# Automated Verification of UPC Memory Consistency

By
Oystein Thorsen

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of
MASTER OF SCIENCE IN COMPUTER SCIENCE

MICHIGAN TECHNOLOGICAL UNIVERSITY
2006

This thesis, ”Automated Verification of UPC memory consistency”, is hereby approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE IN COMPUTER SCIENCE.

DEPARTMENT: Computer Science

Signatures:

Thesis Advisor: ______________________
Dr. Charles Wallace

Department Chair: ______________________
Dr. Linda Ott

Date: ______________________

*To my wife for all her love and support.*

**Abstract**

UPC is a shared memory extension to ANSI C. Its memory consistency model is relaxed, allowing for optimization, but also permitting behavior which may be surprising to the naive programmer. To allow better understanding of this memory model, we present a tool for analyzing the behavior of UPC programs. Given an execution trace, the tool determines whether the results are compatible with the UPC memory model. The tool is targeted at newcomers to UPC who want to learn about its memory model and at developers who want to verify possible behaviors of their programs.

# Contents

**1 Background and Motivation** **7**
1.1 Message Passing Model . . . . . 7
1.2 Shared Memory Model . . . . . 8
1.3 Sequential Memory Consistency . . . . . 9
1.4 UPC Memory Consistency . . . . . 10
1.5 Related Work . . . . . 11

**2 UPC Language Features** **12**
2.1 `upc_fence` . . . . . 13
2.2 Collective Operations . . . . . 13
2.2.1 `upc_barrier` . . . . . 13
2.2.2 `upc_notify` and `upc_wait` . . . . . 13

**3 Input** **15**
3.1 Operations Representation . . . . . 15
3.2 Tuple Generation . . . . . 17

**4 Output** **19**

**5 Implementation** **21**
5.1 What is SAT and CNF-SAT . . . . . 21
5.2 Using SAT To Find Observed Orderings . . . . . 21
5.3 Reducing To CNF-SAT . . . . . 22
5.4 SAT encoding . . . . . 22
5.5 UPC Memory Model Constraints In First Order Logic . . . . . 23
5.5.1 Helper Functions . . . . . 23
5.5.2 Basic Constraints . . . . . 24
5.5.3 UPC Constraints . . . . . 24
5.5.4 requireProgramOrder . . . . . 24
5.5.5 requireReadValue . . . . . 26
5.6 Clause Generation . . . . . 26
5.7 Extracting Orderings from SAT Results . . . . . 27

**6 Future Work** **28**

**7 Examples** **29**
7.1 Example 1 . . . . . . . . . . 30
7.2 Example 2 . . . . . . . . . . 31
7.3 Example 3 . . . . . . . . . . 32
7.4 Example 4 . . . . . . . . . . 33
7.5 Example 5 . . . . . . . . . . 34
7.6 Example 6 . . . . . . . . . . 35
7.7 Example 7 . . . . . . . . . . 36
7.8 Example 8 . . . . . . . . . . 37
7.9 Example 9 . . . . . . . . . . 38
7.10 Example 10 . . . . . . . . . . 39
7.11 Example 11a . . . . . . . . . . 40
7.12 Example 11b . . . . . . . . . . 41
7.13 Example 11c . . . . . . . . . . 42
7.14 Example 12 . . . . . . . . . . 43

# 1 Background and Motivation

A uniprocessor computer is sufficient for most applications, but a single processor can only get you so far. In addition, the more you try to stretch this limit the cost increases exponentially, practically limiting the maximum computation power one processor can give you. Instead of spending a lot of money on that last bit of performance boost, it could in many cases be much better to use more than one computer or processor at a time. This way it will take longer to execute each instruction, but we can execute more than one instruction at a time. Furthermore, as long as all instructions do not rely on the result of the previous instruction, we can solve much bigger problems than the fastest uniprocessor computer in much less time, because of the availability of increased raw computing power.

Any simulation where each data point changes based on the neighboring data points can be executed on many processors in parallel. (Many physical phenomena like weather simulations fall under this category.) If we use a grid to divide the data set between the processors each processor can compute all data points in the interior of its part of the data set. Then it would need to do some communication to get the neighboring data points on the exterior edge of its data partition. Organizing and optimizing this communication between the processors is far from a trivial task and can be difficult for even the most experienced programmer.

In a regular C program you do computation based on some initial values and give some output and any intermediate results are stored in variables for reuse. A parallel program also has initial values in the same way, but intermediate results are often calculated by other threads and it is up to the programmer to make sure that the thread retrieves those results before continuing with its computation. This communication is solved in one of two ways based on the programming language (or parallel extension) you use.

## 1.1 Message Passing Model

The *Message Passing Interface*[8] (MPI) is the de facto standard for parallel computing on distributed memory systems. Its name implies that this language is indeed a good example of the *message passing* programming model.

Message passing provides the most explicit way of handling communication between processors by providing functions to send and receive chunks of data between processors. This basic form of processor interaction consists of one processor sending some data and one processor receiving that data. With this we can create very specialized communication patterns because we control the exchange

of every single bit of data.

The disadvantage is that it can get very complicated even for small programs. In many cases we need to do load-balancing during runtime so we do not always know which processors should communicate at any given time. But since a thread is required to know where to send its data and/or receive data we need to calculate this before communicating. In some cases this may not even be possible.

## 1.2 Shared Memory Model

The other alternative is that every thread operates on a *global shared address space*. This means that every processor can access global shared memory the same way it accesses its local memory. Allowing each thread to read and write to a shared memory location using regular reads and writes instead of special message functions means that the programmer no longer needs to keep track of how the memory is distributed among the threads.

The behavior of such a program is described by how each thread observes the changes to memory. These changes to memory are read and write operations of all the threads running the program. Since a memory operation cannot be observed instantaneously by all threads, there is no way of making sure that all threads observe the operations in the same order. If there are no constraints on the orderings of these operations, it will be impossible for a programmer to predict the behavior of his program. The example in Figure 1 shows a UPC program with a very nonintuitive output that is actually possible when there are few or no constraints on the ordering of how a thread observes changes to memory.

Normally a programmer would assume that both threads would output 1 since this would be the output if both threads executed each line of code at the same time. A programmer would also normally assume that one of the threads outputed 1 imagining that one thread executed its program before the other.

This kind of unusual behavior is often experienced due to optimizations like reordering of operations (if the instructions on line 3 and 4 were reordered[1]) or caching (if the written values on line 4 were not observed by the other thread until after it executed the instruction on line 5). Sometimes this behavior is acceptable to the programmer, but most often it is necessary to limit the possible behavior to just a few possibilities.

To do this we use a memory (consistency) model which works as a contract

[1] There is no apparent reason not to reorder these instructions since they operate on different variables.

| | | Thread 0 | Thread 1 |
|---|---|---|---|
| **Program** | 1 | `shared int x = 0;` | `shared int x = 0;` |
| | 2 | `shared int y = 0;` | `shared int y = 0;` |
| | 3 | `upc_barrier;` | `upc_barrier;` |
| | 4 | `x = 1;` | `y = 1;` |
| | 5 | `print(y);` | `print(x);` |
| **Output** | | `0` | `0` |

Figure 1: An example of unusual, but possible program behavior with a relaxed memory model. (The `upc_barrier` is a synchronization function that makes sure that both threads declare the variables before continuing. `upc_barrier` will be explained in detail in section 2.2.1.)

between the language implementors and the programmer. This model puts constraints on how operations should be ordered and determines what legal values the read operations can return. We call this a contract because a programmer usually wants a model that is easy to understand while a language implementor usually wants a model that allows as much optimization as possible. Usually a model that is easy to understand means more constraints on the observed ordering which means less possibilities for optimization. A memory model is usually a compromise between these two philosophies.

## 1.3 Sequential Memory Consistency

Sequential memory consistency[13] (SC) is the first proposed memory model. It is the easiest memory model to understand and the least optimizable. SC is similar to the uniprocessor model. This means that all operations are serialized among the threads, causing them to be observed in the same order on all threads. This common order preserves the order dictated by each program. It is important to notice that this model still has the non-deterministic nature of a parallel program. Imagine having two decks of cards (representing the program instructions for two processors) that you shuffle once. The combined deck (the observed order for both threads) preserves the order from the initial decks, but the cards can still be combined into many different orderings. The fact that you can have all these

different orderings even under the strictest memory model is the implicit cost of executing a program in parallel.

If we take another look at the program in Figure 1 we can imagine that Thread 0 prints $0$ because it printed before Thread 1 updated $y$ (line 3). We can also imagine that Thread 1 can print $0$ before Thread 0 updates $x$, but the output specified in Figure 1 (where both threads print $0$) cannot be explained in terms of sequential consistency.

If we had no memory model it would be like shuffling the two decks as many times as you would like and then throwing them up in the air, and every card that was overlapping when they fell to the ground was observed in the same order by all threads. It would be completely impossible for the programmer to anticipate the behavior.

