N3132=10-0122 Revision: 5190 August 23, 2010

Mathematizing C++ Concurrency: The Post-Rapperswil Model

Mark Batty Scott Owens Susmit Sarkar Peter Sewell Tjark Weber

University of Cambridge

`http://www.cl.cam.ac.uk/users/pes20/cpp`

Abstract

In this paper we describe a rigorous semantics for C++ concurrency. To the best of our knowledge, this captures the intent of the Final Committee Draft (N3092) text, modified as discussed at the Rapperswil meeting of the C++ Standards Committee in July 2010. We discuss some issues with the N3092 text that led to those changes.

To make our semantics mathematically precise and unambigous, we express it in machine-formalised mathematics, in the Isabelle/HOL proof assistant. To make it accessible, we introduce it with a series of examples, and give both an English-prose translation of the definitions and a typeset version of the mathematics, side-by-side; it should be possible to read either one in isolation. To make it possible to explore the consequences of the semantics, we have developed a tool (CPPMEM) that calculates the allowed executions of litmus-test example programs (using checking code automatically generated from our Isabelle/HOL definitions, for high assurance).

We further validate the semantics by proving that a proposed x86 implementation of the concurrency primitives is correct with respect to the x86-TSO memory model.

We hope that this will aid discussion of any further changes to the draft standard, provide an unambiguous correctness condition for compilers, and give a much-needed basis for analysis and verification of concurrent C and C++ programs.

Contents

1. Introduction

1.1 Context

C and C++ are defined by standards, but those standards have historically not covered the behaviour of concurrent programs, motivating an ongoing effort to specify concurrent behaviour in a forthcoming revision of C++ (unofficially, C++0x) [AB10, BA08, Bec10]. The next C standard (unofficially, C1X) [C1X] is expected to follow suit.

The key issue here is the multiprocessor relaxed-memory behaviour induced by hardware and compiler optimisations. The design of such a language involves a tension between usability and performance: choosing a very strong memory model, such as sequential consistency (SC) [Lam79], simplifies reasoning about programs but at the cost of invalidating many compiler optimisations, and of requiring expensive hardware synchronisation instructions (e.g. fences). The C++0x design resolves this by providing a relatively strong guarantee for typical application code together with various *atomic* primitives, with weaker semantics, for high-performance concurrent algorithms. Application code that does not use atomics and which is race-free (with shared state properly protected by locks) can rely on sequentially consistent behaviour; in an intermediate regime where one needs concurrent accesses but performance is not critical one can use *SC atomics*; and where performance is critical there are *low-level atomics*. It is expected that only a small fraction of code (and of programmers) will use the latter, but that code —concurrent data structures, OS kernel code, language runtimes, GC algorithms, etc.— may have a large effect on system performance. Low-level atomics provide a common abstraction above widely varying underlying hardware: x86 and Sparc provide relatively strong TSO memory [SSO^{+}10, Spa];

Power and ARM provide a weak model with cumulative barriers [Pow09, ARM08, AMSS10]; and Itanium provides a weak model with release/acquire primitives [Int02]. Low-level atomics should be efficiently implementable above all of these, and prototype implementations have been proposed, e.g. for x86 [Ter08].

The current draft standard covers all of C++ and is rather large (1357 pages), but the concurrency specification is mostly contained within three chapters [Bec10, Chs.1, 29, 30]. As is usual for industrial specifications, it is a prose document. Mathematical specifications of relaxed memory models are usually either operational (in terms of an abstract machine or operational semantics, typically involving explicit buffers etc.) or axiomatic, defining constraints on the relationships between the memory accesses in a complete candidate execution, e.g. with a happens-before relation over them. The draft concurrency standard is in the style of a prose description of an axiomatic model: it introduces various relationships, identifying when one thread *synchronizes with* another, what a *visible side effect* is, and so on (we introduce these in §2), and uses them to define a happens-before relation. It is the result of extensive and careful deliberation. but (almost inevitably, for a prose text) is still rather far from a completely clear and rigorous definition: there are points where the text is unclear, places where it does not capture the intent of its authors, points where a literal reading of the text gives a broken semantics, and some open questions. Moreover, the draft is very subtle. For example, driven by the complexities of the intended hardware targets, the happens-before relation it defines is intentionally non-transitive. The bottom line is that, given just the final committee draft standard text, the basic question for a language definition, of what behaviour is allowed for a specific program, can be a matter for debate.

Given previous experience with language and hardware memory models, e.g. for the Java Memory Model [Pug00, MPA05, CKS07, vA08, TVD10] and for x86 multiprocessors [SSZN$^+$09, OSS09, SSO$^+$10], this should be no surprise. Prose language definitions leave much to be desired even for sequential languages; for relaxed-memory concurrency, they almost inevitably lead to ambiguity, error and confusion. Instead, we need rigorous (but readable) mathematical semantics, with tool support to explore the consequences of the definitions on examples, proofs of theoretical results, and support for testing implementations. Interestingly, the style of semantics needed is quite different from that for conventional sequential languages, as are the tools and theorems.

1.2 Contributions

The model In this paper we establish a mathematically rigorous semantics for C++ concurrency. It is *precise*, formalised in Isabelle/HOL [Isa], and is rather *complete*, covering essentially all the concurrency-related semantics from the draft standard, without significant idealisation or abstraction. It includes the data-race-freedom (DRF) guarantee of SC behaviour for race-free code, locks, SC atomics, the various flavours of low-level atomics, and fences. It covers initialisation but not allocation, and does not address the non-concurrent aspects of C++. Our model builds on the informal-mathematics treatment of the DRF guarantee by Boehm and Adve [BA08]. We have tried to make it as *readable* as possible, using only minimal mathematical machinery (mostly just sets, relations and first-order logic with transitive closure) and introducing it with a series of examples. Finally, wherever possible it is a *faithful* representation of the draft standard and of the intentions of its authors, as far as we understand them.

Issues with previous drafts of the standard In developing our semantics, we identified a number of issues in several drafts of the C++0x standard, discussed these with members of the concurrency subgroup, and made several suggestions for changes. These are of various kinds, ranging from editorial clarifications, substantive changes to the text that are in line with the authors' intent as we understand it, and some open questions. We discuss a selection of these in Section 5.

Tool support for exploring the model Experience shows that tool support is needed to work with an axiomatic relaxed memory model, to develop an intuition for what behaviour it admits and forbids, and to explore the consequences of proposed changes to the definitions. At the least, such a tool should take an example program, perhaps annotated with constraints on the final state or on the values read from memory, and find and display all the executions allowed by the model. This can be combinatorially challenging, but for C++ it turns out to be feasible, for typical test examples, to enumerate the possible witnesses. We have therefore built a CPPMEM tool (§7) that exhaustively considers all the possible witnesses, checking each one with code automatically generated from the Isabelle/HOL axiomatic model (§6). The front-end of the tool takes a program in a fragment of C++ and runs a symbolic operational semantics to calculate possible memory accesses and constraints. We have also explored the use of a model generator (the SAT-solver-based Kodkod [TJ07], via the Isabelle Nitpick interface [BN10]) to find executions more efficiently, albeit with less assurance. Most of the examples in this paper have been checked (and their executions drawn) using CPPMEM.

Correctness of compilation to x86 As a theoretical test of our semantics, we prove a correctness result (§8) for the proposed x86 implementation of the C++ concurrency primitives [Ter08] with respect to our x86-TSO memory model [SSO$^+$10, OSS09]. We show that any x86-TSO execution of a translated C++ candidate execution gives behaviour that the C++ semantics would admit, which involves delicate issues about initialisation. This result establishes some confidence in the model and is a key step towards a verified compilation result about translation of programs.

Applications Our work provides a basis for improving both standards, both by the specific points we raise and by giving a precisely defined checkpoint, together with our CPPMEM tool for exploring the behaviour of examples in our model and in variants thereof. The C and C++ language standards are a central interface in today's computational infrastructure, between what a compiler (and hardware) should implement, on the one hand, and what programmers can rely on, on the other. Clarity is essential for both sides, and a mathematically precise semantics is a necessary foundation for any reasoning about concurrent C and C++ programs, whether it be by dynamic analysis, model-checking, static analysis and abstract interpretation, program logics, or interactive proof. It is also a necessary precondition for work on compositional semantics of such programs.

2. Informal introduction to the model

Here we describe C++ concurrency informally. In this section we do not distinguish between the C++ Final Committee Draft standard, which is the work of the Concurrency subcommittee of WG21, and our formal model, but in fact there are substantial differences between them. We highlight some of these (and our rationale for various choices) in Section 5.

2.1 Overall structure of the model

The memory model determines the set of *executions* that are allowed for a given C++ program. This is defined in several steps:

1. We first enumerate the possible control-flow paths through the program, without taking the memory model into account. For each, we consider the possible sets of *actions* performed on

that path: the reads, writes, locks, unlocks and fences, together with three relations over those actions: *sequenced-before*, *data-dependency*, and *additional-synchronized-with*.

2. For each of those possibilities, we enumerate all the possible choices of three further ("witness") relations: *rf* (a reads-from map), *modification-order* (a coherence order), and *sc* (an order over sequentially consistent actions). Together, this enumerates the *candidate executions*, each of which comprises a set of actions together with the six relations mentioned.
3. For each candidate execution, we calculate several *derived relations* (*synchronizes-with*, *happens-before*, etc.).
4. This is used to determine whether each candidate execution (seperately) is *consistent*.
5. Finally, we check whether any consistent candidate execution of the program exhibits a data race, an unsequenced race, or an indeterminate read. If any does, then the program has undefined behaviour, otherwise its semantics is the set of all its consistent executions.

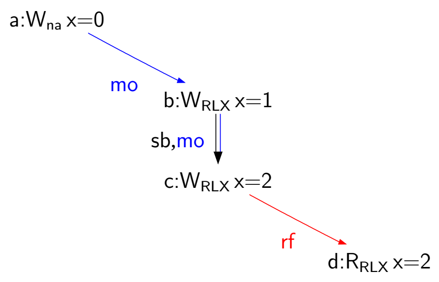
There is a list of all the relations in §6.1 on page 14. In this section we provide example programs and executions that illustrate the key relations of the model, without giving all the details of the definition. We start with a simple example that introduces some new source code syntax and our representation of executions.

2.2 Program and execution example

For brevity, the code examples in this document are written in a mild extension of C++: thread creation gives rise to many memory actions (for passing function arguments and writing and reading the thread id) which clutter examples, so here we use a more concise parallel composition, written `{{{ ... ||| ... }}}`. The main thread of the following example uses this parallel composition syntax to spawn two new threads: one that writes `x` twice and another that reads `x` once.

```
int main() {
  atomic_int x = 0;
  {{{ { x.store(1,mo_relaxed);
        x.store(2,mo_relaxed); }
  ||| { printf("%d\n",x.load(mo_relaxed)); } }}}
  return 0; }
```

One consistent execution of this code is shown below. The actions are drawn as the vertices of a graph, and the edges on the graph correspond to relations over the actions. In this execution there are four actions spanning three threads (in the three columns). The diagram shows three kinds of edge: *modification-order* (mo), *sequenced-before* (sb) and *rf* (rf). We explain the meaning of these in the rest of this section (and also that of the other edges of the execution, which are suppressed here).



../examples/t40-mo-asw-sb-rf.c

Actions are reads, writes, locks, unlocks, and fences. Each read and write action has an annotation that decides which ordering relationships it takes part in and consequently which values they may read and write. Regular reads and writes from memory are *non-atomic*. Atomic reads, atomic writes, and fences, have a *memory order* annotation that enable the programmer to choose how strongly they are ordered. Reads, writes, locks and unlocks take place at *locations*, each of which has a *kind* that can be either non-atomic, atomic or mutex. The actions that are performed at each location must agree with the kind (although non-atomic initialization writes are allowed on atomic locations).

The memory actions are written in a concise and regular form made up of the following parts (from left to right): a unique identifier (a, b,. . .) and a colon; then an R (for a read), W (for a write), RMW (for read-modify-write), L or U (for a lock or unlock), or F (for a fence); a subscript abbreviation of a memory order (mo); a location name (x, y,. . .); an equals sign and either a single value (v) or two values separated by a forward slash (for the values read and written by a read-modify-write).

action ::=

	$\mathsf{a{:}R_{na}\ x{=}v}$	non-atomic read
|	$\mathsf{a{:}W_{na}\ x{=}v}$	non-atomic write
|	$\mathsf{a{:}R_{mo}\ x{=}v}$	atomic read
|	$\mathsf{a{:}W_{mo}\ x{=}v}$	atomic write
|	$\mathsf{a{:}RMW_{mo}\ x{=}v1/v2}$	atomic read-modify-write
|	a:L x	lock
|	a:U x	unlock
|	$\mathsf{a{:}F_{mo}}$	fence

Memory orders are shown as follows:

mo ::=

	SC	`memory_order_seq_cst`
|	RLX	`memory_order_relaxed`
|	REL	`memory_order_release`
|	ACQ	`memory_order_acquire`
|	CON	`memory_order_consume`
|	A/R	`memory_order_acq_rel`

Actions also contain a thread identifier, but we usually elide it: in diagrams we usually collect the actions of each thread into a column.

In the rest of this section we will introduce the relations and definitions of the memory model by example. The intention is to provide an intuition about the model rather than to formally define it (the formal definition of the model is in §6).

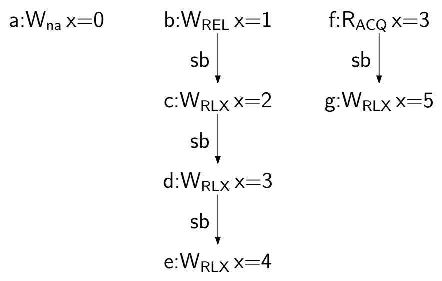
2.3 Relations determined by syntax and control-flow

The *sequenced-before*, *additional-synchronized-with* and *data-dependency* relations of a candidate execution are determined by the syntactic structure of the source code and a choice of the particular path of control flow through each thread of the program.

Sequenced before Sequenced-before relates memory actions according to the order they are written in the code, though it is not necessarily total for each thread because function and operator arguments in C and C++ are not necessarily ordered. We show the sequenced-before edges of one consistent execution of the following example code below.

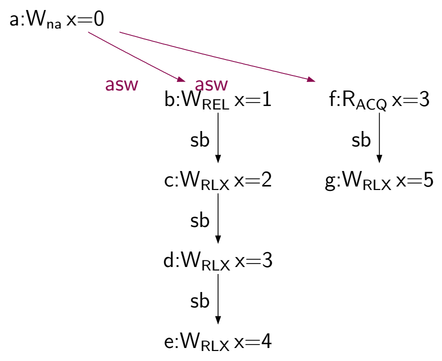
This example also illustrates another extension: when working with memory-model litmus tests, one is usually concerned only with the possible executions of a test in which reads read some particular values. Instead of expressing this with constraints on the values of variables in the final state (which would often require additional locations and memory writes to record them) we annotate reads with the constraint in-line, e.g. with the `.readsvalue(3)` here. As for the parallel construct above, this is simply a convenience for discussing the memory model, not a proposal to extend C++ itself.

```
int main() {
  atomic_int x = 0;
  {{{ { x.store(1,mo_release);
        x.store(2,mo_relaxed);
        x.store(3,mo_relaxed);
        x.store(4,mo_relaxed); }
  ||| { printf("%d\n",x.load(mo_acquire).readsvalue(3) );
        x.store(5,mo_relaxed); } }}}
  return 0; }
```



../examples/t53-sb.c

Additional-synchronizes-with The standard describes several situations that give rise to synchronization edges between actions. The main case (of release/acquire synchronization) is described in the definition of the *synchronizes-with* relation below, and *additional-synchronized-with* collects all of the other cases. We do not list all of these cases here. Instead we give one important example: there are synchronization edges from actions in a parent thread to actions in a thread that it spawns. In a general sense, synchronization edges order one part of a program before another. The following diagram of the previous example execution includes the additional-synchronizes-with edges.

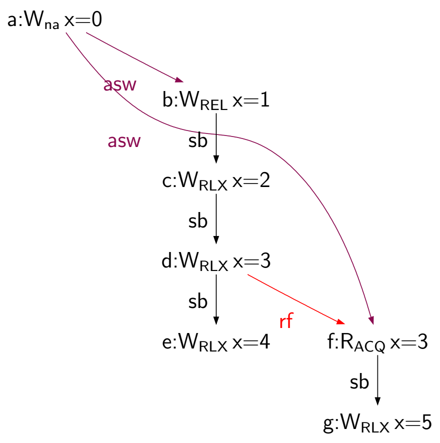


../examples/t54-asw.c

2.4 Witness relations

In addition to its actions and the *sequenced-before*, *additional-synchronized-with* and *data-dependency* relations, a candidate execution also comprises three more relations: *rf*, *sc*, and *modification-order*, which we introduce here. The definition of whether a candidate execution is *consistent* will depend on all these together.

