(R2w), r(Y2r)=w(Y1w)}
Comment:	*Not a solution.*

The proposed linkage is not a solution because it does not uniquely force the variables: simply forces all the variables to be the same. No other linkage produces this behavior either. Thus the requirement that the set of constraints have a unique solution is the mechanism used by CCM Machines to disallow "reads from thin air".

From [Pug04b]:

> *Decision: Forbidden: values are not allowed to come out of thin air*

A.4.2 Test 5

Program

`init x=0;`
`init y=0;`
`init z=0;`
`thread {`
`  r1=x;`
`  y=r1;`
`} |`
`thread {`
`  r2=y`
`  x=r2`
`} |`
`thread {`
`  z=1;`
`} |`
`thread {`
`  r3=z;`
`  x=r3;`
`}`

Behavior:	`r1==r2==1, r3==0`
Decision:	*Forbidden.*
HB semantics:	ok

Similar to the previous case, by Linking Monotonicity.
From [Pug04b]:

> *Decision: Forbidden: values are not allowed to come out of thin air, even if there are other executions in which the thin-air value would have been written to that variable by some not out-of-thin air means.*

A.4.3 Test 10

Program	Action Set

init x=0;
init y=0;
init z=0;
thread {
r1=x;
if (r1==1) {
y=1;
}
} |
thread {
r2=y;
if (r2==1) {
x=1;
}
}
thread {
z=1;
} |
thread {
r3=z;
if (r3==1) {
x=1;
}
}
	w(Xi,0,x), w(Xi)=0,
	w(Yi,0,y), w(Yi)=0,
	w(Zi,0,z), w(Zi)=0,
	i(T1c, T1),
	r(X1r,1,x),
	w(R1w,1,r1), w(R1w)=r(X1r),
	r(R1r,1,r1),
	(r(R1r)==1) -> (w(Y1w,1,y), w(Y1w)=1),
	x(T1d,T1),
	Xi<<Yi<<Zi<<T1c<<X1r<R1w<<R1r<<Y1w<<T1d,
	
	i(T2c,T2),
	r(Y2r,T2,y),
	w(R2w,T2,r2), w(R2w)=r(Y2r),
	r(R2r,T2,r2),
	r(R2r)==1) -> (w(X2w,T2,x), w(X2w)=1),
	x(T2d,T2),
	Zi<<T2c<<Y2r<<R2w<<R2r<<X2w<<T2d
	
	i(T3c,T3),
	w(Z3w,T3,z), w(Z3w)=1,
	x(T3c,T3),
	Zi<<Z3W
	
	i(T4c,T4),
	r(Z4r,T4,4),
	w(R3w,T4,r3), w(R3w)=r(Z4r),
	r(R3r,T4,r3),
	r(R3r)==1) -> (w(X4w,T4,x), w(X4w)=1),
	x(T4d,T4),
	Zi<<T4c<<X4r<<R3w<<R3r<<X4w<<T4d

Behavior:	r1==r2==1, r3==0
Decision:	*Forbidden.*
HB semantics:	ok
Linking:	{ r(X1r)=(r(R2r)==1)?w(X2w):w(Xi), r(Y2r)=(r(R1r)==1?w(Y1w):w(Yi), r(Z4r)=w(Zi)}

To obtain r3==0, r(Z4r) must be linked to w(Zi). But then the only possi-

ble linking for `r(X1r)` that can generate the value `1` is the given one. However, the constraints simplify to `r(R1r)=1 <-> r(R2r)=1` which has many solutions. Hence this solution must be rejected. There are no other possibilities for obtaining the desired result.

Operationally, note that there is no way to unconditionally establish a value for `X1r`. This means that multiple choices are possible, because the condition can always be falsified yielding a situation in which the variable is not constrained.