Even though sequential consistency is desired sometimes, it is too strict under most circumstances. The sequential memory model leaves very little room for compiler optimization techniques like caching and operation reordering.

## 1.4 UPC Memory Consistency

Like many other memory models [6, 9, 10, 11, 18] the UPC memory model allows a more relaxed ordering of operations than the SC memory model. As mentioned above it is very easy for a programmer to reason about a very strict memory model, but it is hard to optimize. The more relaxed a memory model is the harder it is for a programmer to reason about. The reason for this of course is the increased non-determinism caused by different observed orderings on different threads and even reordering of instruction execution.

In the UPC memory model [21] all memory operations come in two flavors: a strict version and a relaxed version. A program with only strict operations obeys sequential consistency. Only relaxed operations allow the compiler to re-order most operations (this will be explained later), and to decide when a thread observes a change in memory (it must happen sometime before the program ends).

Anyone new to UPC would find it hard to understand the UPC memory model (even many experienced UPC programmers find it hard to understand [12]) so it would be very beneficial for them to have a tool which can show how unusual behavior is actually possible with the UPC memory model.

For experienced programmers who already know the memory model, it would be a handy tool to verify larger programs which would allow them to take maximum advantage of the optimization possibilities.

## 1.5 Related Work

The work done in this thesis report is built on the work done by Yang in his Ph.D. Thesis [20] where he created a similar tool for the Intel Itanium memory model [3]. He shows that a non-operational memory model specification can be made executable by converting the problem to a SAT problem and then using a SAT solver to determine legality, which we will discuss the details of later.

This work and Yang's work fit into the "model checking" approach to verification. In this approach, the system being verified is modeled and an exhaustive search is performed on the model space, to determine whether the correctness conditions are met in all possible searches. There has been a significant amount of work on automated reasoning about memory consistency using model checking [7, 14, 16]. Unlike these earlier efforts, this work and Yang's work use Boolean propositions rather than ordered binary decision diagrams (OBDDs) as system models. Verification then reduces to the SAT problem. Recently this approach has shown to significantly outperform the OBDD approach [5].

We can think of model checking as a bottom-up approach, with intensive search of possible model configurations. In contract, the top-down term-rewriting approach [17, 4] takes the original execution trace and rewrites it, based on a set of axioms that constitute the memory model specification. The term rewriting procedure aims to reduce the original input to a tautology, much like a standard deductive proof.

# 2 UPC Language Features

To understand the behavior of a program we have two sources of information; the source code and the program's output. Since the program can (potentially) output any values read from memory, we will assume that we know everything the program reads from memory.

In many parallel programs it is not predetermined which thread should execute a task or a segment of code. But to prevent all threads from executing the same task there must be some preconditions that ensure that only one thread executes that part of the code. And since we know the observed state of the memory we can assume that we know what operations/instructions where executed by each thread.

Based on this information we can define an *execution trace*, which is an idealized list of operations for each thread ordered by when they were executed. The list is idealized because the compiler can rearrange operations before execution, but from the programmer's view the execution trace is the order of execution.

From the program in Figure 1 the execution trace for Thread 0 would be a write to $x$ followed by a read from $y$. For Thread 1 we would have a write to $y$ followed by a read from $x$. From the source code we know that the two writes use the value $1$ and based on the output specified in Figure 1, we know that the two reads returned the value $0$.

The UPC memory model constrains how communication between threads is observed, it also constrains how operations can be reordered on a thread before execution. We will look at these constraints in detail later.

As explained earlier we want a more relaxed memory model so that we can optimize better. This includes reading and writing values from/to memory earlier or later (by reordering operations) that what was specified by the programmer in the source code. Sometimes we may even use an earlier calculated value instead of checking the shared memory. We can do this because it is the same as if other threads observed a write immediately followed by a read from the same thread. So if the memory model constraints allows the read to be observed right after the write then everything is ok. So even if a thread does not always execute all operations they "act" as if they did, so when we create an observed ordering for all threads we must include all writes to memory.

We know that strict operations are sequentially consistent and must form an order that all threads agree upon. This means that every thread must observe all strict operations and that if you only look at the strict operations they should all be observed in the same order on all threads. The same does not apply to relaxed

operations so they can be observed in a different order on different threads.

An execution trace from a UPC program is legal if all constraints described in the UPC memory model are satisfied. This includes an observed temporal ordering for each thread where the read value of a read match with the latest observed write to the same variable/memory location. All strict operations must have been observed in the same order by all threads (the global strict order).

## 2.1 `upc_fence`

The `upc_fence` operation is a way of ensuring that everything a thread has done before the fence will be observed by all other threads before the thread's operations after the fence. But since this is the same as any strict operation we can safely treat `upc_fence` as a strict operation that does nothing[2].

## 2.2 Collective Operations

UPC [19] has certain *collective* operations. A collective operation is an operation which is called by all threads and in the same order. We will discuss the collective operations that are used for synchronization purposes.

### 2.2.1 `upc_barrier`

This is conceptually the easiest synchronization operation offered by UPC. When a thread calls `upc_barrier` it cannot execute any further instructions until all other threads have called the same `upc_barrier`. When each thread has called `upc_barrier` they should also agree on the content of the shared memory. This practically breaks up the program into two subprograms where the first program "returns" whatever is in memory at the barrier and the second program uses that memory as its initial values.

In terms of constraints we can think of `upc_barrier` as a collective version of `upc_fence`, where the operations issued before the `upc_barrier` has to be observed before before every other threads call to that `upc_barrier`.

### 2.2.2 `upc_notify` and `upc_wait`

In many cases a barrier would be sufficent to synchronize threads, but sometimes some threads get to the barrier much earlier than others. So instead of letting these

[2]The UPC Spec describes this as a strict no-op.

threads wait idly until the last thread gets to the barrier, we want them to use this time to perform some work.

The *split-phase barrier* consists of two collective operations, where `upc_notify` signifies the beginning of the barrier and `upc_wait` signifies the end. When a thread calls `upc_notify` it notifies the other threads that it has reached the barrier. Between calling `upc_notify` and calling `upc_wait` it can execute any number of instructions, but without the guarantee that all the threads have finished their computations up until the barrier. Normally a programmer would let the thread do computation on local data or shared data that has not changed since before an earlier barrier. When a thread eventually calls `upc_wait` it must stop and do nothing just like a normal barrier until all threads have called `upc_notify`.

The split-phase barrier is the most tricky of the operations to model because of its complicated collective behavior. On a thread the notify and wait both work as a fence since no operations can be moved past any of them, but operations issued before a `upc_notify` can be observed anytime before the corresponding `upc_wait` on other threads and operations issued after `upc_wait` can be observed anytime after `upc_notify` on other threads.

If no operations are issued between `upc_notify` and `upc_wait` the pair of operations works exactly like `upc_barrier`.

Example 11 (page 40) shows how a barrier prevents a thread from observing operations issued (by other threads) after the barrier. In Example 11b (page 41) we can see that we have the same problem when the barrier is replaced by a split-phase barrier. Example 11c is an example of what the read operation must have observed for this execution trace to be legal.

# 3 Input

The input to the tool must include the sequence of operations executed by each thread, as well as the values returned by all read operations (an execution trace). Following the naming conventions in the UPC memory model we have developed a grammar (Figure 3) that encapsulates all the information needed to apply the rules of the UPC memory model. Example 1 (page 30) is a very simple example of an execution trace following the grammar. The execution trace shows a list of initial values and a list of operations for each thread which is ordered based on the program order (the order specified by the programmer). In addition to the constraints of the context free grammar, all threads must call the collective operations (in the same order), a `upc_wait` must be preceded by a `upc_notify` and no collective calls are allowed between a `upc_notify` and a `upc_wait`.

When we compare operations we can check the line in the execution trace to see if they are read or write operations, if they are strict or relaxed and what values they read/write. Other information like what thread they are on requires a little more searching. Instead of doing this searching every time we need this information, it is much more practical to store the operations as a set of tuples where all possible information is stored in one place. We will discuss how we store the same information as tuples and we will explain how the tool converts an execution trace written according to the grammar on page 16 to a set of tuples.

## 3.1 Operations Representation

Operations are encoded as tuples because the explicit information allows us to describe the ordering constraints between them in a better way. To be able to store all the needed information we use a 9-tuple, but not every field is used for all tuples.