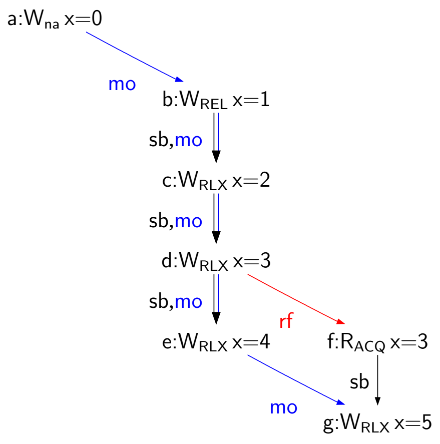
Reads-from When describing how a read reads from a write, the Final Committee Draft uses various terminology, including phrases like "takes the value of" and "observes the value written by". We model this with a *reads-from* relation, relating each write to every read that takes its value from that write. Displaying the reads-from edges of the previous candidate execution, we have:



../examples/t55-rf.c

Note that the *rf* edges relate writes and reads with the same value, as they must in any consistent execution.

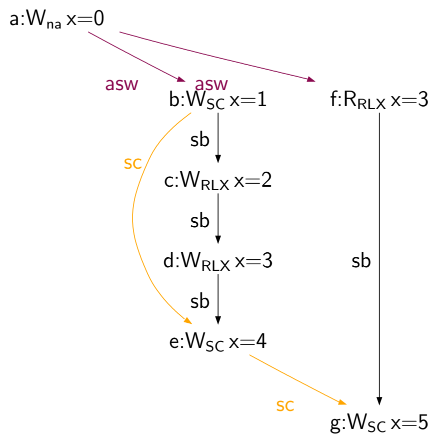
Modification order A candidate execution has, for each atomic location, a total order of the writes to that location. The union of all these relations is *modification-order*. In our example execution, the writes in the two threads are inter-related by modification order.



../examples/t43-mo.c

Sequentially consistent order A candidate execution has a total order, *sc*, over all of the atomic actions of sequentially consistent memory order and all of the lock and unlock actions of the program. In a variant of the previous example program, with two `mo_seq_cst` actions, there is a consistent execution with an *sc* relation as shown below.

```
int main() {
  atomic_int x = 0;
  {{{ { x.store(1,mo_seq_cst);
        x.store(2,mo_relaxed);
        x.store(3,mo_relaxed);
        x.store(4,mo_seq_cst); }
  ||| { printf("%d\n",x.load(mo_relaxed).readsvalue(3) );
        x.store(5,mo_seq_cst); } }}}
  return 0; }
```



../examples/t56-sc.c

Actions of mo_seq_cst memory order related by modification order must be related by *sc* order in the same direction.

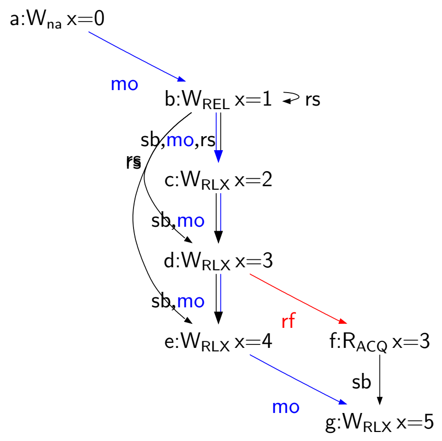
2.5 Derived relations

Summarising, a candidate execution comprises its actions, the location kinds, the relations determined by the syntax and a choice of control-flow (*sequenced-before*, *additional-synchronized-with*, and *data-dependency*), and the witness relations (*rf*, *modification-order*, and *sc*),

Given that, we define various derived relations. These are completely determined by the data of a candidate execution; they will be used in the definition of whether a candidate execution is consistent.

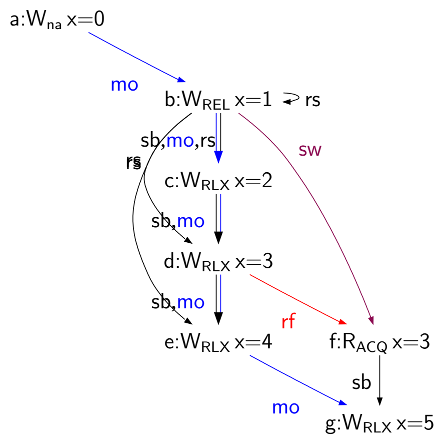
Release Sequence A *release action* is an atomic write or fence that has either $mo\_release$, mo_acq_rel or mo_seq_cst memory order. Each release write has a *release sequence*, a contiguous part of the modification order that starts with the release, where each write in the sequence must be on the same thread as the release or a read-modify write action. Returning to our previous (non-mo_seq_cst) code and our previous execution, below we display the release sequence of the write-release on the first thread, expressing it with an rs edge from the write-release to each element of its release sequence.

```
int main() {
  atomic_int x = 0;
  {{{ { x.store(1,mo_release);
        x.store(2,mo_relaxed);
        x.store(3,mo_relaxed);
        x.store(4,mo_relaxed); }
  ||| { printf("%d\n",x.load(mo_acquire).readsvalue(3) );
        x.store(5,mo_relaxed); } }}}
  return 0; }
```



../examples/t42-rs.c

Synchronizes-with An *acquire action* is an atomic read or fence that has either $mo\_acquire$, mo_acq_rel or mo_seq_cst memory order or a fence of $mo\_consume$ order. A release action *synchronizes with* an acquire action on another thread if the acquire action reads from a write in the release sequence of the release. There is a synchronizes-with (sw) edge between the acquire and release actions in the execution below because the write that the acquire reads from is in the release sequence. (There are also synchronizes-with edges from a to b and f arising from the additional-synchronizes-with edges from thread creation, but they are not shown in this diagram.)

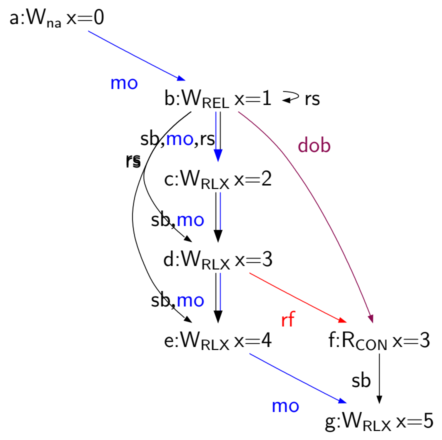


../examples/t44-sw.c

Carries-a-dependency-to and Dependency-ordered-before Dependency-ordered-before is a similar relationship to synchronizes-with but for release/consume pairs. As we shall see later, the inter-thread ordering that synchronizes-with provides extends to all actions in the threads it relates, along sequenced-before, whereas dependency-ordered-before only extends its ordering through data dependence.

A *consume action* is an atomic read that has $mo\_consume$ memory order. A release action is dependency-ordered-before a consume action if the consume action reads from a write in the release sequence of the release. Dependency-ordered-before extends transitively through carries-a-dependency-to, a relation made up of data dependency and the reads-from relation restricted to be thread local. The following example, where f is a consume rather than an acquire, shows an execution with a dependency-ordered-before edge.

```
int main() {
  atomic_int x = 0;
  {{{ { x.store(1,mo_release);
        x.store(2,mo_relaxed);
        x.store(3,mo_relaxed);
        x.store(4,mo_relaxed); }
  ||| { printf("%d\n",x.load(mo_consume).readsvalue(3) );
        x.store(5,mo_relaxed); } }}}
  return 0; }
```



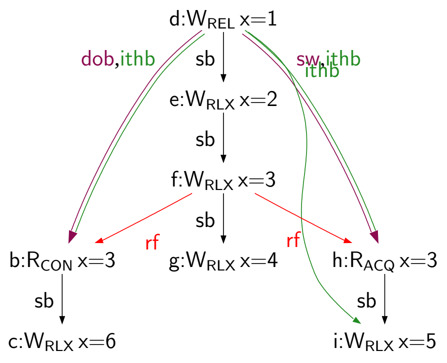
../examples/t45-cad-dob.c

Inter-thread-happens-before and happens-before The main derived relation of the memory model is *happens-before*, an inter-thread relation that collects together several of the ordering relations we have discussed so far. Happens-before is the closest thing to a global time ordering that exists in the memory model, but it is not total or transitive.

Happens-before is defined as the union of a transitive relation called *inter-thread-happens-before* with sequenced before.

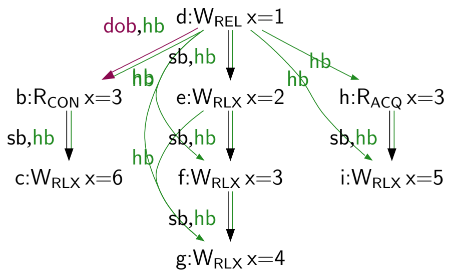
Roughly speaking, inter-thread-happens-before is the transitive closure of the union of synchronizes-with, dependency-ordered-before and sequenced-before, but without the edges resulting from the composition of dependency-ordered-before and sequenced-before edges. The following execution, with a release/consume pair and a release/acquire pair, includes the transitive reduction of inter-thread-happens-before (showing the whole relation makes the diagram too busy) (the initialisation of x has also been suppressed for clarity).

```
int main() {
  atomic_int x = 0;
  {{{ { printf("%d\n",x.load(mo_consume).readsvalue(3) );
        x.store(6,mo_relaxed); }
  ||| { x.store(1,mo_release);
        x.store(2,mo_relaxed);
        x.store(3,mo_relaxed);
        x.store(4,mo_relaxed); }
  ||| { printf("%d\n",x.load(mo_acquire).readsvalue(3) );
        x.store(5,mo_relaxed); } }}}
  return 0; }
```



../examples/t46-ithb.c

We show the full happens-before relation of this candidate execution below (note that there is no hb edge from d to c, despite the presence of edges from d to b and b to c, as this is a release/consume pair and c does not depend on b, whereas there is from d to i, as this is a release/acquire pair and i need only be sequenced-after h).

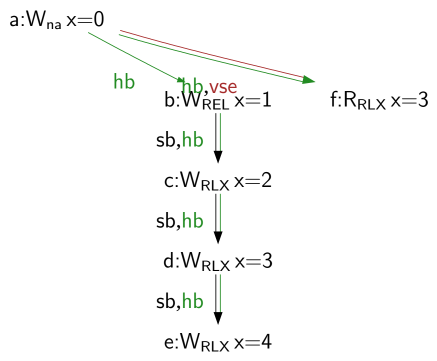


../examples/t47-hb.c

There are two additional restrictions on happens-before in consistent executions. Firstly, actions of *mo_seq_cst* memory order related by happens-before must be related by *sc* order in the same direction. Secondly, executions with cycles in their happens-before relation are not consistent.

Visible-side-effect Happens-before decides which writes can be read by a given read action in a consistent execution. Every write action that happens before the read with no intervening write is a *visible side effect* of the read. Non-atomic actions must read from one of their visible-side effects, and for an atomic read, its visible side effects decide the earliest writes in modification order that may be read. The following execution displays the visible side effects of the read action, with a vse edge.

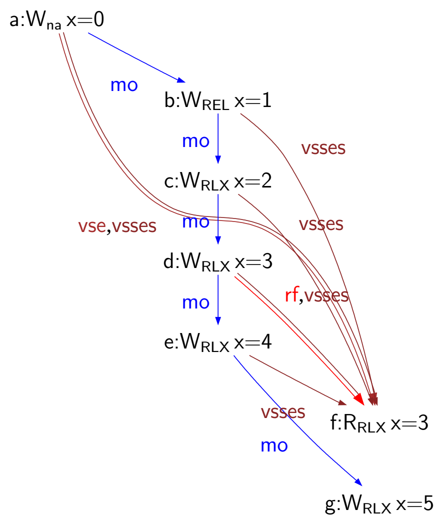
```
int main() {
  atomic_int x = 0;
  {{{ { x.store(1,mo_release);
        x.store(2,mo_relaxed);
        x.store(3,mo_relaxed);
        x.store(4,mo_relaxed); }
  ||| { printf("%d\n",x.load(mo_relaxed).readsvalue(3) ); } }}}
  return 0; }
```



../examples/t48-vse.c

Visible-sequence-of-side-effects Atomic reads take their values from a write in a *visible sequence of side effects* of the read. This is a contiguous part of the modification order that starts with a visible side effect and ends before the first write that the read happens-before. In our diagrams of executions, we draw these sequences as vsses edges from their elements to the read that they correspond to (the order among the sequence is simply modification order). The following execution shows a read from a write in a visible-sequence-of-side-effects.

```
int main() {
  atomic_int x = 0;
  {{{ { x.store(1,mo_release);
        x.store(2,mo_relaxed);
        x.store(3,mo_relaxed);
        x.store(4,mo_relaxed); }
  ||| { printf("%d\n",x.load(mo_relaxed).readsvalue(3) );
        x.store(5,mo_relaxed); } }}}
  return 0; }
```



../examples/t49-vsses.c

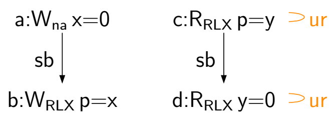
It can be shown that in a consistent execution there can only be one visible sequence of side effects for each read.

2.6 Races and indeterminate reads

If any consistent execution of a program exhibits an *indeterminate read* (ir), an *unsequenced race* (ur) or a *data race* (dr) then the the program has undefined behavior will result from attempting to run it. It is the responsibility of programmers to ensure their programs do not contain races or indeterminate reads.

Indeterminate read A read with no incoming reads-from edge is of indeterminate value. A program that has a consistent execution with an indeterminate read, like the one below, has undefined behaviour. The atomic library function `atomic_load_explicit` loads the value of the object specified in its first parameter with the memory order specified in its second.

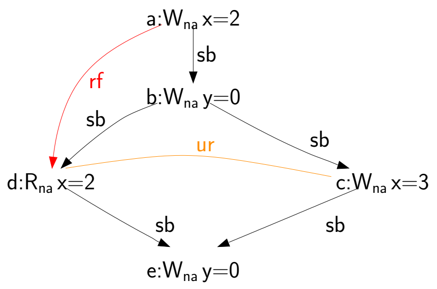
```
int main() {
  atomic_address p;
  {{{ { atomic_int x = 0;
        p.store(&x,mo_relaxed); }
  ||| { printf("%d\n",
          atomic_load_explicit (
            p.load(mo_relaxed),mo_relaxed)
          ); } }}}
  return 0; }
```



../examples/t65-ir-hand.exc

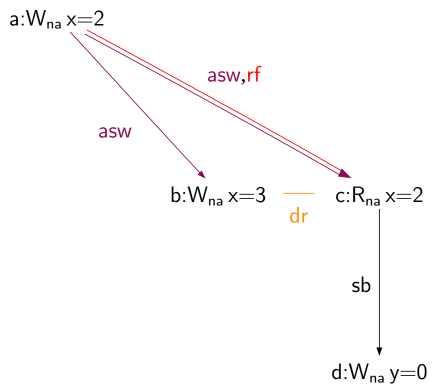
Unsequenced race We introduce the notion of an *unsequenced race* to capture the circumstances in §1.9p15 of N3092 that result in undefined behaviour. Two non-atomic actions on the same thread and location, one of which is a write, participate in an unsequenced race if neither is sequenced before the other. A program that has a consistent execution with an unsequenced race, like the one below, has undefined behaviour.

```
int main() {
  int x = 2;
  int y = 0;
  y = (x == (x=3));
  return 0; }
```



Data race Two actions on different threads but the same location, with at least one a write and one non-atomic, participate in a *data race* if neither happens-before the other. A program that has a consistent execution with a data race, like the one below, has undefined behaviour.

```
int main() {
  int x = 2;
  int y;
  {{{  x=3;
  |||  y=(x==3);
  }}};
  return 0; }
```

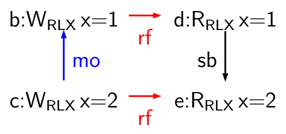


../examples/t4.c

2.7 Memory coherence

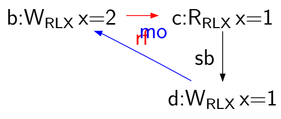
The key restriction that the memory model makes is on the values that are read by memory actions. As we stated in the previous section, non-atomic reads take the value of one of their visible side effects and atomic actions take the value of a write in one of their visible sequences of side effects. In addition there are four *coherence restrictions* that apply to the values that are read from a single location. Each restriction is explained below using a fragment of an execution that represents forbidden behaviour.