From [Pug04b]:

> *Decision: Forbidden. This is the same as test case 5, except using control dependences rather than data dependences.*

A.4.4 Test 13

Program

`init x=0;`
`init y=0;`
`thread {`
`r1=x;`
`if (r1==1) {`
`y=1;`
`}`
`}`
`thread {`
`r2=y;`
`if (r2==1) {`
`x=1;`
`}`
`}`

Behavior:	`r1==r2==1`
Decision:	*Forbidden.*
HB semantics:	ok
Comments:	Same as Test 10 with extra threads. By Linking Monotonicity (Section 2.6) this is also forbidden.

From [Pug04b]:

> *Decision: Disallowed. In all sequentially consistent executions, no writes to x or y occur and the program is correctly synchronized. The only SC behavior is r1 == r2 == 0.*

A.4.5 Figure 11

Program	Simplified Action Set

`init x=0;`
`init y=0;`
`thread {`
`  r1=x;`
`  if (r1!=0) {`
`   y=1;`
`  }`
`}`
`thread {`
`  r2=y;`
`  if (r2!=0) {`
`   x=1;`
`  }`
`}`
	`Xi=0, Yi=0,`
	`R1=X1r,`
	`R1!=0 -> Y2w=1,`
	`R2=Y2r,`
	`R2!=0 -> X2w=1`

Behavior:	`r1==r2==0`
Decision:	*Should be the only behavior.*
HB semantics:	ok

The desired behavior arises from the linking `X1r=Xi,Y1r=Yi`. Another possibility is `X1r=(R2!=0)?X2w:Xi`. This case reduces to the previous one if the linking for `Y1r` is `Y1r=Yi`. Symmetrically for the case in which `Y2r=(R1!=0)?Y1w:Yi` and `X1r=Xi`. Consider the only case left,

`{X1r=(R2!=0)?X2w:Xi, Y2r=(R1!=0)?Y1w:Yi}`

This reduces to `R1=(R2!=0)?1:0, R2=(R1!=0)?1:0` which has two solutions (`R1=R2=1` and `R1=R2=0`) and is hence inadmissible.

Operationally, there is no way to discharge either conditional, hence `1` cannot be unconditionally liberated for `x` or `y`.

A.4.6 Figure 19

Program	Action Set

init x=0;
init y=0;
init z=0;
thread {
r1=x;
if (r1!=0) {
y=r1;
}
} |
thread {
r2=y;
if (r2!=0) {
x=r2
}
} |
thread {
z=1;
} |
thread {
r0=z;
if (r0==1) {
x=42;
}
}
	w(Xi)=0, w(Yi)=0, w(Zi)=0,
	R1=r(X1r),
	R1!=0 -> w(Y1W)=R1,
	
	R2=r(Y2r),
	R2!=0 -> w(X2w)=R2,
	
	w(Z3w)=1,
	
	R0= r(Z4r),
	R0==1 -> w(X4w)=42

Behavior:	r0=0, r1==r2==42
Decision:	*Forbidden.*
HB semantics:	ok
Linkage:	{r(X1r)=(R2!=0)?w(X2w):w(Xi), r(Y2r)=(R1!=0)?w(Y1w):w(Yi),} r(Z4r)=w(Zi)}

The analsysis is similar to Figure 11. The constraints simplify to R1=(R2!=0)?R2:0 and R2=(R1!=1)?R1:0. This is the same as R1=R2 and is vacuously true, i.e. any valuation is a solution. Hence this solution must be rejected.

A.5 Array references

A.5.1 Test 12

Program	Action Set

init x=0;
init y=0;
init a=[1,2];
thread {
r1=x;
a[r1]=0;
r2=a[0];
y=r2;
} |
thread {
r3=y;
x=r3;
}
	w(Xi,0,x), w(Xi)=0,
	w(Yi,0,y), w(Yi)=0,
	w(A0i,0,a[0]), w(A0i)=1,
	w(A1i,0,a[1]), w(A1i)=2,
	i(T1c, T1),
	r(X1r,T1,x),
	w(R1w,T1,r1), w(R1w)=r(X1r),
	r(R1r,1,r1),
	w(AR1w,T1, a[R1r]), w(AR1w)=0,
	r(A01r, T1, a[0]),
	w(R2w,T1,r2), w(R2w)=r(A01r)
	r(R2r,T1,r2),
	w(Y1w,T1,y), w(Y1w)=r(R2r),
	x(T1d,T1),
	Xi<<Yi<<A0i<<A1i<<T1c<<X1r<R1w<<R1r<<AR1w,
	AR1w<<A01r<<R2w<<R2r<<Y1w<<T1d,
	