Tuple $i = \langle id, src, iss, obs, pc, op, cons, var, data \rangle$ where

| | |
|---|---|
| execution_trace | : init_part[a] exec_part |
| init_part | : **startvalues** init_expr* |
| init_expr | : variable **=** value |
| exec_part | : **numthreads =** value thread_trace* |
| thread_trace | : **thread** instr* |
| instr | : reg_instr<br>\| sync_instr |
| reg_instr | : op **(** variable **,** value **)** |
| op | : **RR**<br>\| **RW**<br>\| **SR**<br>\| **SW** |
| sync_instr | : **upc_fence**<br>\| **upc_barrier**<br>\| **upc_notify**<br>\| **upc_wait** |

[a] Optional

Figure 2: Execution trace grammar.

| | |
|---|---|
| **id** $i = id$ | : global ID of the tuple |
| **src** $i = src$ | : the ID of the original operation |
| **iss** $i = iss$ | : issuing thread $\in [0..\mathit{THREADS}-1]$ |
| **obs** $i = obs$ | : observing thread $\in [0..\mathit{THREADS}-1]$ |
| **pc** $i = pc$ | : program counter |
| **op** $i = op$ | : operation type $\in \{Read, Write, Notify, Wait, Fence\}$ |
| **cons** $i = cons$ | : coherence type $\in \{Strict, Relaxed\}$ |
| **var** $i = var$ | : shared memory variable |
| **data** $i = data$ | : a read or stored value |

and *THREADS* is the total number of threads involved in the execution trace.

## 3.2 Tuple Generation

Each operation in the execution trace is represented by either one tuple or one tuple for each thread. The general rule is that if one of the fields in the tuple can have a different value for each thread we create a tuple for each thread such that all the possible values are represented.

An operation that is represented by more than one tuple will have a different **obs** value for each of the tuples. We will call the tuple where **iss** $i$ = **obs** $i$ the original tuple and the other tuples will be called duplicates since they are the observed version of the original operation/tuple.

Let us look at an example set of tuples that were generated from the execution trace in example 1 (page 30).

| id | src | iss | obs | pc | op | var | cons | data |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | Read | x | Relaxed | 1 |
| 1 | 1 | 0 | 0 | 1 | Write | x | Relaxed | 2 |
| 2 | 1 | 0 | 1 | 1 | Write | x | Relaxed | 2 |
| 3 | 3 | 1 | 1 | 0 | Read | x | Relaxed | 2 |
| 4 | 4 | 1 | 1 | 1 | Write | x | Relaxed | 1 |
| 5 | 4 | 1 | 0 | 1 | Write | x | Relaxed | 1 |

Figure 3: A set of tuples generated from the execution trace in example 1.

The first operation is represented by tuple 0 since only the issuing thread will observe a relaxed read. The first operation from the second thread is represented by tuple 3 for the same reasons. The two write operations are represented by one

tuple for each of the threads where the observed thread is different for the two tuples. Notice that the duplicate tuples have the id of the original tuple in the **src** column.

All strict operations are observed by all threads, but since all operations must be ordered in the same way on all threads with respect to a strict operation, we do not need to duplicate strict operations.

A collective operation is a strict operation, but collective operations are duplicated[3] because they have a different **pc** value for each thread.

[3]For collective operations the "original tuple" will be the one issued by thread 0. It could have been any thread, but it has to be a single thread so that all the calls to the same collective operation have the same value for **src**.

# 4 Output

If the execution trace is legal, we want to show a graph displaying the observed ordering for each thread, and if it is not legal we want to show why. For this we use a graph drawing package [1] that displays the tuples as nodes in a directed graph.

Since the constraints on the tuples are all constraints between pairs of tuples we can have a solution where there is an edge between all pairs of tuples. But since we have a legal execution trace and that each observed order is a temporal order there can be no cycles. So without losing any information in the ordering we can remove edges implied by transitivity. For illegal execution traces we know there must be a cycle preventing this temporal order. In that case we want to show all the tuples and edges involved in that cycle (or cycles).

Based on the outcome of the program, two operations will have a directed edge between them showing which operation must have been observed before the other for the execution trace to be legal. Since the graph represents a temporal order of operations there can be no cycles. If a cycle is shown the program would find the execution trace to be illegal.

To display multiple orderings in a single graph we have two options: each order is represented by different edges, or each order is represented by different nodes[4]. The two options are displayed in Figure 4. We chose to go with the second option because visually it is much easier to find a cycle in the graph than to find a cycle which consists of edges of only one color.

We chose to go with the second option because it is easier to think about constraints that way.

All regular (read and write) strict operations are in black because they are part of the observed order for all threads. See example 9 on page 38 for a graph with strict and relaxed operations.

The `upc_barrier` operation is also in black even though it is represented by multiple tuples because based on the definition of a barrier it is natural to display it as one operation (see example 11a on page 41).

All other tuples are color-coded[5] based on which thread observes that tuple.

In contrast to the `upc_barrier`, `upc_notify` and `upc_wait` are not displayed as one tuple. The reason is that threads can observe other operations between calls to the same collective operation (`upc_notify` or `upc_wait`). Even

[4]This is why some operations are represented by more than one tuple.

[5]In this paper thread 0 is represented by the color blue (or darker) and thread 1 is represented by the color green (or lighter).

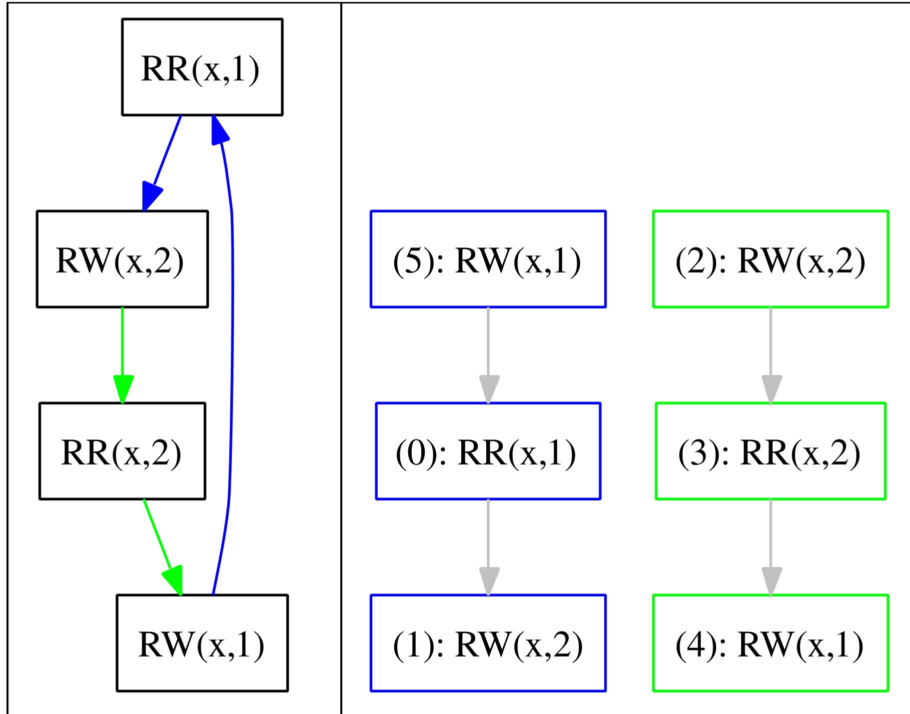



Figure 4: The two possible ways to display all orderings in one graph.

though all threads need to observe all calls to the same collective it is important to distinguish between the calls, so we have decided to color code these collective calls based on the thread that issues them.

To extract the observed order of thread 0 you must follow the graph from top to bottom visiting all nodes in blue and black, all `upc_notify`'s and all `upc_wait`'s without visiting nodes with any other color. The strict ordering can be extracted by visiting all the black nodes, `upc_notify`s and `upc_wait`s. You can visit other nodes, but they are not a part of the strict common order.

The graph shown to the right in Figure 4 we will get the following orderings. For Thread 0 the observed ordering is RW(x,1) → RR(x,1) → RW(x,2) where the first operation is issued by Thread 1. The observed order for Thread 1 is RW(x,2) → RR(x,2) → RW(x,1) where the first operation is issued by thread 0. There is no common strict order since there are no strict operations in this execution trace. See example 6 (page 35) for an example where there are both relaxed and strict operations in the graph.

# 5 Implementation

The program reads the execution trace and converts it into a set of tuples using a bison/flex parser. The tuples are then fed into the SAT generation program. When all the clauses are generated the SAT solver tries to solve the problem. If a solution is found the program prints a graph showing a possible ordering. If a solution is not found we extract the cycle from the SAT encoding and outputs a graph showing the cycle.

## 5.1 What is SAT and CNF-SAT

*The Boolean satisfiability problem* (SAT) is a decision problem where you have a Boolean expression containing variables and the logical operators $\wedge$ (AND), $\vee$ (OR) and $\neg$ (NOT). The problem is to decide whether there is an assignment of true and false to the variables (a variable assignment) such that the whole Boolean expression is true. If there is such an assignment, the problem is considered satisfiable, otherwise it is considered unsatisfiable.