CoRR This restriction forbids two reads in a single thread from observing two writes in an order inconsistent with modification order. In the forbidden example execution fragment below, the reads in one thread observe writes in the opposite order to their modification order.



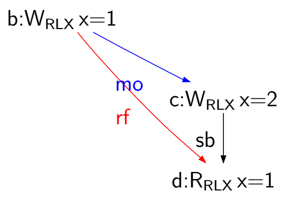
../examples/t61-corr-forbid.exc

CoRW This coherence restriction requires that a read that is sequenced before a write should not be able to read from a later write in modification order. The following forbidden execution fragment exhibits precisely this behaviour:



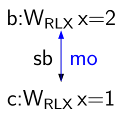
../examples/t62-corw-forbid.exc

CoWR This coherence axiom requires that a read that follows a write in sequenced-before should not be able to read from an earlier write in modification order. The following forbidden execution fragment exhibits precisely this behaviour:



../examples/t63-cowr-forbid.exc

CoWW This axiom requires modification order to agree with happens-before; if two write actions to the same location are related by happens-before then they are related by modification order in the same direction. The following forbidden execution illustrates this.



../examples/t64-coww-forbid.exc

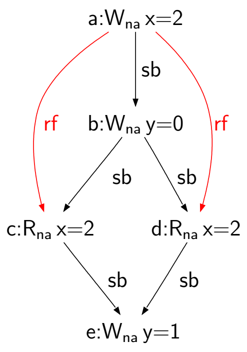
These coherence requirements are not all in the final committee draft (N3092), but there seemed to be consensus at the Rapperswil meeting that they should be added. We discuss this (and our other contributions to the model) in §5.

3. Exploring the model by example

In this section we will present some executions of several more examples in order to illustrate how the relations described above work together.

Single and multi-threaded executions We begin with the following single-threaded program:

```
int main() {
  int x = 2;
  int y = 0;
  y = (x == x);
  return 0; }
```



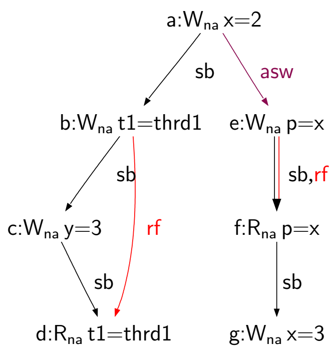
../examples/t1.c

This has only one execution, as shown. There are five actions, labelled a–e, all by the same thread. These are all non-atomic memory reads (R_{na}) or writes (W_{na}), with their address (x or y) and value (0,1, or 2). Actions a and b are the initialisation writes, c and d are the reads of the operands of the == operator, and e is a write of the result of ==.

The evaluations of the arguments to == are *unsequenced* in C++ (as are arguments to functions), meaning that they could be in either order, or even overlapping.

We now present an example with multiple threads. The program spawns a thread that writes 2 to x and concurrently writes 3 into y in the original thread.

```
void foo(int* p) {*p=3;}
int main() {
  int x = 2;
  int y;
  thread t1(foo, &x);
  y = 3;
  t1.join();
  return 0; }
```

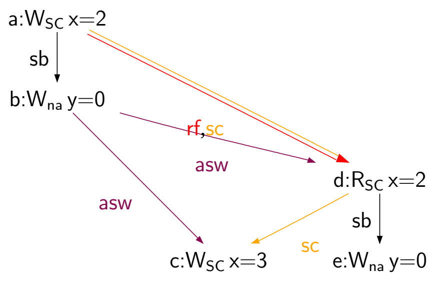


../examples/t3.c

The thread creation gives rise to additional-synchronized-with edges from sequenced-before-maximal actions of the parent thread before the thread creation to sequenced-before-minimal edges of the child.

Sequentially consistent atomics avoid a data-race We can alter the example with a data-race from the previous section to use an atomic object x and sequentially consistent atomic operations, providing the following, in which the concurrent access to x is not considered a data race, and so the program does not have undefined behaviour.

```
int main() {
  atomic_int x;
  x.store(2);
  int y=0;
  {{{ x.store(3);
  ||| y = ((x.load()) == 3);
  }}};
  return 0; }
```



../examples/t6.c

There is no race because sequentially consistent atomics are totally ordered, and so can be thought of as interleaving with each other in a global time-line. Their semantics are covered in detail in [BA08] and we will describe their precise integration into happens-before in §5.

Release/consume atomics On multiprocessors with weak memory orders, notably Power, release/acquire pairs are cheaper to implement than sequentially consistent atomics but still significantly more expensive than plain stores and loads. For example, the proposed Power implementation of load-acquire, `ld; cmp; bc; isync`, involves an `isync` [MS10]. However, Power (and also ARM) does guarantee that certain dependencies in an assembly program are respected, and in many cases those suffice, making the `isync` sequence unnecessary. As we understand it, this is the motivation for introducing a *read-consume* variant of read-acquire atomics. On a stronger processor (e.g. a TSO x86 or Sparc), or one where those dependencies are not respected, read-consume would be implemented just as read-acquire.

Read-consume enables efficient implementations of algorithms that use pointer reassignment for commits of their data, e.g. read-copy-update [MW]. For example, suppose one thread writes some data (perhaps spanning multiple words) then writes the address of that data to a shared atomic pointer, while the other thread reads the shared pointer, dereferences it and reads the data.

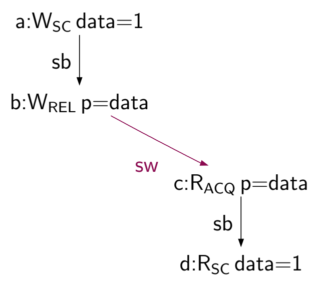
```
// sender                          | // receiver
data = ...                         | r1 = p
p = &data;                         | r2 = *r1;   // data
```

Here there is a dependency at the receiver from the read of p to the read of data. This can be expressed using a write-release and an atomic load of p annotated MO_CONSUME:

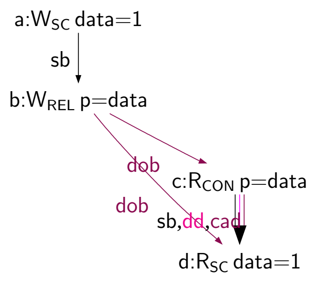
```
int main() {
  int data; atomic_address p;
  {{{ { data=1;
        p.store(&data, mo_release); }
  ||| printf("%d\n", *(p.load(mo_consume)) );
  }}};
  return 0; }
```

As we saw in the previous section, release/acquire pairs introduce synchronized-with edges, and happens-before includes the transitive closure of synchronized-with and sequenced-before — for a release/acquire version of this example, we would have the edges on the left below, and hence a happens before d.

First is an execution of the code above, and second is the execution of the same code using a read-consume in place of the read-acquire. Happens-before is the same for both fragments, but it is derived differently for consumes.



../examples/t28-datadep-rel-acq.c

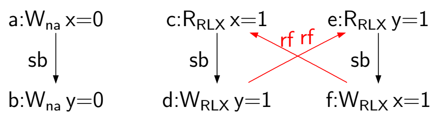


../examples/t20-simple-rel-con.c

For release/consume, the key fact is that there is a *data dependency* (dd) from c to d, as shown above. The (dd) edge gives rise to a *carries-a-dependency-to* (cad) edge, which extends data dependency with thread-local reads-from relationships. In turn, this gives rise to a *dependency-ordered-before* (dob) edge, which is the release/consume analogue of the release/acquire synchronizes-with edge. This involves release sequences as before (in the example just the singleton [b]).

Thin-air reads The model defined here allows *thin-air reads*: the program below reads the value 1, yet there is *no* occurrence of 1 in the program source.

```
int main() {
  int r1, r2;
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { r1 = x.load(mo_relaxed));
        y.store(r1,mo_relaxed); }
  ||| { r2 = y.load(mo_relaxed));
        x.store(r2,mo_relaxed); }
  }}}
  return 0; }
```



../examples/t30-lb-relaxed.c

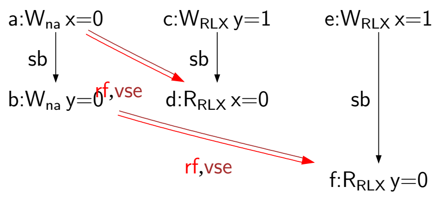
An execution like this would be surprising, and in fact would not happen with typical hardware and compilers. In the Java Memory Model [MPA05], much of the complexity of the model arises from the desire to outlaw thin-air reads, which there is essential to prevent forging of pointers. The C++0x final committee draft attempts to forbid such executions as well, but the restrictions it imposes to that end seem to have unfortunate consequences, as we discuss in Section 5; they are not incorporated into the formal model presented here.

4. Classic litmus tests

We now illustrate the varying strength of the different memory orders that actions can take by showing the semantics of some 'classic' examples. In all cases, variants of the examples with sequentially consistent atomics do not have the weak-memory behaviour. As in our other diagrams, to avoid clutter we only show selected edges.

Store Buffering (SB) Here two threads write to separate locations and then each reads from the other location. In Total Store Order (TSO) models both can read from before (w.r.t. coherence) the other write in the same execution. In C++0x this behaviour is allowed if those four actions are relaxed, for release/consume pairs and for release/acquire pairs. Assuming that the initialization writes to the locations are relaxed or non-atomic, it is still allowed even if those four are sequentially consistent. If every action (including the two initial writes) is sequentially consistent, this is forbidden by the consistent_sc_order condition. We show the relaxed example below:

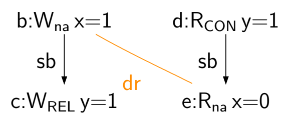
```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;

  {{{ { y.store(1,mo_relaxed);
        printf("1:%d\n",x.load(mo_relaxed).readsvalue(0)); }
  ||| { x.store(1,mo_relaxed);
        printf("2:%d\n", y.load(mo_relaxed).readsvalue(0)); }
  }}}
  return 0;
}
```



../examples/t37a-sb-relaxed.c

Message Passing (MP) Here one thread (non-atomically) writes data and then an atomic flag while a second thread waits for the flag and then (non-atomically) reads data; the question is whether it is guaranteed to see the data written by the first. With a release/acquire pair it is. A release/consume pair gives the same guarantee precisely when there is a dependency between the reads, otherwise there is a consistent execution in which there is a data race (here the second thread sees the initial value of x; the candidate execution in which the second thread sees the write x=1 is ruled out as that does not happen-before the read and so is not a visible side effect).

```
int main() {
  int x = 0; atomic_int y;
  {{{ { x=1;
        y.store(1,mo_release); }
  ||| { printf("%d\n", y.load(mo_consume).readsvalue(1));
        printf("%d\n", x.readsvalue(0)); }  }}}
  return 0;
}
```

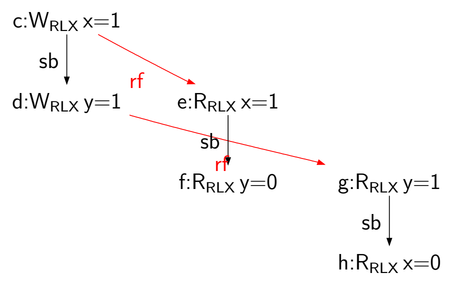


../examples/t29-mp-consume.c

The same holds with relaxed flag operations.

In a variant in which all writes and reads are release/consumes or relaxed atomics, eliminating the race, and there are two copies of the reading thread, the two reading threads can see the two writes of the writing thread in opposite orders (as below) — consistent with what one might see on Power, for example.

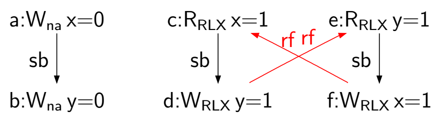
```
int main() {
  atomic_int x = 0; atomic_int y = 0;
  {{{  { x.store(1, mo_relaxed);
         y.store(1, mo_relaxed); }
  ||| { printf("%d\n", x.load(mo_relaxed).readsvalue(1));
        printf("%d\n", y.load(mo_relaxed).readsvalue(0)); }
  ||| { printf("%d\n", y.load(mo_relaxed).readsvalue(1));
        printf("%d\n", x.load(mo_relaxed).readsvalue(0)); }
  }}};
  return 0; }
```



../examples/irdw-relaxed.c

Load Buffering (LB) In this variant of the SB example the question is whether the two reads can both see the (sequenced-before) later write of the other thread in the same execution. With relaxed atomics this is allowed, as below:

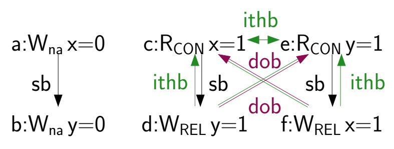
```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;

  {{{ { printf("1:%d\n",x.load(mo_relaxed).readsvalue(1));
        y.store(1,mo_relaxed); }
  ||| { printf("2:%d\n", y.load(mo_relaxed).readsvalue(1));
        x.store(1,mo_relaxed); }
  }}}
  return 0;
}
```



../examples/t30-lb-relaxed.c

but with release/consumes (with dependencies) it is not (as shown in the forbidden execution below), because inter-thread-happens-before would be cyclic. It is not allowed for release/acquire and sequentially consistent atomics (which are stronger than release/consumes with dependencies), both because of the cyclic inter-thread-happens-before and for other reasons.

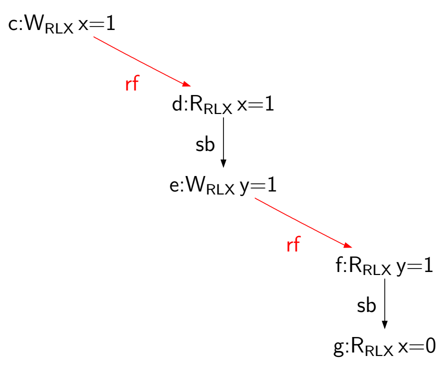
```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { printf("%d\n",x.load(mo_consume).readsvalue(1));
        y.store(1,mo_release); }
  ||| { printf("%d\n", y.load(mo_consume).readsvalue(1));
        x.store(1,mo_release); } }}} ;
  return 0; }
```



../examples/t31-lb-consume.c

Write-to-Read Causality (WRC) Here the first (non-initialisation) thread writes to x; the second reads from that and then (w.r.t. sequenced-before) writes to y; the third reads from that and then (w.r.t. sequenced-before) reads x. The question is whether it is guaranteed to see the first thread's write.

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;

  {{{ x.store(1,mo_relaxed);
  ||| { printf("1:%d\n",x.load(mo_relaxed).readsvalue(1));
        y.store(1,mo_relaxed); }
  ||| { printf("2:%d\n",y.load(mo_relaxed).readsvalue(1));
        printf("3:%d\n",x.load(mo_relaxed).readsvalue(0)); }
  }}}
  return 0;
}
```

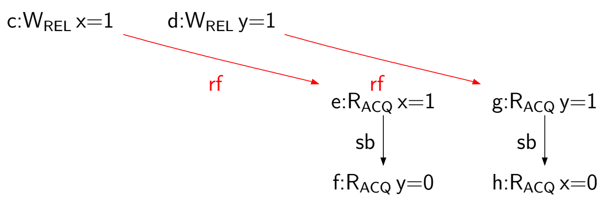


../examples/t32-wrc-relaxed.c

With relaxed atomics, this is not guaranteed, as shown above, while with release/acquires it is, as the *synchronizes-with* edges in the inter-thread-happens-before relation interfere with the required read-from map.

Independent Reads of Independent Writes (IRIW) Here the first two (non-initialisation) threads write to different locations; the question is whether the last two threads can see those writes in different orders. With relaxed, release/acquire, or release/consume atomics, they can.