	i(T2c,T2),
	r(Y2r,T2,y),
	w(R3w,T2,r3), w(R3w)=r(Y2r),
	r(R3r,T2,r3),
	w(X2w,T2,x), w(X2w)=r(R3r),
	x(T2d,T2),
	A1i<<T2c<<Y2r<<R3w<<R3r<<X2w<<T2d

Behavior:	r1==r2==r3==1
Decision:	*Forbidden.*
HB semantics:	*ok*
Linkage:	r(X1r)=w(X2w),r(Y2r)=w(Y1w), r(A01r)=(r(R1r)=0)?w(AR1w):w(A0i)

The Order Model forces the linking for `r(A01r)`, since the only accesses to the array are from Thread T1.

But this constraint simplifies to `r(R1r)=(r(R1r)=0)?0:1` which has two solutions `r(R1r)=0` and `r(R1r)=1`. Hence this linking is not accepted.

No other valid linking generates the required behavior.

Let a be the action set given above. Note that the following linking does

generate the desired behavior:

$$\{\texttt{r(A01r)} = \texttt{w(A0i)}, \texttt{r(X1r)} = \texttt{w(X2w)}, \texttt{r(Y2r)} = \texttt{w(Y1w)}\}$$

However, this is not a valid linking for a: the write term `w(A0i)` is not visible to `A01r` in a. To see this, consider the extension $a' = a \cup \{\texttt{r(R1r)} = \texttt{0}\}$. Now $a' \vdash w(AR1w, T1, a[0])$. The visibility rules of the underlying Order Model force this event on $a[0]$ to hide the event `A0i`.

Thus, through the write visibility criteria, CCM Machines enforce that a linking cannot answer a read event with a write event that becomes visible to the read *only if* the linking is established. Visibility must be established *before* the linking is established.

From [Pug04b]:

> *Decision: Disallowed. Since no other thread accesses the array a, the code for thread 1 should be equivalent to:*
>
> ```
> r1 = x
> a[r1] = 0
> if (r1 == 0)
>           r2 = 0
>         else
>           r2 = 1
> y = r2
> ```
>
> *With this code, it is clear that this is the same situation as test 4.*

A.6 Statement Reordering

A.6.1 Test 7

Program	Simplified Action Set

`init x=0;`
`init y=0;`
`init z=0;`
`thread {`
`r1=z;`
`r2=x;`
`y=r2;`
`} |`
`thread {`
`r3=y;`
`z=r3;`
`x=1;`
`}`
	`R1=r(Z1r),`
	`R2=r(X1r),`
	`w(Y1w)=R2,`
	
	`R3=r(Y2r),`
	`w(Z2w)=R3,`
	`w(X2w)=1`

Behavior:	`r1==r2==r3==1`
Decision:	*Allowed.*
HB semantics:	*Allowed.*
Linking:	{ `r(Z1r)=w(Z2w), r(X1r)=w(X2w),` `r(Y2r)=w(Y1w)`}
Comments:	

The obvious linking works. This is the classic example of the commutativity and associativity of constraints: the order of constraint imposition does not matter.