CNF-SAT is a constrained version of the satisfiability problem. In CNF-SAT the Boolean expression is in *conjunctive normal form* which means that it is a conjunct of disjunct clauses. A disjunct clause is a list of variables that are OR'ed together. A conjunct of these clauses means that these disjunct clauses are AND'ed together. This means if the expression is satisfiable, then every clause must be satisfiable. Therefore if we can represent every constraint as a clause and the expression is satisfiable then every constraint must be satisfiable.

## 5.2 Using SAT To Find Observed Orderings

Despite the fact that SAT (and CNF-SAT) is NP-complete we can solve interesting problems (like ours) in a reasonable amount of time. Extensive research has been done on SAT solvers and as a result many large problems can be solved.

Finding a satisfying variable assignment is a state-space search where a good heuristic function and/or good pruning can drastically reduce the time it takes to find a solution. For a more comprehensive overview of the advances done in SAT solving over the last few years consult [22]. One of the fastest general purpose SAT solvers is zChaff[15, 2].

Due to advancements in SAT solvers and the ability to easily express some problems as Boolean expressions, many problems are reduced to the SAT problem and a SAT solver is used to find a solution.

To do this with our problem, we must first show that it is in NP. This means that a solution to our problem can be verified (check that the solution is valid) in polynomial time. We prove this as follows.

Let $t$ be the number of threads in the execution trace and $n$ be the maximum number of operations for any thread. If we have an ordering for each thread we can check every pair of operations ($n^2$) with every constraint in the memory model (there is a constant number of constraints). In the worst case all operations are observed by all threads giving us a complexity of $O(tn^2)$ which is polynomial. To check if read operations return a legal value, we can simply start at the node representing the read operation and traverse the graph in the opposite direction. If we get to a corresponding write without encountering a conflicting write the read returns a legal value. For each read operation we need to check at most $n - 1$ tuples. Since this has to be done at most $n$ times it is polynomial as well.

## 5.3 Reducing To CNF-SAT

When we take a problem and encode it as another problem we call it a reduction. Since CNF-SAT is an NP-complete problem and our problem is in NP we can reduce it to CNF-SAT. This means that if our problem is legal then the encoded CNF-SAT version is satisfiable and if our problem is not legal the CNF-SAT version will not be satisfiable.

In addition to determining legality we also use the variable assignment from the CNF-SAT solution to determine the observed ordering. Our SAT-solver of choice (zChaff) also gives us a list of unsatisfiable clauses if the entire expression is unsatisfiable. We use these clauses to find the tuples that cause the execution trace to be illegal.

## 5.4 SAT encoding

As shown in [20], using a SAT solver to find a legal ordering for an execution trace is possible. The reduction to CNF-SAT is very intuitive because the question whether one operation should be ordered before another is the same as one variable in SAT is true or false. We use two variables for each pair of operations ($i$ and $j$), one for "$i$ must be observed before $j$" ($\mathbf{order}\ i\ j$) and one for "$j$ must be observed before $i$" ($\mathbf{order}\ j\ i$)[6]. If the memory model specifies that two oper-

[6] We could have used only one variable, but that would imply that every two operations must be ordered in some way. But we know that for example relaxed reads are only part of the observed

ations cannot be observed at the same time we must make sure that one operation is observed before the other. In SAT we encode this as $\mathbf{order}\ i\ j \vee \mathbf{order}\ j\ i$. This encoding ensures that $i$ is observed before $j$ or $j$ is observed before $i$. Since most of the constraints in the UPC memory model are written like this they are easy to translate into SAT.

To make sure that every read returns a legal value it must refer to a write that writes that value. We use the variable $\mathbf{reads}\ a\ b$ for this relationship. There is one of these variables for every pair of corresponding reads and writes.

## 5.5 UPC Memory Model Constraints In First Order Logic

The UPC memory model is already stated as a set of Boolean expressions, but these expressions only show the constraints on how a thread can observe operations. Since we want to encode the observed order for all the threads at once, we need to modify those expressions a little. As a basis for our constraints we use the predicate $\mathbf{order}\ i\ j$ as explained above and predicate $\mathbf{reads}\ i\ j$ to say that tuple $i$ (a read) gets its value from tuple $j$ (a write). Both these predicates are represented by variables in the SAT encoding.

$tuples$ is the complete set of tuples and $order$ is the set of ordering predicates (the solution). $Threads$ is the total number of threads.

### 5.5.1 Helper Functions

To make the constraints more compact and intuitive we have used some helper functions.

**isRead** $i \equiv$ **op** $i = Read$
**isWrite** $i \equiv$ **op** $i = Write$
**isStrict** $i \equiv$ **cons** $i = Strict$
**isRelaxed** $i \equiv$ **cons** $i = Relaxed$

**observed** $tuples\ thr \equiv \{i \in tuples \mid$ **obs** $i = thr \vee$ **isStrict** $i\}$

Same as above only for tuples observed by $thr$.

ordering for the issuing thread and should therefore not be ordered with operations not in that ordering.

### 5.5.2 Basic Constraints

These are constraints which are not explicitly mentioned in the UPC memory model, but are implied by the description of what an ordering is and what must be true for the ordering to be legal.

**requireTransitiveOrder** $tuples\ order \equiv \forall i, j, k \in tuples.$
  (**order** $i\ j \wedge$ **order** $j\ k) \Rightarrow$ **order** $i\ k$

If $i$ is ordered before $j$ and $j$ is ordered before $k$, then that means that $i$ must also be ordered before $k$.

**requireIrreflexiveOrder** $tuples\ order \equiv \forall i \in tuples.\ \neg($**order** $i\ i)$

Operation $i$ cannot be ordered before itself.

**requireDeterministicOrder** $tuples\ order \equiv \forall t \in$ Threads $\ \forall i, j \in tuples$
$((\mathbf{obs}\ j = t) \wedge (\mathbf{obs}\ i = t)) \vee ((\mathbf{isStrict}\ i) \vee (\mathbf{isStrict}\ j))$
  $\Rightarrow$ **order** $i\ j \vee$ **order** $j\ i$

A thread cannot observe two operations at the same time so one operation must have been observed before the other. The **obs** value specifies the observing thread, but since strict operations are observed by all threads the constraint also makes sure that tuples are ordered, in some way, with respect to strict operations. This also assures that all strict operations are ordered with respect to each other. The last constraint makes sure that all operations observed by one thread are ordered in some way and the first two make sure that there are no cycles.

### 5.5.3 UPC Constraints

### 5.5.4 requireProgramOrder

Some of the operations (specified by dependOnThreads) must not be reordered according to the program order (program counter).

**requireProgramOrder** $tuples\ order \equiv \forall i\ j \in tuples.$
  **dependOnThreads** $i\ j\ \Rightarrow$ **order** $i\ j$

The operations that cannot be reordered are operations which operate on the same variable where one is a write[7]. No operations can be reordered with a strict operation on the same thread [8].

$\textbf{dependOnThreads}\ i\ j \equiv (\mathbf{pc}\ i < \mathbf{pc}\ j) \wedge (\textbf{iss}\ i = \textbf{iss}\ j)$
$\quad \wedge(\textbf{conflicting}\ i\ j\ \vee \textbf{strictOnThreads}\ i\ j)$

$\textbf{conflicting}\ i\ j \equiv \textbf{var}\ i = \textbf{var}\ j\ \wedge (\textbf{isWrite}\ i\ \vee \textbf{isWrite}\ j)$

$\textbf{strictOnThreads}\ i\ j \equiv (\textbf{isStrict}\ i\ \vee \textbf{isStrict} j)$

If an operation cannot be reordered with an operation on the issuing thread, then those two operations must be observed in the same way on other threads that observe both of those operations.

$\textbf{collectiveOrder}\ tuples\ order \equiv \forall i, j \in tuples$
$(\mathbf{op}\ i = \mathbf{notify}) \wedge (\mathbf{op}\ j = \mathbf{wait}) \wedge (\mathbf{src}\ i = \mathbf{src}\ j)$
$\Rightarrow \mathbf{order}\ i\ j$

From the definition of the split-phase barrier no operations can be reordered with `upc_notify` or `upc_wait`(as if they were strict operations), all operations issued before a `upc_notify` must be observed before the `upc_wait` on other threads and all operations issued after a `upc_wait` must be observed after `upc_notify` on all other threads. The first part is taken care of by requireProgramOrder since `upc_notify` and `upc_wait` are considered strict operations. The remaining two are taken care of by collectiveOrder where each `upc_notify` must be ordered before all other (matching) `upc_wait`s.

$i$ and $j$ could be on the same thread, but in that case this constraint would not matter since they would already be ordered by program order.

For `upc_barrier` we do a little trick. Instead of making constraints on all the tuples that represent a `upc_barrier` we create constraints only on the original tuple. When, for example, requireProgramOrder creates a constraint between an operation and a call to the barrier operation based on its program counter, we replace the duplicate tuple with the original tuple in the order predicate. This way we do not have to create any constraints between the `upc_barrier` tuples, we just treat it as a regular strict operation which has a different **pc** on each thread.