```
int main() {
  atomic_int x = 0; atomic_int y = 0;
  {{{ x.store(1, mo_release);
  ||| y.store(1, mo_release);
  ||| { printf("1%d",  x.load(mo_acquire).readsvalue(1));
        printf("%d\n", y.load(mo_acquire).readsvalue(0)); }
  ||| { printf("2%d",  y.load(mo_acquire).readsvalue(1));
        printf("%d\n", x.load(mo_acquire).readsvalue(0)); }
  }}};
  return 0; }
```

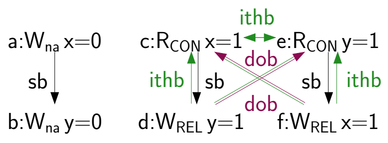


../examples/t9.c

5. From standard to formalisation and back

We developed the formal model presented in Section 6 by a lengthy iterative process: building formalisations of various drafts of the standard, and of Boehm and Adve's model without low-level atomics [BA08]; considering the behaviour of examples, both by hand and with our tool; trying to prove properties of the formalisations; and discussing issues with members of the Concurrency subcommittee of the C++ Standards Committee (TC1/SC22/WG21). To give a flavour of this process, and to explain how our formalisation differs from the final committee draft (N3092) of the standard, we describe several issues with that draft. These were discussed at the Rapperswil meeting (we provide the issue numbers from the meeting), and before; in most cases there seems to be consensus on a proposed fix, and our formal model incorporates it. These issues also serve to bring out the delicacy of the standard, and the pitfalls of prose specification, even when carried out with great care.

Acyclicity of happens-before [issues CA 8 and GB 10] N3092 defines happens-before, making plain that it is not necessarily transitive, but does not state whether it is required to be acyclic (or whether, perhaps, a program with a cyclic execution is deemed to have undefined behaviour). The release/consume LB example of the previous section has a cyclic inter-thread-happens-before, as shown there, but is otherwise a consistent execution.



../examples/t31-lb-consume.c

After discussion, it seems clear that executions with cyclic inter-thread-happens-before (or, equivalently, cyclic happens-before) should not be considered, so we impose that explicitly.

Additional happens-before edges [issue CA 9] There are 6 places where N3092 adds happens-before relationships explicitly (in addition to those from sequenced-before and inter-thread-happens-before), e.g. between the invocation of a thread constructor and the function that the thread runs. As happens-before is carefully *not* transitively closed, such edges would not be transitive with (e.g.) sequenced-before. Accordingly, we suggested that they be added to the synchronized-with relation; for those within the C++ fragment supported by our tool, our operational semantics introduces them into additional-synchronized-with.

'Subsequent' in visible sequences of side effects [issue CA 11, not a defect] In the final committee draft, with cyclic inter-thread-happens-before permitted, the definition of visible sequence of side effects in 1.10p12 had a pathological case which could be removed by deleting the word "subsequent" there. With the acyclicity condition, this makes no semantic difference, though it still improves the clarity of the definition.

'Maximal' in release sequences [issues CA 12 and GB 9] N3092 defines a release sequence as follows: *A release sequence on an atomic object M is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first operation is a release, and every subsequent operation: (1) is performed by the same thread that performed the release, or (2) is an atomic read-modify-write operation.*

We initially read that as maximal w.r.t. sequence inclusion, which would preclude synchronizing with releases which have other releases sequenced-before them. The intended concept, however, seems to be that of the maximal release sequence *from a particular release operation*; we formalise that (and explicit use of 'maximal' turns out to be unnecessary).

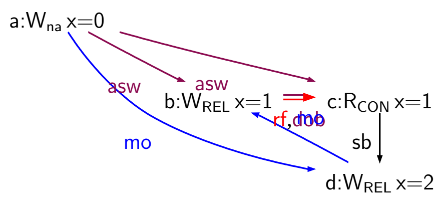
Non-unique visible sequences of side effects and happens-before ordering [issue CA 18] We suggested altering the wording of the definition of visible sequence of side effects (§1.10p13 of N3092) so that it refers to "a" sequence rather than "the" sequence. Without our additional coherence axioms multiple sequences were allowed for a single read action, but we have proved that with the axioms the sequence is unique. In CA 18 we suggested adding the following note to 1.10p13 to allude to this:

> "[Note - It can be shown that the visible sequence of side effects of a value computation is unique given the coherence requirements below. - end note]"

Coherence requirements [issues GB 11, GB 12, CA 18, GB 11, CA 19, GB 12, and CA 20] N3092 enforces some coherence of reads and writes to a single location, but during our iterative process of building our formalized model, we noticed that not all aspects of coherence were required (in particular, CoRW and CoWR were not enforced), which permitted pathological executions for several examples. We believe that current typical hardware (including x86, Power and ARM) satisfies all four axioms without any barriers, and suggested adding the remaining coherence axioms.

If the additional coherence axioms are not added then some unintuitive behaviours are allowed. For example, the following execution is allowed without the CoRW coherence requirement:

```
int main() {
  atomic_int x = 0;
  {{{ x.store(1, mo_release);
  ||| { printf("%d\n", x.load(mo_consume).readsvalue(1));
        x.store(2, mo_release); } }}};
  return 0; }
```

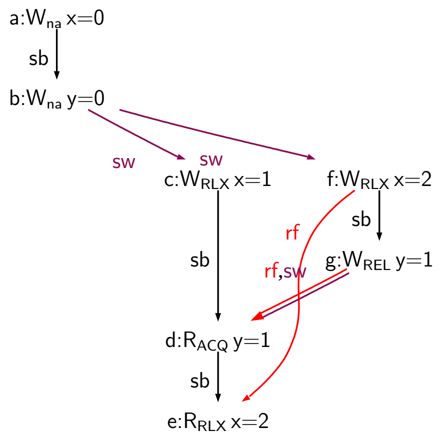


../examples/t26-anti-mo-consume-full.c

Here there is a dependency-ordered-before edge for a write-release/read-consume pair (b),(c), with (c) reading from (b), but the write (b) *follows* (in modification order) another write (d) that *follows* the read in sequenced-before. The read (c) reads from a write that is ordered after it by sequenced before and dependency-ordered-before.

If the CoWR restriction is omitted the following example is permitted:

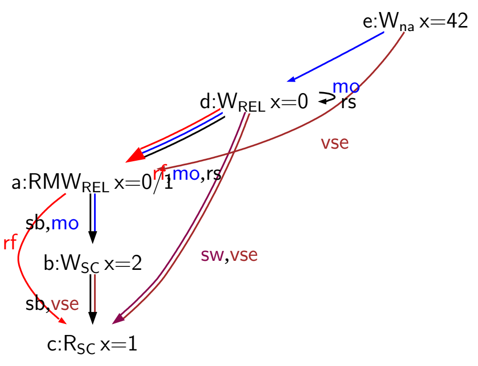
```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(1,mo_relaxed);
        printf("%d\n", y.load(mo_acquire).readsvalue(1));
        printf("%d\n", x.load(mo_relaxed).readsvalue(2)); }
  ||| { x.store(2,mo_relaxed);
        y.store(1,mo_release); } }}};
  return 0; }
```



../examples/cowr-mod.c

Here (g) synchronizes with (d), making (f) happen before (e). Now (f) is a visible side effect of (e), and can be read from, even though (f) is earlier in modification order than (c). The read (e) reads from a write that is not the most recent in the union sequenced before and modification-order.

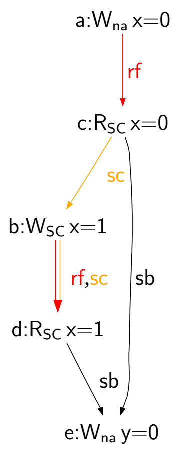
Another even more unintuitive execution is permitted if the CoWR restriction is omitted. Here the read-acquire (c) synchronizes-with the write-release (d) by reading from the RMW (a), which is in the release sequence headed by (d) (as read-modify-writes from any thread are allowed by the definition of release-sequence). Counter-intuitively, this can happen despite the presence of an intervening write (b) in sequenced-before. This execution is particularly confusing because one might expect the relatively strongly ordered sequentially consistent atomics to forbid it.



../examples/t36-hidden-rmw.exc

Overlapping executions and thin-air reads The N3092 draft standard attempts to forbid thin air reads, with: *An atomic store shall only store a value that has been computed from constants and program input values by a finite sequence of program evaluations, such that each evaluation observes the values of variables as computed by the last prior assignment in the sequence.* This seems to be overly constraining. For example, two subexpression evaluations (in separate threads) can overlap (e.g. if they are the arguments of a function call) and can contain multiple actions. With relaxed atomics there can be consistent executions in which it is impossible to disentangle the two into any sequence, for example as below, where the SC-write of x must be between the two reads of x. In our formalisation we currently do not impose any thin-air condition.

```
int main() {
  atomic_int x = 0;
  int y;
  {{{ x.store(1);
  ||| { y = (x.load()==x.load()); }
  }}};
  return 0; }
```



../examples/t27-inter-evaluation.c

The Final Committee Draft defines the model in terms of actions and evaluations of expressions. Our model does not keep track of which evaluations give rise to which actions, and instead considers only actions.

6. The model as formalised

In this section we present our formalized version of the C++0x memory model simultaneously in prose and typeset mathematics. It should be possible to understand the model from the prose alone, skipping over all the boxed mathematics. However, the mathematical version is the actual definition (and is the one used in our CPPMEM tool).

6.1 Overall structure of the model

A candidate execution X of a C++ program consists of two parts, X_{opsem}, which is data given by the path of control flow and syntactic structure of the program, and X_{witness}, which is constrained only by the memory model. The X_{opsem} part of a candidate execution X consists of

- threads, a set of thread ids
- $actions$, a set of actions (§7)
- *location-kind*, a location typing (§5)

and four binary relations over its actions:

- *sequenced-before* (sb),
- *additional-synchronized-with* (asw),
- *data-dependency* (dd), and
- *control-dependency* (cd) (unused at present).

The X_{witness} part of a candidate execution X consists of a further three binary relations over its actions:

- *rf*,
- *sc*, and
- *modification-order* (mo).

Given a candidate execution $X = (X_{\text{opsem}}, X_{\text{witness}})$, we define various derived relations (§6.11–6.15):

- *release-sequence* (rs)
- *hypothetical-release-sequence* (hrs)
- *synchronizes-with* (sw)
- *carries-a-dependency-to* (cad)
- *dependency-ordered-before* (dob)
- *inter-thread-happens-before* (ithb)
- *happens-before* (hb)
- *visible-side-effect* (vse)
- *visible-sequences-of-side-effects* (vsses)

together with the predicates required to define

- $consistent\_execution$ (§6.21)

and our three sources of undefined behaviour (§6.22):

- *unsequenced-race* (ur),
- data races dr, and
- indeterminate reads ir.

The top-level definition is cpp_memory_model (§6.23), which, given an operational semantics and a program, is either null if the program has undefined behaviour or the set of all consistent executions if it does not.

In the Isabelle/HOL source each definition is explicitly parameterised on the components of a candidate execution and the required derived relations, but here we suppress that parameterisation to reduce clutter. The Isabelle/HOL also contains set-typed versions of some of the predicates, for use in code extraction; we suppress those here also.

6.2 Notation

We let a, b, c, x, y, z, rs_head, and $vsse\_head$ range over actions, l range over locations, v range over values, aid range over action ids, tid range over thread ids, and mem_ord range over memory orders.

The relations of a candidate execution, e.g. *sequenced-before*, are mostly binary relations over actions. We write $a \xrightarrow{\textit{sequenced-before}} b$ to mean that action a is related to b by *sequenced-before*.

The notation used in the mathematical definitions is largely standard; it is summarised in Fig. 1 for reference.

6.3 Auxiliary definitions

A relation is over a set if both the domain and range of the relation are subsets of the set.

$$\boxed{\text{relation\_over } s\ rel = \text{domain } rel \subseteq s \land \text{range } rel \subseteq s}$$

A relation restricted to a set is the intersection of the relation with the cartesian product of the set with itself.

$$\boxed{\xrightarrow{rel}|_s = rel \cap (s \times s)}$$

A strict preorder is an irreflexive and transitive order.

$$\boxed{\text{strict\_preorder } ord = \text{irreflexive } ord \land \text{trans } ord}$$

A relation that is total over a set is

1. a relation over the set, and
2. any two elements of the set either have a directed edge between them in one direction or the elements must be equal.

$$\boxed{\begin{array}{l}\text{total\_over } s\ ord = \\ \quad \text{relation\_over } s\ ord \land \\ \quad (\forall x \in s.\ \forall y \in s.\ x \xrightarrow{ord} y \lor y \xrightarrow{ord} x \lor (x = y))\end{array}}$$

A strict total order over a set is a strict preorder that is a total order over the set.

$$\boxed{\begin{array}{l}\text{strict\_total\_order\_over } s\ ord = \\ \quad \text{strict\_preorder } ord \land \text{total\_over } s\ ord\end{array}}$$

An element x is adjacent-less-than y according to an order, such that some predicate holds if

1. the predicate holds on x,
2. $x \xrightarrow{ord} y$ and
3. there is no element z such that $x \xrightarrow{ord} z$, $z \xrightarrow{ord} y$, and for which the predicate holds.

$$\boxed{\begin{array}{l}x \overset{ord}{\longmapsto}_{pred}\ y = \\ \quad pred\ x \land x \xrightarrow{ord} y \land \neg(\exists z.\ pred\ z \land x \xrightarrow{ord} z \xrightarrow{ord} y)\end{array}}$$

An element x is adjacent-less-than y according to an order if

1. $x \xrightarrow{ord} y$ and
2. there is no element z such that $x \xrightarrow{ord} z$ and $z \xrightarrow{ord} y$.

$$\boxed{\begin{array}{l}x \overset{ord}{\longmapsto} y = \\ \quad x \xrightarrow{ord} y \land \neg(\exists z.\ x \xrightarrow{ord} z \xrightarrow{ord} y)\end{array}}$$

Types We use base types bool and string. We define type abbreviations action_id, thread_id, location, and val, and define enumeration types memory_order_enum, action, and location_kind.

Constructors The definitions of enumeration types introduce constructors, e.g. MO_ACQUIRE of type memory_order_enum. Constructors can take arguments, which are written without parentheses. For example,

LOAD `"a"` `"thread0"` `"loc10"` `"v0"`

is an action consisting of the LOAD constructor applied to action id `"a"`, thread id `"thread0"`, location `"loc10"`, and value `"v0"`.

Options We use option types in several places. For example, the value_read function (§7) takes an action a and returns either NONE, if a does not read any value, or SOME v, if a reads value v.

Formulas (or predicates)

T	true
F	false
$\neg P$	not P
$P \vee Q$	P or Q
$P \wedge Q$	P and Q
$P \implies Q$	P implies Q
$P = Q$	P equals Q
$\forall x.\ P$	for all x, P holds
$\exists x.\ P$	there exists x such that P
$\forall x \in A.\ P$	for all x in A, P holds
$\exists x \in A.\ P$	there exists x in A such that P
$x \in A$	x is an element of set A

Sets

$\{x.\ P\}$	the set of all x that satisfy P
$A \cup B$	the union of sets A and B
$A \cap B$	the intersection of sets A and B

Relations The relations of a candidate execution, e.g. *sequenced-before*, are mostly binary relations over actions (or, equivalently, sets of pairs of actions). We write $a \xrightarrow{\textit{sequenced-before}} b$ (or equivalently $(a, b) \in$ *sequenced-before*) to mean that action a is related to b by *sequenced-before*. We write $r \circ s$ for the composition of relations r and s, so $a \xrightarrow{r \circ s} c$ if and only if there exists some b such that $a \xrightarrow{r} b$ and $b \xrightarrow{s} c$. We write r^+ for the transitive closure of relation r.

Local definitions and case analysis Local definition of x to be $t1$ in $t2$:

let $x = t1$ **in** $t2$

Conditional:

if P **then** $t1$ **else** $t2$

Case analysis of t, matching against patterns $pat1 \ldots patn$:

case t **of** $pat1 \rightarrow t1 \parallel \ldots \parallel patn \rightarrow tn$

Patterns can include variables and wildcards (_) and compound patterns built by applying constructors, e.g. LOCK aid _ _ is a pattern that matches LOCK actions and picks out the action id thereof.

Top-level definitions Auxiliary functions, relations and predicates are usually defined just by writing

$f\ arg1 \ldots argn = RHS$

where $arg1 \ldots argn$ are the formal parameters of f. Functions can also be defined by pattern matching, by giving a conjuction of clauses, e.g. as for the definition of action_id_of (§7).

Figure 1. Mathematical Notation

6.4 Memory actions: types

The following four type abbreviations are all synonyms for string, for use in our extracted checker.

```
type_abbrev action_id : string
```

```
type_abbrev thread_id : string
```

```
type_abbrev location : string
```

```
type_abbrev val : string
```

A memory order is one of the six options below (29.3p1).

```
memory_order_enum =
     MO_SEQ_CST
   | MO_RELAXED
   | MO_RELEASE
   | MO_ACQUIRE
   | MO_CONSUME
   | MO_ACQ_REL
```

An action is either

1. a lock (with an identifier) by a thread at a location (30.4),
2. an unlock (with an identifier) by a thread at a location (30.4),
3. an atomic-load (with an identifier) by a thread, with a particular order, at a location, of a value (29.6),
4. an atomic-store (with an identifier) by a thread, with a particular order, at a location, of a value (29.6),
5. an atomic-read-modify-write (with an identifier) by a thread, with a particular order, at a location, that reads a value and writes a value (29.6),
6. a load (with an identifier) by a thread, at a location, of a value,
7. a store (with an identifier) by a thread, at a location, of a value or
8. a fence (with an identifier) by a thread, with a particular order (29.8).