From [Pug04b]:

> *Decision: Allowed. Intrathread transformations could move r1 = z to after the last statement in thread 1, and x = 1 to before the first statement in thread 2.*

A.6.2 Test 11

Program

`init x=0;`
`init y=0;`
`init z=0;`
`thread {`
`  r1=z;`
`  w=r1;`
`  r2=x;`
`  y=r2;`
`} |`
`thread {`
`  r4=w;`
`  r3=y;`
`  z=r3;`
`  x=1;`
`}`

Behavior:	`r1==r2==r3==r4==1`
Decision:	*Allowed.*
HB semantics:	ok

The obvious linking works.
From[Pug04b]:

> *Decision: Allowed. Reordering of independent statements can transform the code to:*
>
> ```
> Thread 1:
> r2 = x
> y = r2
> r1 = z
> w = r1
>
> Thread 2:
> x = 1
> r3 = y
> z = r3
> r4 = w
> ```
>
> *after which the behavior in question is SC.*
>
> *Note: This is similar to test case 7, but extended with one more rung in the ladder*

A.6.3 Test 16

Program	Simplified Action Set

`init x=0;`
`thread {`
`r1=x;`
`x=1;`
`} |`
`thread {`
`r2=x;`
`x=2;`
`}`
	`R1=X1r,`
	`X1w=1,`
	
	`R2=X2r,`
	`X2w=2`

Behavior:	`r1==2,r2==1`
Decision:	*Allowed.*
HB semantics:	ok
Linking:	`X1r=X2w, X2r=X1w`
Comments:	The obvious linking works.

The obvious linking works.
From [Pug04b]:

> *Decision: Allowed.*

A.6.4 Figure 10

Program	Simplified Action Set

`init x=0;`
`init y=0;`
`thread {`
`  x=1;`
`  r1=y;`
`} |`
`thread {`
`  y=2;`
`  r2=x;`
`}`
	`X1w=1,`
	`R1=Y1r,`
	
	`Y2w=2,`
	`R2=X2r,`

Behavior:	`r1==1,r2==0`
Decision:	*Allowed.*
HB semantics:	ok
Linking:	`Y1r=Yi, X2r=Xi`
Comments:	The obvious linking works.

The obvious linking works.
From [Pug04b]:

> *Decision: Allowed.*

A.7 Volatile Reads

A.7.1 Test 14

Program

```
init a=0;
init b=0;
volatile y=0;
thread {
 r1=a;
 if (r1==0) {
  y=1;
 } else {
  b=1;
 }
} |
thread {
 do {
  r2=y;
  r3=b;
 } while (r2+r3==0)
 a=1;
```

Behavior:	`r1==r3==1, r2==0`
Decision:	*Forbidden.*
HB semantics:	*ok*

Let us consider the action sets generated by the main thread and the two threads separately. The maximal action set generated by the main thread is s_0:

```
n(a), w(Ai,0,a), w(Ai)=0,
n(b), w(Bi,0,b), w(Bi)=0,
v(y), w(Yi,0,y), w(Yi)=0,
Ai<<Bi<<Yi
```

The maximal action set that can be generated by the first thread is s_1:

```
i(T1c,T1)
r(A1r,T1,a),
w(R1w,T1,r1), w(R1w)=r(A1r),
r(R1r,T1,r1),
r(R1r)==0 -> (w(Y1w,T1,y),w(Y1w)=1),
!(r(R1r)==0) -> (w(B1w,T1,b),w(B1w)=1),
x(T1d,T1),
T1c<<A1r<<R1w<<R1r,
```

```
R1r<<Y1w<<T1d,
R1r<<B1w<<T1d
```

The action sets generated by the second thread depend on how many times the loop is unrolled. For instance, unrolling it once ($k = 1$) will give:

```
r2=y;
r3=b;
if (r2+r3==0)) {
  do {
   r2=y;
   r3=b;
  } while (r2+r3==0)
}
a=1;
```

Similarly for $k > 1$. For $k = 1$ we can get the action set s_2:

```
r(Y2r,T2,y),
w(R2w,T2,r2), w(R2w)=r(Y2r),
r(B2r,T2,b),
w(R3w,T2,r3), w(R3w)=r(B2r),
r(R2r,T2,r2),
r(R3r,T2,r3),
!(r(R2r)+r(R3r)==0) -> (w(A2w,T2,a),w(A2w)=1)
```