[7]Identified by SameVariable in the graph.
[8]Identified by StrictOnThread in the graph.

### 5.5.5 requireReadValue

Since every thread must be able to explain a value returned by a read, we must apply this constraint on one thread at a time.

$$\begin{aligned}
&\textbf{requireReadValue}\ \textit{tuples order} \equiv \forall\, t \in THREADS \\
&\quad \forall\, i, j, k \in (\textbf{observed}\ \textit{tuples}\ t) \\
&\quad \textbf{isRead}(k)\wedge \\
&\quad \textbf{isWrite}(i) \wedge (\textbf{var}\ i = \textbf{var}\ k) \wedge (\textbf{value}\ i = \textbf{value}\ k)\wedge \\
&\quad \textbf{isWrite}(j) \wedge (\textbf{var}\ j = \textbf{var}\ k) \wedge (\textbf{value}\ j \neq \textbf{value}\ i) \\
&\quad \Rightarrow \neg\textbf{reads}\ k\ i\ \vee \textbf{order}\ j\ i \vee \textbf{order}\ k\ j
\end{aligned}$$

If $k$ reads from $i$ all conflicting writes must be ordered before $i$ or after $k$.

$$\begin{aligned}
&\textbf{requireReadOrder}\ \textit{tuples order} \equiv \forall k, i \in \textit{tuples} \\
&\quad \textbf{isRead}(k)\wedge \\
&\quad \textbf{isWrite}(i) \wedge (\textbf{var}\ i = \textbf{var}\ k) \wedge (\textbf{value}\ i = \textbf{value}\ k)\wedge \\
&\quad \Rightarrow \neg\textbf{reads}\ k\ i \vee \textbf{order}\ i\ k
\end{aligned}$$

If $k$ reads from $i$ then $i$ must come before $k$.

If there is an initial value for the variable in question then the read can refer to this and $\neg\textbf{reads}\ k\ \textit{init} \vee \textbf{order}\ k\ j$ is created instead to force all conflicting writes to be observed after $k$.

For every read operation we create a clause where one of the $\textbf{reads}\ k\ i$ predicates must be true.

The sum of these constraints forces every read to get its value from a write that is observed before the read. All other writes to the same location that writes a different value must be observed before the write or after the read. $\textbf{reads}\ k\ i$ can be true for more than one $i$ for one $k$, but since every $i$ writes the same value it does not matter (the only constraint we want to apply is that there exists at least one).

## 5.6 Clause Generation

For every constraint we find tuples that satisfy the regular constraints and create a clause based on the implication.

## 5.7 Extracting Orderings from SAT Results

As mentioned in the Output Section we use the variable assignment to create a graph for a legal execution trace. For an illegal execution trace we need to do a little extra work because all we get from the solver is the clauses that cannot be satisfied.

We can easily check the clauses for the tuples that are involved in the cycle. The edges between the tuples depend on what kind of clauses we get.

At least one of the clauses will be from the irreflexive constraint just because we got a cycle, but we do not need to create any edges because of this clause since we can see that there is a cycle.

For the requireProgramOrder and collectiveOrder constraints we have enforced an order between pairs of tuples so we create edges for these clauses.

If there is a conflicting write that is forced between a read and its corresponding write, we create edges from the read to the conflicting write and from the conflicting write to the corresponding write (since the conflicting write should have been observed before the write or after the read). In the case where a read returns the initial value there will only be one edge, an edge from the read to the conflicting write.

If no corresponding write can be observed before a read, there will be one cycle for each possible corresponding write, explaining why neither of them cannot be ordered before the read. In this case we have a edge from each of the possible corresponding writes to the read.

# 6 Future Work

Many ideas of additons and modifications to this work has been discovered while creating our tool. We would like to describe a few of them here.

Even though we have shown that this tool makes it easier to reason with small execution traces, it would be interesting to see how it scales to bigger problems, not only in computation time, but also in understandability. Based on these results we may change the output to improve understandability in both small and large problems.

The input is now based on a manual conversion from source code/output to an execution trace. One way to improve this tool would be to minimize this gap or maybe fully automate it.

It is a known fact that memory models are notoriously hard to understand, partly because there are several ways of specifying and thinking about them. It would be valuable to be able to switch between non-operational and operational views.

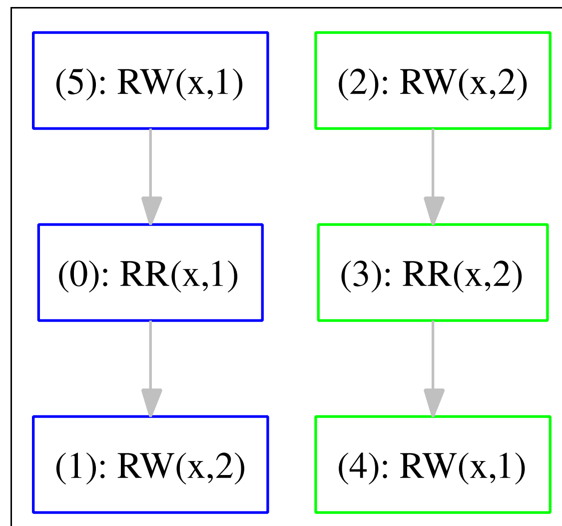
# 7 Examples

All the examples used in the memory model to show the desired behavior can be found in this section. Each example is shown with an execution trace and the solution graph given from our program.

The operations in the graphs are numbered with an id. This id is not unique since relaxed write operations are duplicated. Refer to the execution trace to see which of the threads issued the operation.

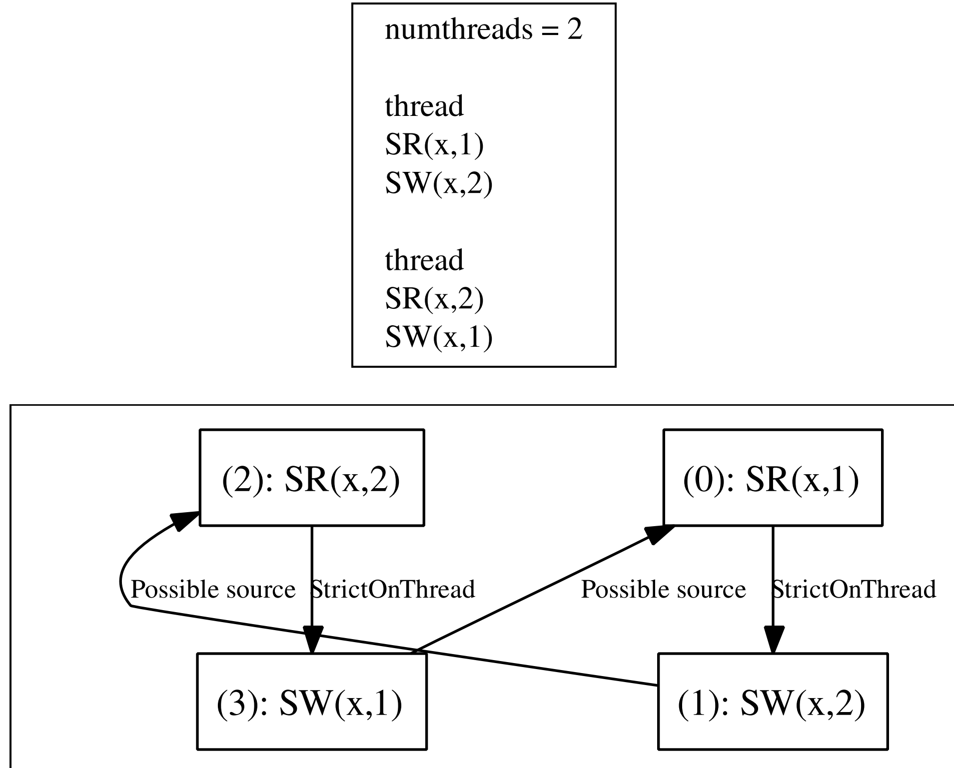
## 7.1 Example 1

```
startvalues
x = 0

numthreads = 2

thread
RR(x,1)
RW(x,2)

thread
RR(x,2)
RW(x,1)
```