```
action =
     LOCK of action_id thread_id location
   | UNLOCK of action_id thread_id location
   | ATOMIC_LOAD of action_id thread_id memory_order_enum location val
   | ATOMIC_STORE of action_id thread_id memory_order_enum location val
   | ATOMIC_RMW of action_id thread_id memory_order_enum location val val
   | LOAD of action_id thread_id location val
   | STORE of action_id thread_id location val
   | FENCE of action_id thread_id memory_order_enum
```

6.5 Memory actions: routine accessor functions

The action_id_of, thread_id_of, memory_order, location, value_read and value_written functions simply take an action and return the relevant component of it.

```
(action_id_of (LOCK aid _ _) = aid) ∧
(action_id_of (UNLOCK aid _ _) = aid) ∧
(action_id_of (ATOMIC_LOAD aid _ _ _ _) = aid) ∧
(action_id_of (ATOMIC_STORE aid _ _ _ _) = aid) ∧
(action_id_of (ATOMIC_RMW aid _ _ _ _ _) = aid) ∧
(action_id_of (LOAD aid _ _ _) = aid) ∧
(action_id_of (STORE aid _ _ _) = aid) ∧
(action_id_of (FENCE aid _ _) = aid)
```

```
(thread_id_of (LOCK _ tid _) = tid) ∧
(thread_id_of (UNLOCK _ tid _) = tid) ∧
(thread_id_of (ATOMIC_LOAD _ tid _ _ _) = tid) ∧
(thread_id_of (ATOMIC_STORE _ tid _ _ _) = tid) ∧
(thread_id_of (ATOMIC_RMW _ tid _ _ _ _) = tid) ∧
(thread_id_of (LOAD _ tid _ _) = tid) ∧
(thread_id_of (STORE _ tid _ _) = tid) ∧
(thread_id_of (FENCE _ tid _) = tid)
```

```
(memory_order (ATOMIC_LOAD _ _ mem_ord _ _) =
  SOME mem_ord) ∧
(memory_order (ATOMIC_STORE _ _ mem_ord _ _) =
  SOME mem_ord) ∧
(memory_order (ATOMIC_RMW _ _ mem_ord _ _ _) =
  SOME mem_ord) ∧
(memory_order (FENCE _ _ mem_ord) =
  SOME mem_ord) ∧
(memory_order _ =
  NONE)
```

```
(location (LOCK _ _ l) = SOME l) ∧
(location (UNLOCK _ _ l) = SOME l) ∧
(location (ATOMIC_LOAD _ _ _ l _) = SOME l) ∧
(location (ATOMIC_STORE _ _ _ l _) = SOME l) ∧
(location (ATOMIC_RMW _ _ _ l _ _) = SOME l) ∧
(location (LOAD _ _ l _) = SOME l) ∧
(location (STORE _ _ l _) = SOME l) ∧
(location (FENCE _ _ _) = NONE)
```

```
(value_read (ATOMIC_LOAD _ _ _ _ v) = SOME v) ∧
(value_read (ATOMIC_RMW _ _ _ _ v _) = SOME v) ∧
(value_read (LOAD _ _ _ v) = SOME v) ∧
(value_read _ = NONE)
```

```
(value_written (ATOMIC_STORE _ _ _ _ v) = SOME v) ∧
(value_written (ATOMIC_RMW _ _ _ _ _ v) = SOME v) ∧
(value_written (STORE _ _ _ v) = SOME v) ∧
(value_written _ = NONE)
```

The is_lock, is_unlock, is_atomic_load, is_atomic_store, is_atomic_rmw, is_load, is_store and is_fence predicates simply take an action and return true or false depending on whether it is an action of the indicated kind.

```
is_lock a =
  case a of LOCK _ _ _ → T || _ → F
```

```
is_unlock a =
  case a of UNLOCK _ _ _ → T || _ → F
```

```
is_atomic_load a =
  case a of ATOMIC_LOAD _ _ _ _ _ → T || _ → F
```

```
is_atomic_store a =
  case a of ATOMIC_STORE _ _ _ _ _ → T || _ → F
```

```
is_atomic_rmw a =
  case a of ATOMIC_RMW _ _ _ _ _ _ → T || _ → F
```

```
is_load a = case a of LOAD _ _ _ _ → T || _ → F
```

```
is_store a = case a of STORE _ _ _ _ → T || _ → F
```

```
is_fence a = case a of FENCE _ _ _ → T || _ → F
```

6.6 Memory actions: useful collections of actions

A *lock-or-unlock action* is a lock action or an unlock action.

```
is_lock_or_unlock a = is_lock a ∨ is_unlock a
```

An *atomic action* is an atomic load, atomic store or atomic read-modify-write action.

```
is_atomic_action a =
  is_atomic_load a ∨ is_atomic_store a ∨ is_atomic_rmw a
```

A *load-or-store action* is a load action or a store action.

```
is_load_or_store a = is_load a ∨ is_store a
```

A *read action* is an atomic load, an atomic read-modify-write, or a non-atomic load.

```
is_read a =
  is_atomic_load a ∨ is_atomic_rmw a ∨ is_load a
```

A *write action* is an atomic store, an atomic read-modify-write, or a non-atomic store.

```
is_write a =
  is_atomic_store a ∨ is_atomic_rmw a ∨ is_store a
```

An *acquire action* is a read or a fence with memory order MO_ACQUIRE, MO_ACQ_REL, or MO_SEQ_CST, a fence with memory order MO_CONSUME, or a lock (1.10p4).

```
is_acquire a =
  (case memory_order a of
    SOME mem_ord →
      (mem_ord ∈
        {MO_ACQUIRE, MO_ACQ_REL, MO_SEQ_CST} ∧
      (is_read a ∨ is_fence a)) ∨
      (* 29.8:5 states that consume fences are acquire fences. *)
      ((mem_ord = MO_CONSUME) ∧ is_fence a)
  || NONE → is_lock a)
```

A *consume action* is a read with memory order MO_CONSUME (1.10p4).

```
is_consume a =
  is_read a ∧ (memory_order a = SOME MO_CONSUME)
```

A *release action* is a write or a fence with memory order MO_RELEASE, MO_ACQ_REL, or MO_SEQ_CST, or an unlock (1.10p4).

```
is_release a =
  (case memory_order a of
    SOME mem_ord →
      mem_ord ∈ {MO_RELEASE, MO_ACQ_REL, MO_SEQ_CST} ∧
    (is_write a ∨ is_fence a)
  || NONE → is_unlock a)
```

A *seq-cst action* is an action with memory order MO_SEQ_CST.

```
is_seq_cst a = (memory_order a = SOME MO_SEQ_CST)
```

6.7 Location kinds

Locations are subject to a very weak type system: each location stores a particular kind of object. The atomic actions can only be performed on ATOMIC locations. The non-atomic reads and writes can be performed on either ATOMIC or NON_ATOMIC locations. Locks and unlocks are *mutex* actions and can only be performed on MUTEX locations.

```
location_kind =
    MUTEX
  | NON_ATOMIC
  | ATOMIC
```

The *location kind* map associates a location kind, tagged with SOME , to each location mentioned in the candidate execution, and associates NONE to all other locations. The actions of an execution respect the location kinds if there are only lock and unlock actions on mutex locations, loads and stores on non-atomic locations and loads, stores, atomic-loads and atomic-stores on atomic locations.

```
actions_respect_location_kinds =
  ∀a.
    case location a of SOME l →
      (case location-kind l of
        MUTEX → is_lock_or_unlock a
       || NON_ATOMIC → is_load_or_store a
       || ATOMIC → is_load_or_store a ∨ is_atomic_action a)
    || NONE → T
```

There is a check that an action is at a location of a given kind.

```
is_at_location_kind =
  case location a of
    SOME l → (location-kind l = lk0)
  || NONE → F
```

The is_at_mutex_location, is_at_non_atomic_location and is_at_atomic_location predicates check that a particular action is at a location of their respective kinds.

```
is_at_mutex_location a =
  is_at_location_kind a MUTEX
```

```
is_at_non_atomic_location a =
  is_at_location_kind a NON_ATOMIC
```

```
is_at_atomic_location a =
  is_at_location_kind a ATOMIC
```

6.8 Well-formed threads

Actions on the same thread are actions that have the same thread id.

```
same_thread a b = (thread_id_of a = thread_id_of b)
```

A threadwise a relation over a set is a relation over that set that only relates actions that are in the same thread as one another.

```
threadwise_relation_over s rel =
  relation_over s rel ∧ (∀(a, b) ∈ rel. same_thread a b)
```

Actions on the same location are actions that have the same location.

```
same_location a b = (location a = location b)
```

The locations of a set of actions is the set containing the location of every member of the action set

```
$\text{locations\_of } \mathit{actions} = \{l.\ \exists a.\ (\text{location } a = \textsc{Some } l)\}$
```

A well formed action is an atomic load, an atomic store, or an atomic read-modify-write with an apporpriate memory order (as specified below), or any other action.

```
$\text{well\_formed\_action } a =$
  $\textbf{case } a \textbf{ of}$
    $\textsc{Atomic\_load}\ \_\ \_\ \mathit{mem\_ord}\ \_\ \_ \rightarrow \mathit{mem\_ord} \in$
      $\{\textsc{Mo\_relaxed}, \textsc{Mo\_acquire}, \textsc{Mo\_seq\_cst}, \textsc{Mo\_consume}\}$
  $\|\ \textsc{Atomic\_store}\ \_\ \_\ \mathit{mem\_ord}\ \_\ \_ \rightarrow \mathit{mem\_ord} \in$
      $\{\textsc{Mo\_relaxed}, \textsc{Mo\_release}, \textsc{Mo\_seq\_cst}\}$
  $\|\ \textsc{Atomic\_rmw}\ \_\ \_\ \mathit{mem\_ord}\ \_\ \_\ \_ \rightarrow \mathit{mem\_ord} \in$
      $\{\textsc{Mo\_relaxed}, \textsc{Mo\_release}, \textsc{Mo\_acquire},$
        $\textsc{Mo\_acq\_rel}, \textsc{Mo\_seq\_cst}, \textsc{Mo\_consume}\}$
  $\|\ \_ \rightarrow \mathbf{T}$
```

A set of well formed threads has

1. an injective action id map,
2. well formed actions,
3. *sc*, *data-dependency* and *control-dependency* relations that are strict preorders and threadwise over the actions,
4. an *additional-synchronized-with* relation that is over the actions,
5. actions that have a thread id of any thread of the execution and that respect the location kinds and
6. a *data-dependency* relation that is a subset of *sequenced-before*.

```
$\text{well\_formed\_threads} =$
  $\text{inj\_on action\_id\_of } (\mathit{actions}) \wedge$
  $(\forall a.\ \text{well\_formed\_action } a) \wedge$
  $\text{threadwise\_relation\_over } \mathit{actions}\ \textit{sequenced-before} \wedge$
  $\text{threadwise\_relation\_over } \mathit{actions}\ \textit{data-dependency} \wedge$
  $\text{threadwise\_relation\_over } \mathit{actions}\ \textit{control-dependency} \wedge$
  $\text{strict\_preorder } \textit{sequenced-before} \wedge$
  $\text{strict\_preorder } \textit{data-dependency} \wedge$
  $\text{strict\_preorder } \textit{control-dependency} \wedge$
  $\text{relation\_over } \mathit{actions}\ \textit{additional-synchronized-with} \wedge$
  $(\forall a.\ \text{thread\_id\_of } a \in \text{threads}) \wedge$
  $\text{actions\_respect\_location\_kinds} \wedge$
  $\textit{data-dependency} \subseteq \textit{sequenced-before}$
```

6.9 Well-formed reads-from mapping

A well formed reads from mapping is a relation over the actions such that

1. if $a \xrightarrow{rf} b$ and $a' \xrightarrow{rf} b$ then a and a' are the same action and
2. if $a \xrightarrow{rf} b$ then
   (a) a and b must be at the same location,
   (b) b must read the value that a writes,
   (c) a cannot equal b,
   (d) if on a mutex location then a must be an unlock and b a lock,
   (e) if on a non-atomic location then a must be a store and b a load and
   (f) if on an atomic location then a must be an atomic store or read-modify-write and b must be an atomic load or read-modify-write.

```
$\text{well\_formed\_reads\_from\_mapping} =$
  $\text{relation\_over } \mathit{actions}\ (\xrightarrow{rf}) \wedge$
  $(\forall a.\ \forall a'.\ \forall b.\ a \xrightarrow{rf} b \wedge a' \xrightarrow{rf} b \implies (a = a')) \wedge$
  $(\forall (a,b) \in \xrightarrow{rf}.$
    $\text{same\_location } a\ b \wedge$
    $(\text{value\_read } b = \text{value\_written } a) \wedge$
    $(a \neq b) \wedge$
    $(\text{is\_at\_mutex\_location } a \implies$
      $\text{is\_unlock } a \wedge \text{is\_lock } b) \wedge$
    $(\text{is\_at\_non\_atomic\_location } a \implies$
      $\text{is\_store } a \wedge \text{is\_load } b) \wedge$
    $(\text{is\_at\_atomic\_location } a \implies$
      $(\text{is\_atomic\_store } a \vee \text{is\_atomic\_rmw } a \vee \text{is\_store } a)$
      $\wedge (\text{is\_atomic\_load } b \vee \text{is\_atomic\_rmw } b \vee \text{is\_load } b)))$
```

6.10 Consistent locks

The set of all lock or unlock actions in a set as at a location is the set of actions in as that are either locks or unlocks at that location.

```
$\text{all\_lock\_or\_unlock\_actions\_at } \mathit{lopt}\ \mathit{as} =$
  $\{a \in \mathit{as}.\ \text{is\_lock\_or\_unlock } a \wedge (\text{location } a = \mathit{lopt})\}$
```

For all locations that the actions act on, if the location is of mutex kind then, with respect to a $\xrightarrow{lock\_order}$ relation that is $\xrightarrow{sc}$ restricted to the lock and unlock actions at the location,

1. $\xrightarrow{lock\_order}$ is a strict total order over the lock and unlock actions at the location (29.3p2),
2. for all lock and unlock actions at the location, if the action is an unlock, then there exists another action adjacent-less-than it in $\xrightarrow{lock\_rder}$ and on the same thread (30.4.1),
3. for all lock and unlock actions at the location, if the action is a lock, then all actions adjacent-less-than it in $\xrightarrow{lock\_order}$ must be unlocks (30.4.1).