This can be generated by the Out of Order execution rule because:

$$\mathtt{T}, \mathtt{z}, !(\mathtt{r}(\mathtt{R2r}) + \mathtt{r}(\mathtt{R3r}) == 0) \vdash (\mathtt{r}(\mathtt{R2r}) + \mathtt{r}(\mathtt{R3r})) == \mathtt{0} \ \rightarrow \ \mathtt{S} \longrightarrow \epsilon$$

Now, can $s_0 \cup s_1 \cup s_2$ generate the desired behavior? The only possibility is the linking:

$$\begin{array}{l}\{\mathtt{r}(\mathtt{A1r}) = (!(\mathtt{r}(\mathtt{R2r}) + \mathtt{r}(\mathtt{R3r}) == 0))?\mathtt{w}(\mathtt{A2w}) : \mathtt{w}(\mathtt{Ai}),\\ \mathtt{r}(\mathtt{Y2r}) = (\mathtt{r}(\mathtt{R1r}) == 0)?\mathtt{w}(\mathtt{Y1w}) : \mathtt{Yi},\\ \mathtt{r}(\mathtt{B2r}) = (!(\mathtt{r}(\mathtt{R1r}) == 0))?\mathtt{w}(\mathtt{B1w}) : \mathtt{Bi},\\ \mathtt{r}(\mathtt{R1r}) = \mathtt{w}(\mathtt{R1r}), \mathtt{r}(\mathtt{R2r}) = \mathtt{w}(\mathtt{R2w}), \mathtt{r}(\mathtt{R3r}) = \mathtt{w}(\mathtt{R3w})\}\end{array}$$

When added to $s_0 \cup s_1 \cup s_2$, this forces:

$$\begin{array}{l}\{\mathtt{r}(\mathtt{A1r}) = (!(\mathtt{r}(\mathtt{Y2r}) + \mathtt{r}(\mathtt{B2r}) == 0))?\mathtt{1} : \mathtt{0},\\ \mathtt{r}(\mathtt{B2r}) = (\mathtt{r}(\mathtt{A1r})! = 0)?\mathtt{1} : \mathtt{0},\\ \mathtt{r}(\mathtt{Y2r}) = (\mathtt{r}(\mathtt{A1r}) == 0)?\mathtt{1} : \mathtt{0}\}\end{array}$$

and this does indeed have a single solution.

However, this solution is not admissible. Because of the volatile read requirement, the new store must have `Y1w<<Y2r`. Therefore, this constraint has to be added as part of o_i (synchronization conditions). But under this constraint, `A1r<<A2w` and hence the linking for `A1r` is not valid.

From [Pug04b]:

Decision: Disallowed In all sequentially consistent executions, r1 = 0 and the program is correctly synchronized. Since the program is correctly synchronized in all SC executions, no non-sc behaviors are allowed.

A.7.2 Test 15

Program

```
init a=0;
init b=0;
volatile x=0;
volatile y=0;
thread {
 r0=x;
 if (r0==1) {
   r1=a;
 } else {
   r1=0;
 }
 if (r1==0) {
  y=1;
 } else {
  b=1;
 }
} |
thread {
 do {
  r2=y;
  r3=b;
 } while (r2+r3==0);
 a=1;
} |
thread {
 x=1;
}
```

Behavior:	`r0==r1==r3==1, r2==0`
Decision:	*Forbidden.*
HB semantics:	*ok*

This reduces to the previous case if the read for `x` in Thread 1 is answered by the write in Thread 3. This is the only possibility for achieving `r1==1`.

From [Pug04b]:

Decision: Disallowed In all sequentially consistent executions, r1 = 0 and the program is correctly synchronized. Since the program is correctly synchronized in all SC executions, no non-sc behaviors are allowed.