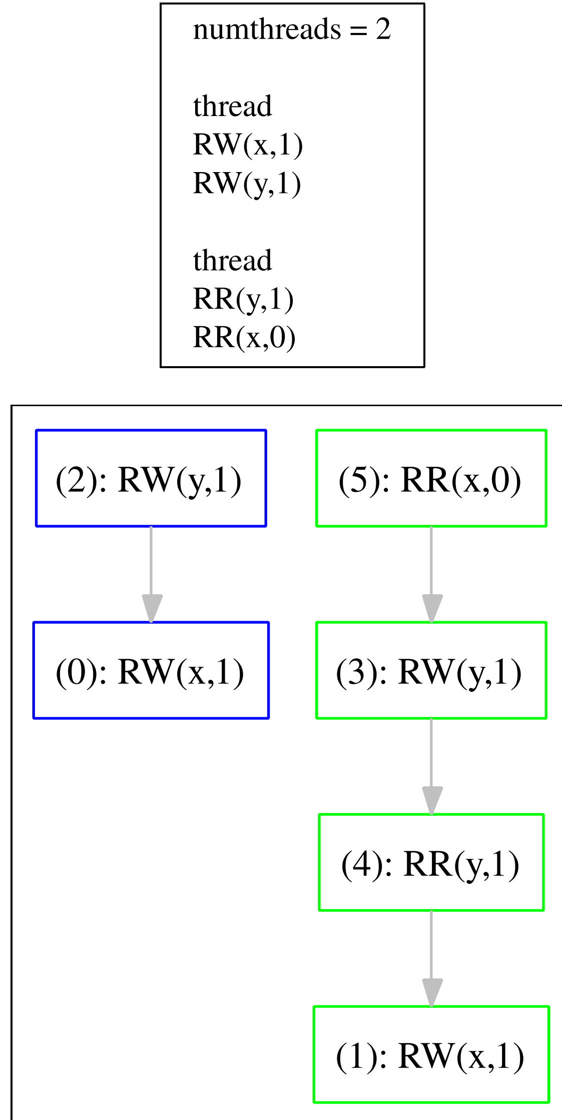

From the solution we can see that we have a legal execution where both threads observes the other threads write before its own operations. We have no strict operations so there is no common strict order.

## 7.2 Example 2



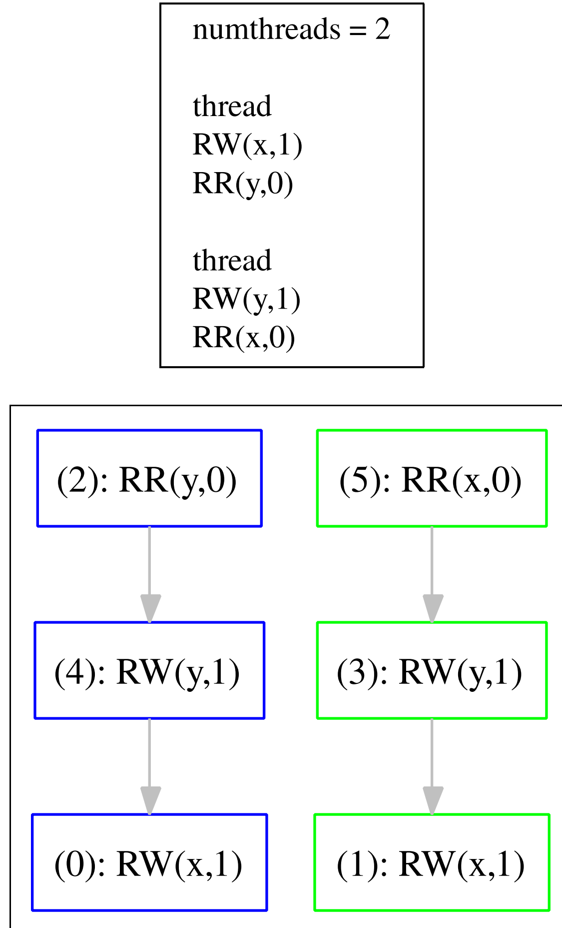

This execution trace is not legal because we have a cycle in the solution. From the graph we can see that (1) has to be the source of (2) and (3) has to be the source of (0). Both of these constraints cannot be satisfied because strict operations cannot be reordered on a thread (StrictOnThread).

## 7.3 Example 3



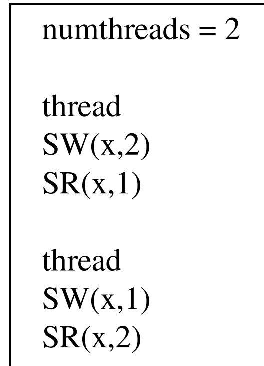

This execution trace is legal because thread 1 can observe the writes from thread 0 in a different order than thread 1 because they are relaxed operations and they operate on different memory locations.

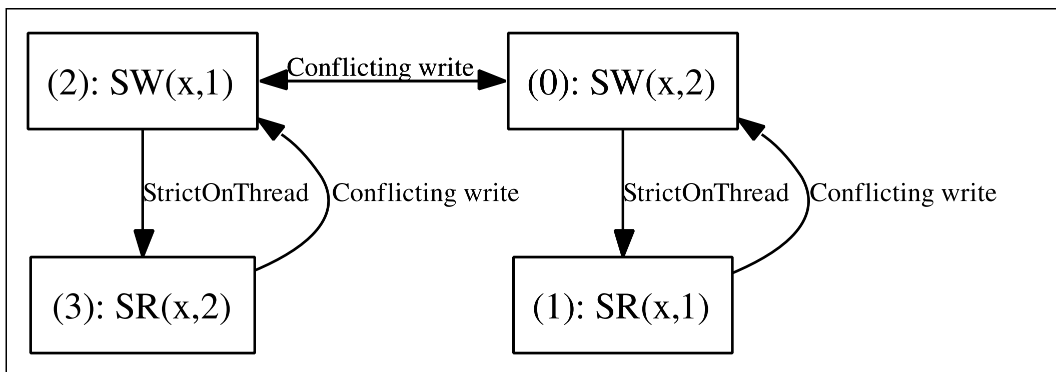
## 7.4 Example 4




Here we can see that each read happens after a corresponding write. This is possible because all operations are relaxed and operating on different memory locations and can be observed in any order.

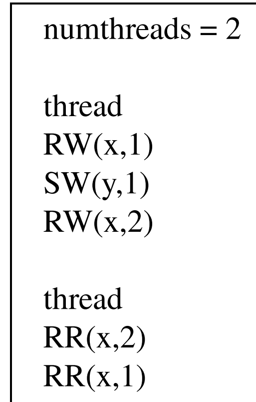
## 7.5 Example 5

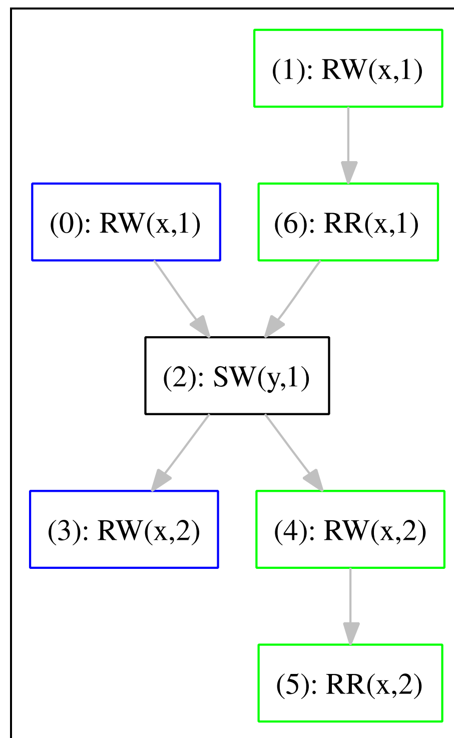



Even though each read is preceded by a corresponding write, one of the writes must come before the other causing one of them to conflict with the other read.

## 7.6 Example 6

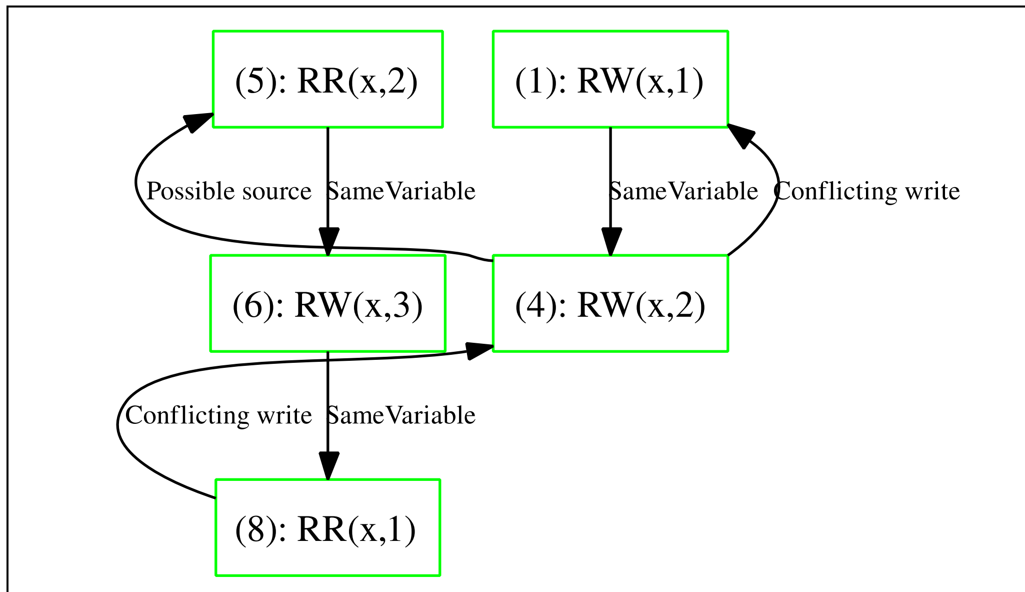



A legal execution where thread 1 is observing its relaxed reads out of order. Note that thread 0 is observing only its own operations and that they are all observed in program order. Thread 1 is observing the relaxed writes from thread 0 on the same side of the strict operation because of the strict on thread constraint.