```
$\text{consistent\_locks} =$
$\forall l \in \text{locations\_of } \mathit{actions}.\ (\textit{location-kind } l = \textsc{Mutex}) \implies ($
  $\textbf{let } \mathit{lock\_unlock\_actions} =$
    $\text{all\_lock\_or\_unlock\_actions\_at } (\textsc{Some } l)\mathit{actions} \textbf{ in}$
  $\textbf{let } \mathit{lock\_order} = \xrightarrow{sc}|_{\mathit{lock\_unlock\_actions}} \textbf{ in}$
  (* 30.4.1:5 - The implementation shall serialize those (lock and
  unlock) operations. *)
  $\text{strict\_total\_order\_over } \mathit{lock\_unlock\_actions}\ \mathit{lock\_order} \wedge$

  (* 30.4.1:1 A thread owns a mutex from the time it successfully
  calls one of the lock functions until it calls unlock.*)
  (* 30.4.1:20 Requires: The calling thread shall own the mutex. *)
  (* 30.4.1:21 Effects: Releases the calling threads ownership of the
  mutex.*)
  $(\forall a_u \in \mathit{lock\_unlock\_actions}.\ \text{is\_unlock } a_u \implies$
    $(\exists a_l \in \mathit{lock\_unlock\_actions}.$
      $a_l \xmapsto{lock\_order} a_u \wedge \text{same\_thread } a_l\ a_u \wedge \text{is\_lock } a_l)) \wedge$

  (* 30.4.1:7 Effects: Blocks the calling thread until ownership of
  the mutex can be obtained for the calling thread.*)
  (* 30.4.1:8 Postcondition: The calling thread owns the mutex. *)
  $(\forall a_l \in \mathit{lock\_unlock\_actions}.\ \text{is\_lock } a_l \implies$
    $(\forall a_u \in \mathit{lock\_unlock\_actions}.$
      $a_u \xmapsto{lock\_order} a_l \implies \text{is\_unlock } a_u)))$
```

6.11 Release sequences

In specifying inter-thread synchronization we use the notion of a *release sequence*, headed by a release and followed by a sequence of writes that follow in modification order. Each release sequence element must be on the same thread as the head or an atomic-read-modify-write (1.10p6).

$$\begin{array}{l}
\text{rs\_element } \mathit{rs\_head}\ a = \\
\quad \text{same\_thread}\ a\ \mathit{rs\_head} \lor \text{is\_atomic\_rmw}\ a
\end{array}$$

$\xrightarrow{\textit{release-sequence}}$ is a relation from the release to each of the writes in the sequence. The relation has edges from a release a to an action b that is at an atomic location such that either a and b are the same action or

1. b is a valid release sequence element with respect to the head a
2. $a \xrightarrow{\textit{modification-order}} b$ and
3. all intervening elements between a and b in *modification-order* are valid release sequence elements.

$$\begin{array}{l}
a_{rel} \xrightarrow{\textit{release-sequence}} b = \\
\quad \text{is\_at\_atomic\_location}\ b \land \\
\quad \text{is\_release}\ a_{rel} \land (\\
\qquad (b = a_{rel}) \lor \\
\qquad (\text{rs\_element}\ a_{rel}\ b \land a_{rel} \xrightarrow{\textit{modification-order}} b \land \\
\qquad\quad (\forall c.\ a_{rel} \xrightarrow{\textit{modification-order}} c \xrightarrow{\textit{modification-order}} b \implies \\
\qquad\qquad \text{rs\_element}\ a_{rel}\ c)))
\end{array}$$

$\xrightarrow{\textit{hypothetical-release-sequence}}$ is relation equal to $\xrightarrow{\textit{release-sequence}}$ for the head if it were a release and its definition simply omits that requirement below (29.8).

$$\begin{array}{l}
a \xrightarrow{\textit{hypothetical-release-sequence}} b = \\
\quad \text{is\_at\_atomic\_location}\ b \land (\\
\quad (b = a) \lor \\
\quad (\text{rs\_element}\ a\ b \land a \xrightarrow{\textit{modification-order}} b \land \\
\qquad (\forall c.\ a \xrightarrow{\textit{modification-order}} c \xrightarrow{\textit{modification-order}} b \implies \\
\qquad\quad \text{rs\_element}\ a\ c)))
\end{array}$$

6.12 Synchronizes-with

$a \xrightarrow{\textit{synchronizes-with}} b$ if

1. $a \xrightarrow{\textit{additional-synchronized-with}} b$ or
2. actions a and b are at the same location and
   (a) a is an unlock, b is a lock and $a \xrightarrow{\textit{sc}} b$ (1.10p7),
   (b) a is a release, b is an acquire, a and b are on different threads, and b reads from an action in the release sequence of a (1.10p7),
   (c) a is a release fence, b is an acquire fence, and there exist atomic write x and atomic read y on the same location such that $a \xrightarrow{\textit{sequenced-before}} x$, $y \xrightarrow{\textit{sequenced-before}} b$ and y reads from an action in the release sequence of x (29.8p2),
   (d) a is a release fence, b is an atomic acquire, and there exist atomic write x on the same location as b such that $a \xrightarrow{\textit{sequenced-before}} x$, b reads from an action in the hypothetical release sequence of x (29.8p3),
   (e) a is an atomic release, b is an acquire fence, and there exist atomic action x on the same location as a such that $x \xrightarrow{\textit{sequenced-before}} b$, and x reads from an action in the release sequence of a (29.8p4).

$$\begin{array}{l}
a \xrightarrow{\textit{synchronizes-with}} b = \\
\quad \text{(* -- additional synchronization, from thread create etc. -- *)} \\
\quad a \xrightarrow{\textit{additional-synchronized-with}} b \lor \\
\\
\quad (\text{same\_location}\ a\ b \land a \in \mathit{actions} \land b \in \mathit{actions} \land (\\
\qquad \text{(* -- mutex synchronization -- *)} \\
\qquad (\text{is\_unlock}\ a \land \text{is\_lock}\ b \land a \xrightarrow{\textit{sc}} b) \lor \\
\\
\qquad \text{(* -- release/acquire synchronization -- *)} \\
\qquad (\text{is\_release}\ a \land \text{is\_acquire}\ b \land \neg\, \text{same\_thread}\ a\ b \land \\
\qquad (\exists c.\ a \xrightarrow{\textit{release-sequence}} c \xrightarrow{\textit{rf}} b)) \lor \\
\\
\qquad \text{(* -- fence synchronization -- *)} \\
\qquad (\text{is\_fence}\ a \land \text{is\_release}\ a \land \text{is\_fence}\ b \land \text{is\_acquire}\ b \land \\
\qquad (\exists x.\ \exists y.\ \text{same\_location}\ x\ y \land \\
\qquad\quad \text{is\_atomic\_action}\ x \land \text{is\_atomic\_action}\ y \land \text{is\_write}\ x \land \\
\qquad\quad a \xrightarrow{\textit{sequenced-before}} x \land y \xrightarrow{\textit{sequenced-before}} b \land \\
\qquad\quad (\exists z.\ x \xrightarrow{\textit{hypothetical-release-sequence}} z \xrightarrow{\textit{rf}} y))) \lor \\
\\
\qquad (\text{is\_fence}\ a \land \text{is\_release}\ a \land \\
\qquad \text{is\_atomic\_action}\ b \land \text{is\_acquire}\ b \land \\
\qquad (\exists x.\ \text{same\_location}\ x\ b \land \\
\qquad\quad \text{is\_atomic\_action}\ x \land \text{is\_write}\ x \land \\
\qquad\quad a \xrightarrow{\textit{sequenced-before}} x \land \\
\qquad\quad (\exists z.\ x \xrightarrow{\textit{hypothetical-release-sequence}} z \xrightarrow{\textit{rf}} b))) \lor \\
\\
\qquad (\text{is\_atomic\_action}\ a \land \text{is\_release}\ a \land \\
\qquad \text{is\_fence}\ b \land \text{is\_acquire}\ b \land \\
\qquad (\exists x.\ \text{same\_location}\ a\ x \land \text{is\_atomic\_action}\ x \land \\
\qquad\quad x \xrightarrow{\textit{sequenced-before}} b \land \\
\qquad\quad (\exists z.\ a \xrightarrow{\textit{release-sequence}} z \xrightarrow{\textit{rf}} x)))))
\end{array}$$

6.13 Carries-a-dependency-to

$a \xrightarrow{\textit{carries-a-dependency-to}} b$ is the transitive closure of the union of $\xrightarrow{\textit{data-dependency}}$ with the intersection of $\xrightarrow{\textit{rf}}$ and $\xrightarrow{\textit{sequenced-before}}$ (1.10p8). It is essentially threadwise data dependence.

$$\begin{array}{l}
a \xrightarrow{\textit{carries-a-dependency-to}} b = \\
\quad a\ ((\xrightarrow{\textit{rf}} \cap \xrightarrow{\textit{sequenced-before}}) \cup \xrightarrow{\textit{data-dependency}})^{+}\ b
\end{array}$$

6.14 Dependency-ordered-before

$a \xrightarrow{\textit{dependency-ordered-before}} d$ if a and d are actions and there exists a consume action b that reads from the release sequence of a (a release action) and either $b \xrightarrow{\textit{carries-a-dependency-to}} d$ or b and d are the same action (1.10p9).

$$\begin{array}{l}
a \xrightarrow{\textit{dependency-ordered-before}} d = \\
\quad a \in \mathit{actions} \land d \in \mathit{actions} \land \\
\quad (\exists b.\ \text{is\_release}\ a \land \text{is\_consume}\ b \land \\
\qquad (\exists e.\ a \xrightarrow{\textit{release-sequence}} e \xrightarrow{\textit{rf}} b) \land \\
\qquad (b \xrightarrow{\textit{carries-a-dependency-to}} d \lor (b = d)))
\end{array}$$

6.15 Inter-thread-happens-before and happens-before

To define $\xrightarrow{\textit{inter-thread-happens-before}}$ we first construct $\xrightarrow{r}$, the union of $\xrightarrow{\textit{synchronizes-with}}$, $\xrightarrow{\textit{dependency-ordered-before}}$ and the composition of $\xrightarrow{\textit{synchronizes-with}}$ and $\xrightarrow{\textit{sequenced-before}}$. $\xrightarrow{\textit{inter-thread-happens-before}}$ is the transitive closure of the union of $\xrightarrow{r}$ and the composition of $\xrightarrow{\textit{sequenced-before}}$ and $\xrightarrow{r}$ (1.10p10).

```
$\xrightarrow{\textit{inter-thread-happens-before}} =$
  let $r = \xrightarrow{\textit{synchronizes-with}} \cup$
          $\xrightarrow{\textit{dependency-ordered-before}} \cup$
          $(\xrightarrow{\textit{synchronizes-with}} \circ \xrightarrow{\textit{sequenced-before}})$ in
  $(\xrightarrow{r} \cup (\xrightarrow{\textit{sequenced-before}} \circ \xrightarrow{r}))^+$
```

(This definition is in a different form to that of N3092, defined just using transitive closure rather than an inductive definition. That makes it simpler to work with, and the two are provably equivalent.)

A consistent $\xrightarrow{\textit{inter-thread-happens-before}}$ is irreflexive. It is a transitive collection of orderings that has been carefully constructed not to include sequenced before.

```
consistent_inter_thread_happens_before =
  irreflexive $(\xrightarrow{\textit{inter-thread-happens-before}})$
```

$\xrightarrow{\textit{happens-before}}$ is the union of $\xrightarrow{\textit{sequenced-before}}$ and $\xrightarrow{\textit{inter-thread-happens-before}}$ (1.10p11).

```
$\xrightarrow{\textit{happens-before}} =$
  $\xrightarrow{\textit{sequenced-before}} \cup \xrightarrow{\textit{inter-thread-happens-before}}$
```

6.16 Consistent SC order

The the set of all sequentially consistent actions includes only those that have sequentially consistent order or are locks or unlocks (29.3p2).

```
all_sc_actions =
  $\{a.\ (\text{is\_seq\_cst}\ \ a \lor \text{is\_lock}\ \ a \lor \text{is\_unlock}\ \ a)\}$
```

A consistent $\xrightarrow{\textit{sc}}$ relation is a strict total order over all sequentially consistent actions such that $\xrightarrow{\textit{happens-before}}$ and $\xrightarrow{\textit{modification-order}}$ restricted to all sequentially consistent actions are each subsets of $\xrightarrow{\textit{sc}}$ (29.3p2).

```
consistent_sc_order =
  let sc_happens_before = $\xrightarrow{\textit{happens-before}}|_{\text{all\_sc\_actions}}$ in
  let sc_mod_order = $\xrightarrow{\textit{modification-order}}|_{\text{all\_sc\_actions}}$ in
  strict_total_order_over all_sc_actions $(\xrightarrow{\textit{sc}}) \land$
  $\xrightarrow{\text{sc\_happens\_before}} \subseteq \xrightarrow{\textit{sc}} \land$
  $\xrightarrow{\text{sc\_mod\_order}} \subseteq \xrightarrow{\textit{sc}}$
```

6.17 Consistent modification order

$\xrightarrow{\textit{modification-order}}$ is consistent if (1.10p5) for all a and b, $a \xrightarrow{\textit{modification-order}} b$ implies that a and b are at the same location and for all locations of the actions

1. if the location is atomic, $\xrightarrow{\textit{modification-order}}$ restricted to the actions at the location is a strict total order over the writes to the location, $\xrightarrow{\textit{happens-before}}$ restricted to the writes at the location is a subset of $\xrightarrow{\textit{modification-order}}$ and the restriction to the writes at the location of the composition of $\xrightarrow{\textit{sequenced-before}}$, $\xrightarrow{\textit{sc}}$ restricted to fences and $\xrightarrow{\textit{sequenced-before}}$ is a subset of modification order (29.3p6).
2. Otherwise $\xrightarrow{\textit{modification-order}}$ restricted to the actions at the location is empty.

Modification order is a per-location total order of memory writes.

```
consistent_modification_order =
  $(\forall a.\ \forall b.\ a \xrightarrow{\textit{modification-order}} b \implies \text{same\_location}\ \ a\ b) \land$
  $(\forall l \in \text{locations\_of}\ \ actions.$ case location-kind $l$ of
      ATOMIC $\rightarrow$ (
        let $actions\_at\_l = \{a.\ (\text{location}\ \ a = \text{SOME}\ \ l)\}$ in
        let $writes\_at\_l = \{a\_at\_l.\ (\text{is\_store}\ \ a \lor$
          $\text{is\_atomic\_store}\ \ a \lor \text{is\_atomic\_rmw}\ \ a)\}$ in
            strict_total_order_over $writes\_at\_l$
            $(\xrightarrow{\textit{modification-order}}|_{actions\_at\_l}) \land$
            (* happens-before at the writes of $l$ is a subset of mo
            for $l$ *)
            $\xrightarrow{\textit{happens-before}}|_{writes\_at\_l} \subseteq \xrightarrow{\textit{modification-order}} \land$
            (* MO_SEQ_CST fences impose modification order *)
            $(\xrightarrow{\textit{sequenced-before}} \circ (\xrightarrow{\textit{sc}}|_{\text{is\_fence}}) \circ \xrightarrow{\textit{sequenced-before}}|_{writes\_at\_l})$
               $\subseteq \xrightarrow{\textit{modification-order}})$
    $\parallel$ _ $\rightarrow$ (
        let $actions\_at\_l = \{a.\ (\text{location}\ \ a = \text{SOME}\ \ l)\}$ in
        $(\xrightarrow{\textit{modification-order}}|_{actions\_at\_l}) = \{\}))$
```

6.18 Visible side effects and visible sequences of side effects

$a \xrightarrow{\textit{visible-side-effect}} b$ if (1.10p12)

1. $a \xrightarrow{\textit{happens-before}} b$,
2. a is a write,
3. b is a read,
4. a and b are at the same location and
5. there is no write c at the same location, not equal to either a or b, such that $a \xrightarrow{\textit{happens-before}} c$ and c *happens-before* b.

```
$a \xrightarrow{\textit{visible-side-effect}} b =$
  $a \xrightarrow{\textit{happens-before}} b \land$
  $\text{is\_write}\ \ a \land \text{is\_read}\ \ b \land \text{same\_location}\ \ a\ b \land$
  $\neg(\exists c.\ (c \neq a) \land (c \neq b) \land$
      $\text{is\_write}\ \ c \land \text{same\_location}\ \ c\ b \land$
    $a \xrightarrow{\textit{happens-before}} c \xrightarrow{\textit{happens-before}} b)$
```

The tail of a *visible sequence of side effects* with respect to a particular read and one of its *visible-side-effect*s is a set of actions that follow the *visible-side-effect* in $\xrightarrow{\textit{modification-order}}$ and do not come after the read in $\xrightarrow{\textit{happens-before}}$ such that there is no intervening action in $\xrightarrow{\textit{modification-order}}$ that follows the read in $\xrightarrow{\textit{happens-before}}$ (1.10p13).

```
visible_sequence_of_side_effects_tail $vsse\_head\ b$ =
  $\{c.\ vsse\_head \xrightarrow{\textit{modification-order}} c \land$
    $\neg(b \xrightarrow{\textit{happens-before}} c) \land$
    $(\forall a.\ vsse\_head \xrightarrow{\textit{modification-order}} a \xrightarrow{\textit{modification-order}} c$
       $\implies \neg(b \xrightarrow{\textit{happens-before}} a))\}$
```

visible-sequences-of-side-effects is a map from a *visible-side-effect* and read pair to the pair consisting of the read and, if the read is at an atomic location,*visible-side-effect* union the tail of the visible sequence of side effects, or if not, the empty set.