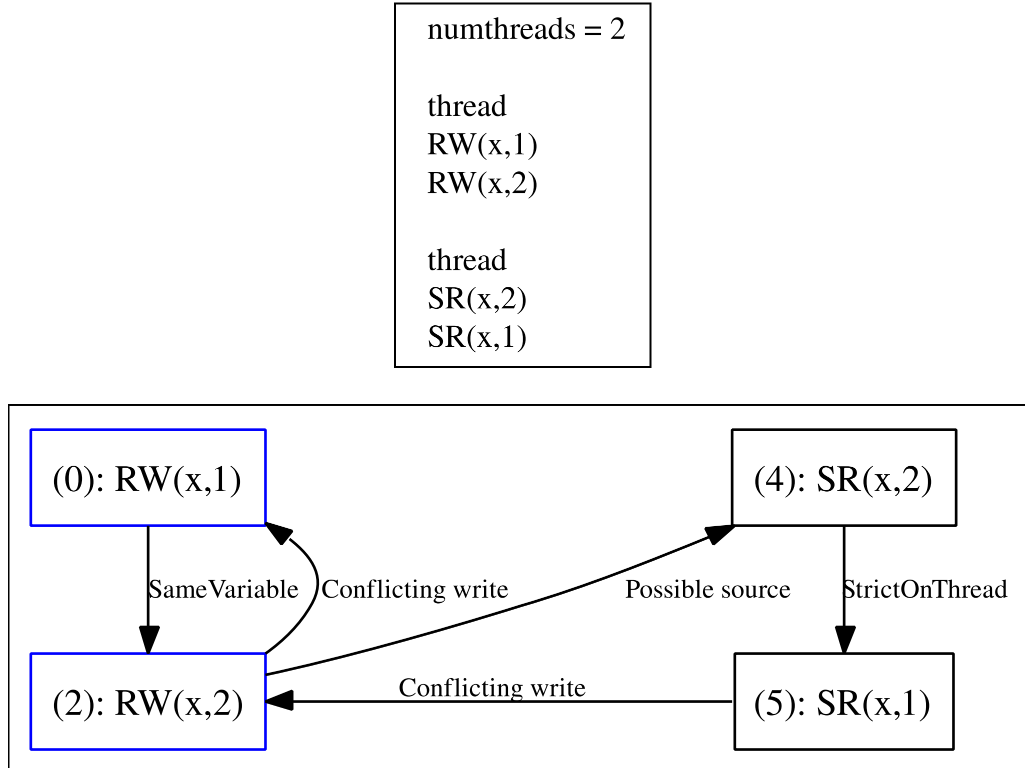
## 7.7 Example 7

```
numthreads = 2

thread
RW(x,1)
SW(y,1)
RW(x,2)

thread
RR(x,2)
RW(x,3)
RR(x,1)
```



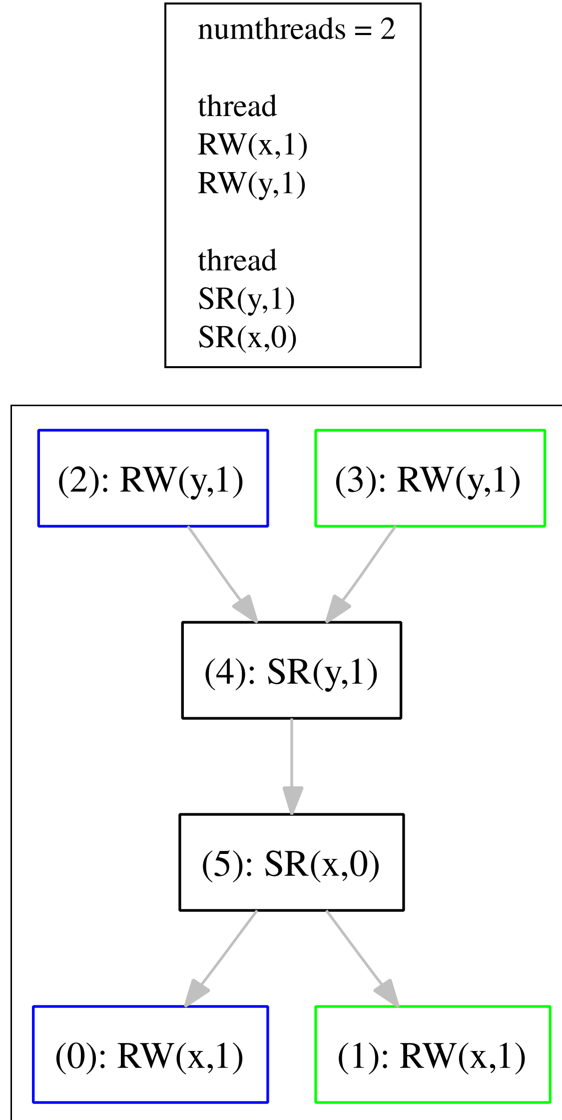

This execution trace is not legal. From the solution we can see that we only have a problem with the observed order for thread 1. (4) is the only possible source for (5) and (1) is the only possible source for (8) and since (1) and (4) cannot be reordered because they operate on the same variable, this is not legal.

## 7.8 Example 8



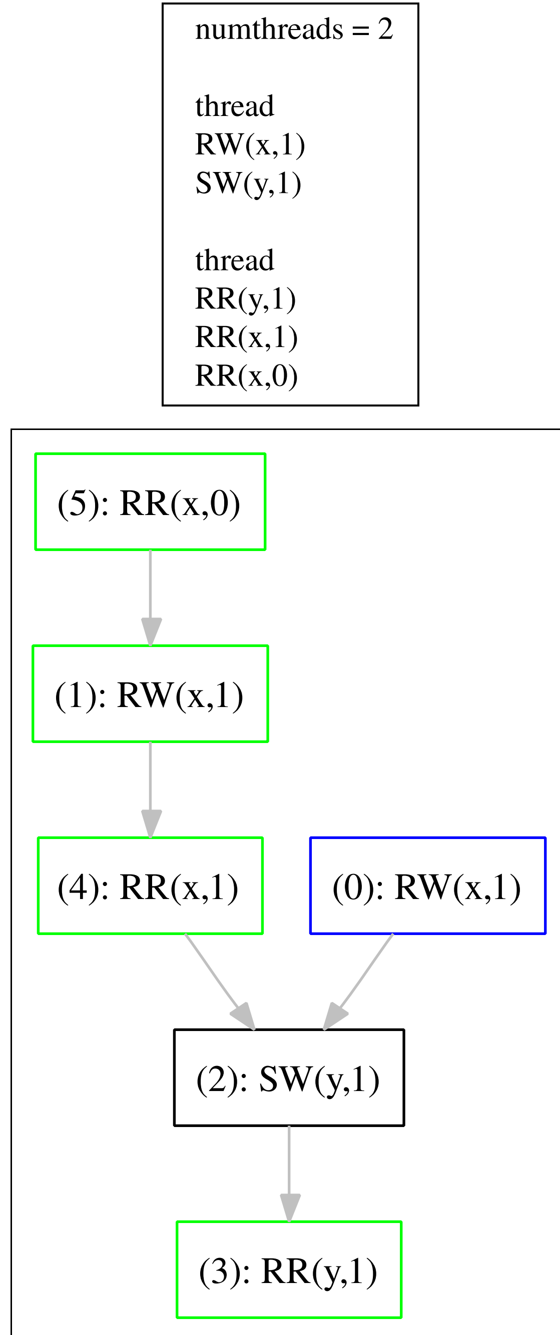

Another execution trace that is not legal. Here we have the same problem as with Example 7 (page 36) where the two writes must be ordered according to program order because they write to the same memory location which causes (2) to conflict with (0) being the source of (5).

## 7.9 Example 9




Now the writes are no longer operating on the same memory location and can therefore be observed out of order, making this a legal execution trace.