```
visible_sequences_of_side_effects =
  $\lambda(vsse\_head, b).$
    $(b,$ if is_at_atomic_location $b$ then
           $\{vsse\_head\} \cup$
           visible_sequence_of_side_effects_tail $vsse\_head\ b$
         else
           $\{\})$
```

6.19 Consistent reads-from mapping

$\xrightarrow{\textit{rf}}$ is consistent if

1. for all reads b at non-atomic locations, $a \xrightarrow{\textit{rf}} b$ for some visible side effect a if one exists. There is no edge in $\xrightarrow{\textit{rf}}$ if b has no *visible-side-effects* (1.10p12).
2. for all reads b at atomic locations, $c \xrightarrow{\textit{rf}} b$ for some c in a visible sequence of side effects of b if one exists. There is no edge in $\xrightarrow{\textit{rf}}$ if b has no *visible-sequences-of-side-effects* (1.10p13).
3. For all $x \xrightarrow{\textit{rf}} a$ and $y \xrightarrow{\textit{rf}} b$, if $a \xrightarrow{\textit{happens-before}} b$ and both a and b are at the same atomic location then either x and y are the same location or $x \xrightarrow{\textit{modification-order}} y$ (1.10p13).
4. For all $a \xrightarrow{\textit{happens-before}} b$ and $c \xrightarrow{\textit{rf}} b$, if a is a write at the same location as b, and it is an atomic location, then either c and a are the same location or $a \xrightarrow{\textit{modification-order}} c$ (6.10).
5. For all $a \xrightarrow{\textit{happens-before}} b$ and $c \xrightarrow{\textit{rf}} a$, if b is a write at the same location as a, and it is an atomic location, then $c \xrightarrow{\textit{modification-order}} b$ (6.10).
6. If $a \xrightarrow{\textit{rf}} b$ and b is an atomic-read-modify-write then a is the last preceding action in modification order.
7. If $a \xrightarrow{\textit{rf}} b$ and b is sequentially consistent then a is either the last preceding element in $\xrightarrow{\textit{sc}}$ restricted to the writes at the same location as the read or a is not sequentially consistent (29.3p2).
8. For all a, y, and edges $x \xrightarrow{\textit{sequenced-before}} b$, if x is sequentially consistent fence, b is an atomic action at the same location as a write a, a is adjacent-less-than x in $\xrightarrow{\textit{sc}}$ and $a \xrightarrow{\textit{rf}} y$ then either y and a are the same action or $a \xrightarrow{\textit{modification-order}} y$ (29.3p3).
9. For all edges $a \xrightarrow{\textit{sequenced-before}} x$, and $y \xrightarrow{\textit{rf}} b$, if a is an atomic write on the same location as b, x is a sequentially consistent fence and $\xrightarrow{x}$*sc* b then either y and a are the same action or $a \xrightarrow{\textit{modification-order}} y$ (29.3p4).
10. For all edges $a \xrightarrow{\textit{sequenced-before}} x$, $y \xrightarrow{\textit{sequenced-before}} b$, and z, if a is an atomic write on the same location as atomic action b, y is a sequentially consistent fence $x \xrightarrow{\textit{sc}} y$ and $z \xrightarrow{\textit{rf}} b$ then either z and a are the same action or $a \xrightarrow{\textit{modification-order}} z$ (29.3p5).

```
consistent_reads_from_mapping =
  $(\forall b.\ (\text{is\_read}\ b \wedge \text{is\_at\_non\_atomic\_location}\ b) \implies$
      $(\textbf{if}\ (\exists a_{vse}.\ a_{vse} \xrightarrow{\textit{visible-side-effect}} b)$
      $\textbf{then}\ (\exists a_{vse}.\ a_{vse} \xrightarrow{\textit{visible-side-effect}} b \wedge a_{vse} \xrightarrow{\textit{rf}} b)$
      $\textbf{else}\ \neg(\exists a.\ a \xrightarrow{\textit{rf}} b))) \wedge$

  $(\forall b.\ (\text{is\_read}\ b \wedge \text{is\_at\_atomic\_location}\ b) \implies$
      $(\textbf{if}\ (\exists (b', vsse) \in \textit{visible-sequences-of-side-effects}.\ (b' = b))$
      $\textbf{then}\ (\exists (b', vsse) \in \textit{visible-sequences-of-side-effects}.$
              $(b' = b) \wedge (\exists c \in vsse.\ c \xrightarrow{\textit{rf}} b))$
      $\textbf{else}\ \neg(\exists a.\ a \xrightarrow{\textit{rf}} b))) \wedge$

  $(\forall (x, a) \in \xrightarrow{\textit{rf}}.$
    $\forall (y, b) \in \xrightarrow{\textit{rf}}.$
      $a \xrightarrow{\textit{happens-before}} b \wedge$
      $\text{same\_location}\ a\ b \wedge \text{is\_at\_atomic\_location}\ b$
        $\implies (x = y) \vee x \xrightarrow{\textit{modification-order}} y) \wedge$
(* new CoWR *)
  $(\forall (a, b) \in \xrightarrow{\textit{happens-before}}.$
    $\forall c.$
      $c \xrightarrow{\textit{rf}} b \wedge$
      $\text{is\_write}\ a \wedge \text{same\_location}\ a\ b \wedge \text{is\_at\_atomic\_location}\ b$
        $\implies (c = a) \vee a \xrightarrow{\textit{modification-order}} c) \wedge$
(* new CoRW *)
  $(\forall (a, b) \in \xrightarrow{\textit{happens-before}}.$
    $\forall c.$
      $c \xrightarrow{\textit{rf}} a \wedge$
      $\text{is\_write}\ b \wedge \text{same\_location}\ a\ b \wedge \text{is\_at\_atomic\_location}\ a$
        $\implies c \xrightarrow{\textit{modification-order}} b) \wedge$

  $(\forall (a, b) \in \xrightarrow{\textit{rf}}.\ \text{is\_atomic\_rmw}\ b$
    $\implies a \xmapsto{\textit{modification-order}} b) \wedge$

  $(\forall (a, b) \in \xrightarrow{\textit{rf}}.\ \text{is\_seq\_cst}\ b$
    $\implies \neg\, \text{is\_seq\_cst}\ a \vee$
    $a \xmapsto{\textit{sc}}_{\lambda c.\ \text{is\_write}\ c \wedge \text{same\_location}\ b\ c}\ b) \wedge$

  (* -Fence restrictions- *)

  (* 29.3:3 *)
  $(\forall a.\ \forall (x, b) \in \xrightarrow{\textit{sequenced-before}}.\ \forall y.$
    $(\text{is\_fence}\ x \wedge \text{is\_seq\_cst}\ x \wedge \text{is\_atomic\_action}\ b \wedge$
      $\text{is\_write}\ a \wedge \text{same\_location}\ a\ b \wedge$
      $a \xmapsto{\textit{sc}} x \wedge y \xrightarrow{\textit{rf}} b)$
    $\implies (y = a) \vee a \xrightarrow{\textit{modification-order}} y) \wedge$

  (* 29.3:4 *)
  $(\forall (a, x) \in \xrightarrow{\textit{sequenced-before}}.\ \forall (y, b) \in \xrightarrow{\textit{rf}}.$
    $(\text{is\_atomic\_action}\ a \wedge \text{is\_fence}\ x \wedge \text{is\_seq\_cst}\ x \wedge$
      $\text{is\_write}\ a \wedge \text{same\_location}\ a\ b \wedge$
      $x \xrightarrow{\textit{sc}} b \wedge \text{is\_atomic\_action}\ b)$
    $\implies (y = a) \vee a \xrightarrow{\textit{modification-order}} y) \wedge$

  (* 29.3:5 *)
  $(\forall (a, x) \in \xrightarrow{\textit{sequenced-before}}.\ \forall (y, b) \in \xrightarrow{\textit{sequenced-before}}.\ \forall z.$
    $(\text{is\_atomic\_action}\ a \wedge \text{is\_fence}\ x \wedge \text{is\_seq\_cst}\ x \wedge$
      $\text{is\_write}\ a \wedge \text{is\_fence}\ y \wedge \text{is\_seq\_cst}\ y \wedge$
      $\text{is\_atomic\_action}\ b \wedge$
      $x \xrightarrow{\textit{sc}} y \wedge z \xrightarrow{\textit{rf}} b)$
    $\implies (z = a) \vee a \xrightarrow{\textit{modification-order}} z)$
```

6.20 Consistent control dependency (unused at present)

All data dependency is the transitive closure of the union of $\xrightarrow{\textit{rf}}$ and $\xrightarrow{\textit{carries-a-dependency-to}}$ (1.10p8).

$$\xrightarrow{all\_data\_dependency} = (\xrightarrow{rf} \cup \xrightarrow{\textit{carries-a-dependency-to}})^+$$

$\xrightarrow{\textit{control-dependency}}$ is consistent if the transitive closure of the union of $\xrightarrow{\textit{data-dependency}}$ and $\xrightarrow{\textit{control-dependency}}$ is irreflexive.

$$\text{consistent\_control\_dependency} = \text{irreflexive}\ ((\xrightarrow{\textit{control-dependency}} \cup \xrightarrow{all\_data\_dependency})^+)$$

6.21 Consistent executions

A consistent execution satisfies the predicate below which draws together the different sorts of consistency and well-formedness that we have defined.

```
consistent_execution =
  well_formed_threads ∧
  well_formed_reads_from_mapping ∧
  consistent_locks ∧
  consistent_inter_thread_happens_before ∧
  consistent_sc_order ∧
  consistent_modification_order ∧
  consistent_reads_from_mapping
```

6.22 Sources of undefined behaviour

The *indeterminate reads* of an execution are the set of read actions with no write related to it them $\xrightarrow{rf}$ (1.9p15).

$$\text{indeterminate\_reads} = \{b.\ \text{is\_read}\ b \wedge \neg(\exists a.\ a \xrightarrow{rf} b)\}$$

The *unsequenced races* of an execution are all the pairs of distinct load or store actions at the same location on the same thread of which at least one is a write, such that the two actions are unrelated by $\xrightarrow{\textit{sequenced-before}}$ (1.9p15).

```
unsequenced_races = {(a, b).
  is_load_or_store a ∧ is_load_or_store b ∧
  (a ≠ b) ∧ same_location a b ∧ (is_write a ∨ is_write b) ∧
  same_thread a b ∧
  ¬(a --sequenced-before--> b ∨ b --sequenced-before--> a)}
```

The *data races* of an execution are all the pairs of distinct actions at the same location on different threads of which at least one is a write, it is not the case that both are atomic anf neither happens before the other (1.10p14).

```
data_races = {(a, b).
  (a ≠ b) ∧ same_location a b ∧ (is_write a ∨ is_write b) ∧
  ¬ same_thread a b ∧
  ¬(is_atomic_action a ∧ is_atomic_action b) ∧
  ¬(a --happens-before--> b ∨ b --happens-before--> a)}
```

6.23 C++ memory model

The top-level definition of the memory model takes a program and an operational semantics as parameters. It tests the set of pre-executions that pass the operational semantics and memory model checks for data-races, indeterminate-reads and unsequenced-races, returning an empty set of executions if any are found and the set of pre-executions otherwise.

```
cpp_memory_model opsem (p : program) =
  let pre_executions = {(X_opsem, X_witness).
        opsem p X_opsem ∧
        consistent_execution (X_opsem, X_witness)} in
    if ∃X ∈ pre_executions .
        (indeterminate_reads X ≠ {}) ∨
        (unsequenced_races X ≠ {}) ∨
        (data_races X ≠ {})
    then NONE
    else SOME pre_executions
```

7. Tool support for exploring the model

Given a a relatively complex axiomatic memory model, as we presented in Section 6.1, it is often hard to immediately see the consequences of the axioms, or what behaviour they allow for particular programs. Our CPPMEM tool takes a program in a fragment of C++0x and calculates the set of its executions allowed by the memory model, displaying them graphically.

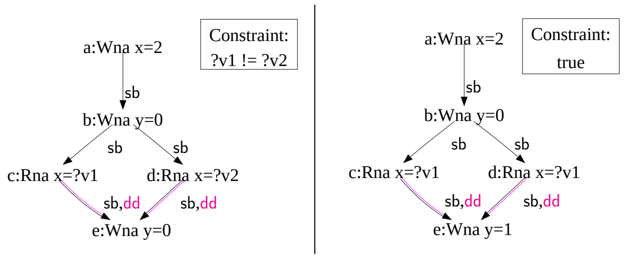
The tool has three main components: an executable symbolic operational semantics to build the X_{opsem} parts of the candidate executions X of a program; a search procedure to enumerate the possible X_{witness} for each of those; and a checking procedure to calculate the derived relations and predicates of the model for each $(X_{\text{opsem}}, X_{\text{witness}})$ pair, to check whether it is consistent and whether it has data races, unsequenced races or indeterminate reads.

Of these, the checker is the most subtle, since the only way to intuitively understand it is to understand the model itself (which is what the tool is intended to aid with), and thus bugs are hard to catch. It also has to be adapted often as the model is developed. We therefore use Isabelle/HOL code generation [Haf09] to build the checker directly from our Isabelle/HOL axiomatisation, to keep the checker and our model in exact correspondence and reduce the possibility for error.

The operational semantics Our overall semantics is stratified: the memory model is expressed as a predicate on the actions and relations of a candidate execution. This means we need an operational semantics of an unusual form to generate all such candidates. In a setting with a global SC memory, the values read by loads can be determined immediately, but here, for example for a program with a single load, in principle we have to generate a large set of executions, each with a load event with one of the possible values. We make this executable by building a symbolic semantics in which the values in actions can be either concrete values or unification variables (shown as $?v$). Control flow can depend on the values read, so the semantics builds a set of these actions (and the associated relations), together with constraints on the values, for each control-flow path of the program. For each path, the associated constraint is solved at the end; those with unsatisfiable constraints (indicating unreachable execution paths) are discarded.

The tool is designed to support litmus test examples of the kind we have seen, not arbitrary C++ code. These do not usually involve many C++ features, and the constraints required are propositional formulae over equality and inequality constraints over symbolic and concrete values. It is not usually important in litmus tests to do more arithmetic reasoning; one could imagine using an SMT solver if that were needed, but for the current constraint language, a standard union-find unifier suffices. The input program is processed by the CIL parser [NMRW02], extended with support for atomics. We use Graphviz [GN00] to generate output. We also allow the user to add explicit constraints on the value read by a memory load in a C++ source program, to pick out candidate executions of interest; to selectively disable some of the checks of the model; and to declutter the output by suppressing actions and edges.

As an example, consider the first program we saw, in §6.10. There are two possibilities: the reads of x either read the same value or different values, and hence the operational semantics gives the two candidate executions and constraints below:



Later, the memory model will rule out the left execution, since there is no way to read anything but 2 at x.

The semantics maintains an environment mapping identifiers to locations. For loads, the relevant location is found in that, and a fresh variable $?v$ is generated to represent the value read.

Other constructs typically combine the actions of their subterms and also build the relations (sequenced-before, data-dependency, etc.) of X_{opsem} as appropriate. For example, for the `if` statement, the execution path splits and two execution candidates will be generated. The one for the true branch has an additional constraint, that the value returned by the condition expression is true (in the C/C++ sense , i.e. different from `0`), and the candidate for the false branch constrains the value to be false. There are also additional *sequenced-before* and *control-dependency* edges from the actions in the condition expression to actions in the branch.

Choosing instantiations of existential quantifiers Given the X_{opsem} part of a finite candidate execution, the X_{witness} part is existentially quantified over a finite but potentially large set. In the worst case, with m reads and n writes, all sequentially consistent (atomic), to the same location, and with the same value, there might be $O(m^{(n+1)} \cdot m! \cdot (m+n)!)$ possible choices of an *rf*, *modification-order* and *sc* relation. In practice, though, litmus tests are much simpler: there are typically no more than 2 or 3 writes to any one location, so we avoid coding up a sophisticated memory-model-aware search procedure in favour of keeping this part of the code simple. For the examples shown here, the tool has to check at most a few thousand alternatives, and takes less than 0.2 seconds. The most complex example we tested (IRIW with all SC) had 162,000 cases to try, and the overall time taken was about 5 minutes.

Checking code extracted from Isabelle We use Isabelle/HOL code generation to produce a checker as an OCaml module, which can be linked in with the rest of the CPPSEM tool. Our model is stated in higher-order logic with sets and relations. Restricted to finite sets, the predicates and definitions are almost all directly executable, within the domain of the code generation tool (which implements finite sets by OCaml lists). For a few cases (e.g importantly transitive closure), we had to write a more efficient function and an Isabelle/HOL proof of equivalence. The overall checking time per example is on the order of 10^{-3} seconds, for examples with around 10 actions.

Finite model generation with Nitpick/Kodkod Given the X_{opsem} part of a candidate execution, the space of possible X_{witness} parts which will lead to valid executions can be explored by tools for model generation. We reused the operational semantics above to produce a X_{opsem} from a program, and then posed problems to Nitpick, a finite model generator built into Isabelle [BN10]. Nitpick is a frontend to Kodkod, a model generator for first order logic extended with relations and transitive closure based on a state-of-the-art SAT solver. Nitpick translates higher-order logic formulae to first-order formulae within Kodkod syntax. For small programs, Nitpick can easily find some consistent execution, or report that none such exists, in a few seconds. In particular, for the IRIW-SC example mentioned above, Nitpick takes 130 seconds to report no execution exists, while other examples take around 5 seconds. Of course, Nitpick can also validate an execution X with both parts X_{opsem} and X_{witness} concretely specified, but this is significantly slower than running the Isabelle-extracted validator. The bottleneck here is the translation process, which is quite involved.

8. Correctness of a Proposed x86 Implementation

The C++0x memory model has been designed with compilation to the various target architectures in mind, and prototype implementations of the atomic primitives have been proposed. For example, the following table presents an x86 prototype by Terekhov [Ter08]:

Operation	x86 Implementation	
Load non-SC	mov	
Load Seq_cst	lock xadd(0)	OR: mfence, mov
Store non-SC	mov	
Store Seq_cst	lock xchg	OR: mov , mfence
Fence non-SC	no-op	
Fence Seq_cst	mfence	

This is a simple mapping from individual source-level atomic operations to small fragments of assembly code, abstracting from the vast and unrelated complexities of compilation of a full C++ language (argument evaluation order, object layout, control flow, etc.). Proposals for the Power [MS10] and other architectures follow the same structure, although, as they have more complex memory models than the x86, the assembly code for some of the operations is more intricate.

Verifying that these prototypes are indeed correct implementations of the model is a crucial part of validating the design. Furthermore, as they represent the atomic-operation parts of efficient compilers (albeit without fence optimisations), they can directly form an important part of a verified C++ compiler, or inform the design and verification of a compiler with memory-model-aware optimisations.

Here, we prove a version of the above prototype x86 implementation [Ter08] correct with respect to our x86-TSO semantics [SSZN^{+}09, OSS09, SSO^{+}10]. Following the prototype, we ignore lock and unlock operations, as well as forks and joins, all of which require significant runtime or operating system support in addition to the the x86 hardware. We also ignore sequentially consistent fences for the time being, but cover all other fences. We do consider read-modify-write actions, implementing them with x86 LOCK'd read-modify-writes; and we include non-atomic loads and stores, which can map to multiple x86 loads and stores, respectively. The prototype mapping is simple, and x86-TSO is reasonably well-understood, so this should be seen as a test of the C++ memory model.

In x86-TSO, an operational semantics gives meaning to assembly programs by creating an *x86 event structures* E_{x86} (analogous to X_{opsem}) comprising a set of events, an intra-thread *program-order* relation (analogous to sequenced-before) that orders events according to the program text. cEvents can be reads, writes, or fences, and certain instructions (e.g. CMPXCHG) create *locked* sets of events that execute atomically. Corresponding to X_{witness}, there are *x86 execution witnesses* X_{x86} which comprise a reads-from mapping and a memory order, which is a partial order over reads and writes that is total on the writes. The remainder of the axiomatisations are very different: x86-TSO has no concept of release, acquire, visible side effect, etc.

Abstracting out the rest of the compiler To discuss the correctness of the proposed mapping in isolation, without embarking on a verification of some particular full compiler, we work solely in terms of candidate executions and memory models.

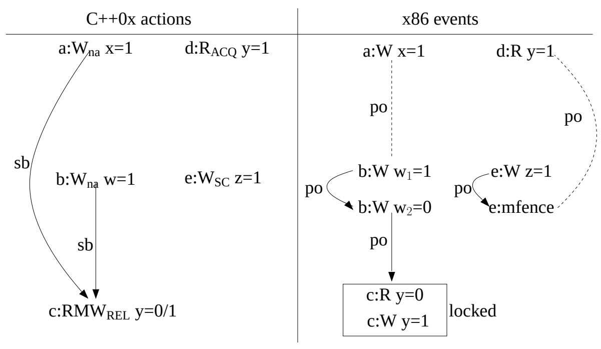
First, we lift the mapping between instructions to a nondeterministic translation $action\_comp$ from C++ actions to small x86 event structures, e.g. relating an atomic read-modify-write action to the events of the corresponding x86 LOCK'd instruction.

To define what it means for the mapping to be correct, suppose we have a C++ program p with no undefined behaviour and an X_{opsem} which is allowed by its operational semantics. We regard an abstract compiler evt_comp as taking such an X_{opsem} and giving an x86 event structure E_{x86}, respecting the $action\_comp$ mapping but with some freedom in the resulting x86 program order.

We say the mapping is correct if given such an abstract compiler, the existence of a valid x86-TSO execution witness for E_{x86} implies the existence of a consistent C++ execution witness X_{witness} for the original actions X_{opsem}. We prove this by lifting such an x86 execution witness to a C++ consistent execution, as illustrated below.

$$\begin{tikzcd}
X_{\mathrm{opsem}} \arrow[r, dotted, "\mathrm{consistent\_execution}"] \arrow[d, "evt\_comp"'] & X_{\mathrm{witness}} \\
E_{\mathrm{x86}} \arrow[r, "\mathrm{valid\_execution}"'] & X_{\mathrm{x86}} \arrow[u, "evt\_comp^{-1}"']
\end{tikzcd}$$

Below we show an X_{opsem} and E_{x86} that could be related by evt_comp. The dotted lines indicate some of the x86 program ordering decisions that the compiler must make, but which evt_comp does not constrain.



In more detail, we use two existentially quantified helper functions $locn\_comp$ and tid_comp to encapsulate the details of a C++ compiler's data layout, its mapping of C++ locations to x86 addresses, and the mapping of C++ threads to x86 threads.

Given a C++ location and value, $locn\_comp$ produces a finite mapping from x86 addresses to x86 values. The domain of the finite map is the set of x86 addresses that corresponds to the C++ location, and the mapping itself indicates how a C++ value is laid out across the x86 addresses. A well-formed $locn\_comp$ has the following properties: it is injective; the address calculation cannot depend on the value; each C++ location has an x86 address; different C++ locations have non-overlapping x86 address sets; and an atomic C++ location has a single x86 address, although a non-atomic location can have several addresses (e.g. for a multi-word object).

Finally, the evt_comp relation specifies valid translations, applying $action\_comp$ with a well-formed $locn\_comp$ and also constraining how events from different actions relate: no single x86 instruction instance can be used by multiple C++ actions, and the x86 *program-order* relation must respect C++'s *sequenced-before*.

The detailed definitions, and the proof of the following theorem, are available online [BOS].

Theorem 1. *Let p be a C++ program that has no undefined behaviour. Suppose also that p contains no SC fences, forks, joins, locks, or unlocks. Then the x86 mapping is correct in the sense above. That is, if $actions$, sequenced-before, and location-kind are members of the X_{opsem} part of a candidate execution resulting from the operational semantics of p, then the following holds:*

$$\begin{aligned}
&\forall comp\ locn\_comp\ tid\_comp\ X_{\mathrm{x86}}.\\
&\quad \mathrm{evt\_comp}\ \ comp\ locn\_comp\ tid\_comp\ actions\\
&\qquad\qquad \textit{sequenced-before location-kind} \wedge\\
&\quad \mathrm{valid\_execution}\ (\cup_{a \in actions}(comp\ a))\ X_{\mathrm{x86}} \Rightarrow\\
&\quad \exists X_{\mathrm{witness}}.\ \mathrm{consistent\_execution}\ (X_{\mathrm{opsem}}, X_{\mathrm{witness}})
\end{aligned}$$

Proof outline. X_{x86} includes a reads-from map and a memory ordering relation that is total on all memory writes. To build X_{witness}, we lift a C++ reads-from map and modification order from these through $comp$ (e.g., $a \xrightarrow{rf} b$ iff $\exists (e_1 \in comp\ a)(e_2 \in comp\ b).\ e_1 \xrightarrow{x86\text{-}rf} e_2$). We create an *sc* ordering by restricting the X_{x86} memory ordering to the events that originate in sequentially consistent atomics, and linearising it using the proof technique from our previous triangular-race freedom work for x86-TSO [Owe10]. We then lift that through $comp$. The proof now proceeds in three steps:

1) We first show that if $a \xrightarrow{happens\text{-}before} b$ and there are x86 events e_1 and e_2 such that $e_1 \in comp\ a$ and $e_2 \in comp\ b$, then e_1 precedes e_2 in either X_{x86}'s memory order or program order. We have machine-checked this step in HOL-4 [HOL].[1]

This property establishes that, in some sense, x86-TSO has a stronger memory model than C++, and so any behaviour allowed by the former should be allowed by the latter. However, things are not quite so straightforward.

2) Check that X_{witness} is a $\mathrm{consistent\_execution}$. Most cases are machine checked in HOL; some are only pencil-and-paper. Many rely upon the property from 1. For example, in showing that (at a non-atomic location) if $a \xrightarrow{rf} b$ then $a \xrightarrow{visible\text{-}side\text{-}effect} b$, we note that if there were a write c to the same location such that $a \xrightarrow{happens\text{-}before} c \xrightarrow{happens\text{-}before} b$, then using the property from 1, there is an x86 write event in $comp\ \ c$ that would come between the events of $comp\ \ a$ and $comp\ \ b$ in X_{x86}, thus meaning that they would not be in X_{x86}'s reads-from map, contradicting the construction of X_{witness}'s reads from map.

3) In some cases, some of the properties required for 2 might be false. For example, in showing that $a \xrightarrow{rf} b$ implies $a \xrightarrow{visible\text{-}side\text{-}effect} b$, we need to show that $a \xrightarrow{happens\text{-}before} b$. Even though there is such a relationship at the x86 level, it does not necessarily exist in C++. In general, x86 executions can establish reads-from relations that are prohibited in C++. Similarly, for non-atomic accesses that span multiple x86 addresses, the lifted reads from-map might not be well-formed.

We show that if one of these violations of 2 arises, then the original C++ program has a data race. We find a minimum violation in X_{x86}, again using techniques from our previous work [Owe10]. Next we can remove the violation, resulting in a consistent X_{witness} for a prefix of the execution, then we add the bad action, note that it creates a data race, and allow the program to complete in any way. The details of this part are by pencil-and-paper proof.

□

[1] The C++ model is in Isabelle/HOL, but x86-TSO is in HOL-4. We support the proof with a semi-automated translation from Isabelle/HOL to HOL-4.

Sequentially consistent atomics The proposal above includes two implementations of sequentially consistent atomic reads and writes; one with the x86 locked instructions, and the other with fence instructions on both the reads and writes. However, we can prove that it suffices either to place an mfence before every sc read, or after every sc write, but that it is not necessary to do both.

This optimisation is a direct result of using triangular-race freedom (TRF) [Owe10] to construct the *sc* ordering in proving Theorem 1. Roughly, our TRF theorem characterises when x86-TSO executions are not sequentially consistent; it uses a pattern, called a triangular race, involving an x86-level data race combined with a write followed, on the same thread, by a read without a fence (or locked instruction) in between. If no such pattern exists, then an execution X_{x86} can be linearised such that each read reads from the most recent preceding write.

Although the entirety of an execution witness X_{x86} might contain triangular races and therefore not be linearisable, by restricting attention to only sc reads and writes we get a subset of the execution that is TRF, as long as there is a fence between each sc read and write on the same thread. Linearising this subset guarantees the relevant property of X_{witness}'s *sc* ordering: that if a and b are sequentially consistent atomics and $a \xrightarrow{\mathit{rf}} b$, then a immediately precedes b in *sc* restricted to that address.

Compiler correctness Although we translate executions instead of source code, Theorem 1 could be applied to full source-to-assembly compilers that follow the prototype implementation. The following diagram presents the overall correctness property.

$$\begin{tikzcd}
p \arrow[r, dotted, "\text{w.f. threads}"] \arrow[d, "\text{compiler}"'] & X_{\mathrm{opsem}} \arrow[r, dotted, "\text{consistent\_execution}"] & X_{\mathrm{witness}} \\
p' \arrow[r, "\text{w.f. events}"'] & E_{\mathrm{x86}} \arrow[u, "f"] \arrow[r, "\textit{valid\_execution}"'] & X_{\mathrm{x86}} \arrow[u, "g"']
\end{tikzcd}$$

If, once we use f, we can then apply evt_comp to get the same event set back, i.e., informally, $evt\_comp(f(E)) = E$, then Theorem 1 ensures that the compiler respects the memory model, and so we only need to verify that it respects the operational semantics. Thus, our result applies to compilers that do not optimise away any instructions that evt_comp will produce. These restrictions apply to the code generation phase; the compiler can perform any valid source-to-source optimisations before generating x86 code.

9. Related work

The starting points for this work were the draft standard itself and the work of Boehm and Adve [BA08], which introduced the rationale for the C++0x overall design and gave a model for non-atomic, lock, and SC atomic operations, without going into low-level atomics or fences in any detail. It was expressed in informal mathematics, an intermediate point between the prose of the standard and the mechanised definitions of our model. The most closely related other work is the extensive line of research on the Java Memory Model (JMM) [Pug00, MPA05, CKS07, vA08, TVD10]. Java imposes very different constraints to C++ as there it is essential to prohibit thin-air reads, to prevent forging of pointers and hence security violations.

There is also a body of research on tool support for memory models, notably including (among others) the MEMSAT of Torlak et al. [TVD10], which uses Kodkod for formalisations of the JMM, and NEMOSFINDER of Yang et al. [YGLS04], which is based on Prolog encodings of memory models and included an Itanium specification. Building on our previous experience with the MEMEVENTS tool for hardware (x86 and Power) memory models [SSZN$^+$09, OSS09, SSO$^+$10, AMSS10], we designed CPPMEM to eliminate the need for hand-coding of the tool to reflect changes in the model, by automatically generating the checker code from the Isabelle/HOL definition. We made it practically usable for exploring our non-idealised (and hence rather complex) C++0x model by a variety of user-interface features, letting us explore the executions of a program in various ways.

Turning to the sequential semantics of C++, Norrish has recently produced an extensive HOL4 model [Nor08], and Zalewski [Zal08] formalised the proposed extension of C++ concepts.

Acknowledgements This work would not have been possible without discussions with members of the C++ Concurrency subcommittee and the `cpp-threads` mailing list, including Hans Boehm, Lawrence Crowl, Peter Dimov, Benjamin Kosnik, Doug Lea, Nick Maclaren, Paul McKenney, Clark Nelson, Anthony Williams, and Michael Wong. We acknowledge funding from EPSRC grants EP/F036345, EP/H005633, EP/H027351, and EP/F067909.

References

[AB10] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *C. ACM*, 2010. To appear.

[AMSS10] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proc. CAV*, 2010.

[ARM08] ARM. *ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition)*. April 2008.

[BA08] H.-J. Boehm and S.V. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, 2008.

[Bec10] P. Becker, editor. *Programming Languages — C++. Final Committee Draft.* March 2010. ISO/IEC JTC1 SC22 WG21 N3092.

[BN10] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Proc. ITP*, 2010.

[BOS] `www.cl.cam.ac.uk/users/pes20/cpp`.

[C1X] JTC1/SC22/WG14 — C. `http://www.open-std.org/jtc1/sc22/wg14/`.

[CKS07] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *Proc. ESOP*, 2007.

[GN00] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, 2000.

[Haf09] Florian Haftmann. *Code Generation from Specifications in Higher-Order Logic*. PhD thesis, TU München, 2009.

[HOL] The HOL 4 system. `http://hol.sourceforge.net/`.

[Int02] Intel. A formal specification of Intel Itanium processor family memory ordering. `http://www.intel.com/design/itanium/downloads/251429.htm`, October 2002.

[Isa] Isabelle 2009-2. `http://isabelle.in.tum.de/`.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, 1979.

[MPA05] J. Manson, W. Pugh, and S.V. Adve. The Java memory model. In *Proc. POPL*, 2005.

[MS10] P. E. McKenney and R. Silvera. Example POWER implementation for C/C++ memory model. `http://www.rdrop.com/users/paulmck/scalability/paper/N2745r.2010.02.19a.html`, 2010.

[MW] P. E. McKenney and J. Walpole. What is RCU, fundamentally? Linux Weekly News, `http://lwn.net/Articles/262464/`.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proc. CC*, 2002.

[Nor08] M. Norrish. A formal semantics for C++. Technical report, NICTA, 2008.

[OSS09] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proc. TPHOLs*, 2009.

[Owe10] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *Proc. ECOOP*, 2010.

[Pow09] *Power ISA Version 2.06*. IBM, 2009.

[Pug00] W. Pugh. The Java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6), 2000.

[Spa] The SPARC architecture manual, v. 9. `http://developers.sun.com/solaris/articles/sparcv9.pdf`.

[SSO$^+$10] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *C. ACM*, 53(7):89–97, 2010.

[SSZN$^+$09] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. POPL*, 2009.

[Ter08] A. Terekhov. Brief tentative example x86 implementation for C/C++ memory model. `cpp-threads` mailing list, `http://www.decadent.org.uk/pipermail/cpp-threads/2008-December/001933.html`, Dec. 2008.

[TJ07] E. Torlak and D. Jackson. Kodkod: a relational model finder. In *Proc. TACAS*, 2007.

[TVD10] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: checking axiomatic specifications of memory models. In *PLDI*, 2010.

[vA08] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, 2008.

[YGLS04] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS*, 2004.

[Zal08] M. Zalewski. *Generic Programming with Concepts*. PhD thesis, Chalmers University, November 2008.