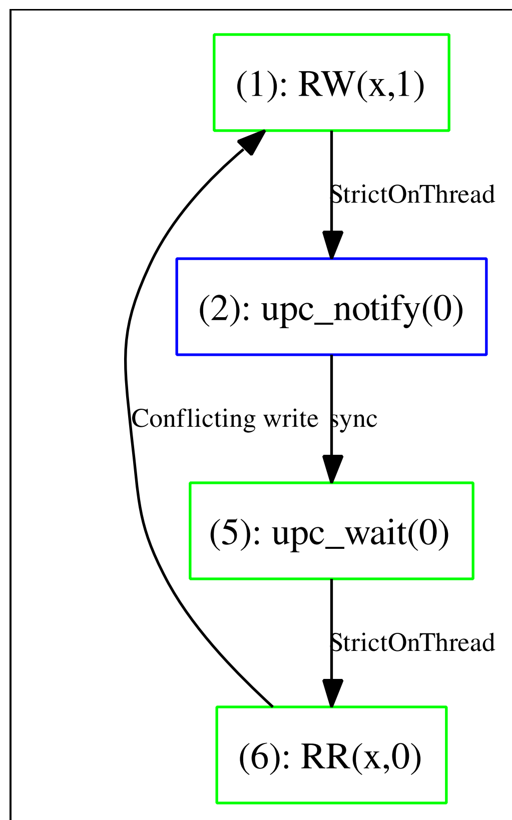
## 7.10 Example 10




This is legal because thread 1 can observe its reads out of order.

## 7.11 Example 11a

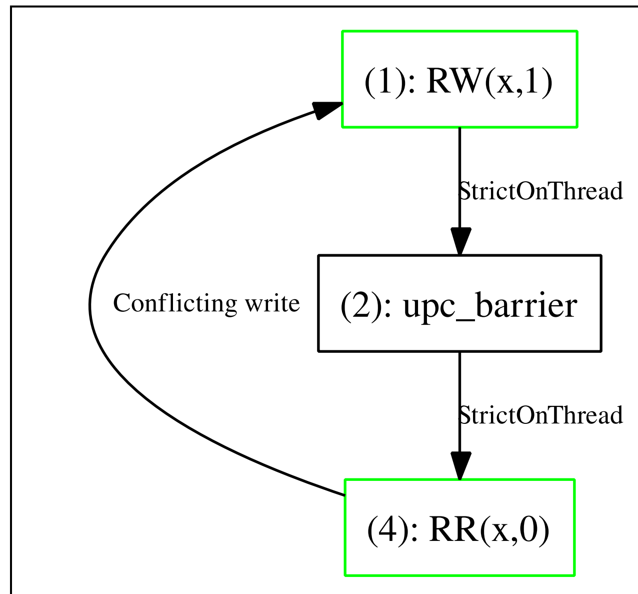
```
numthreads = 2

thread
RW(x,1)
upc_barrier

thread
upc_barrier
RR(x,0)
```




This executiontrace is illegal because the write must be observed before the `upc_barrier` on both threads and will therefore conflict with the read on thread 1.

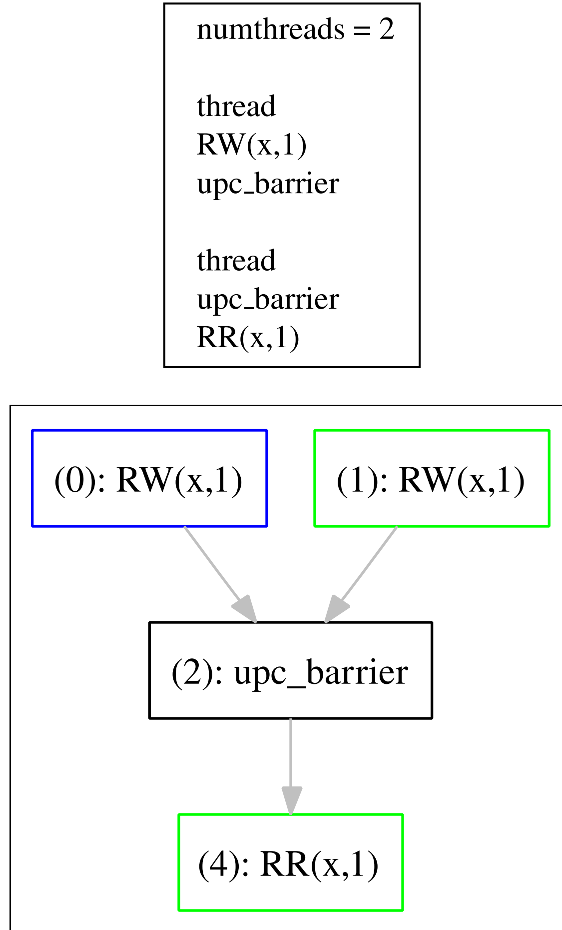
## 7.12 Example 11b

```
numthreads = 2

thread
RW(x,1)
upc_notify
upc_wait

thread
upc_notify
upc_wait
RR(x,0)
```




This is not legal because (1) must be observed before (5) on both threads.

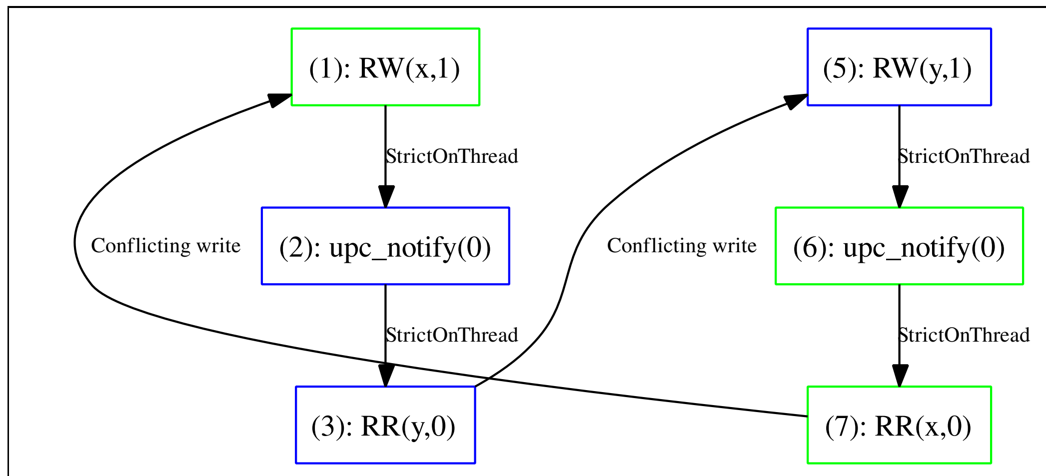
## 7.13 Example 11c




This *is* legal because the read now corresponds with the write issued by thread 0.

## 7.14 Example 12

```
numthreads = 2

thread
RW(x,1)
upc_notify
RR(y,0)

thread
RW(y,1)
upc_notify
RR(x,0)
```




This is not legal because we can only satisfy one of (3) and (7). The other operation would read an illegal value which would make the entire execution trace illegal.

# References


[1] http://www.graphviz.org.

[2] http://www.princeton.edu/˜chaff/zchaff.html.

[3] A formal specification of intel itanium processor family memory ordering. Application Note, Document Number: 251429-001, October 2002.

[4] Arvind and Xiaowei Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3):36–46, May/June 1999.

[5] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of Design Automation Conference*, 1999.

[6] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 297–308, 1996.

[7] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan. Shared memory consistency protocol verification against weak memory models: refinement via model-checking, 2002.

[8] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[9] Guang R. Gao and Vivek Sarkar. Location consistency-a new memory model and cache consistency protocol. *IEEE Transactions on Computers*, 49(8):798–813, 2000.

[10] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.

[11] J. R. Goodman. Cache consistency and sequential consistency. Technical report, Computer Science Dept., U. of Wisconsin-Madison, 1989.

[12] William Kuchera and Charles Wallace. The upc memory model: Problems and prospects. IPDPS, 2004.

[13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 9(29):690–691, 1979.

[14] K. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–51, 1991.

[15] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. 39th Design Automation Conference (DAC 2001), Las Vegas, 2001.

[16] Shaz Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. *IEEE Transactions on Parallel and Distributed Systems*, 14(8):730–741, August 2003.

[17] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-reconcile & fences: A new memory model for architects and compiler writers. In *Proceedings of the 26th International Symposium on Computer Architectures, Atlanta, Georgia*, May 1999.

[18] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-reconcile & fences (CRF): A new memory model for architects and compiler writers. In *ISCA*, pages 150–161, 1999.

[19] The UPC Consortium. *UPC Language Specifications V.1.2*, May 2005.

[20] Yue Yang. *Formalizing Shared Memory Consistency Models for Program Analysis*. PhD thesis, The University of Utah, 2004.

[21] Katherine Yelick, Dan Bonachea, and Charles Wallace. A proposal for a upc memory consistency model, v1.0. Technical Report LBLN-54983, Lawrence Berkeley National Lab, May 2004.

[22] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proc of CAV'2002*, pages 582–595, 2002.