Mathematizing C++ Concurrency

Mark Batty Scott Owens Susmit Sarkar Peter Sewell Tjark Weber
University of Cambridge
http://www.cl.cam.ac.uk/users/pes20/cpp

Abstract

Shared-memory concurrency in C and C++ is pervasive in systems programming, but has long been poorly defined. This motivated an ongoing shared effort by the standards committees to specify concurrent behaviour in the next versions of both languages. They aim to provide strong guarantees for race-free programs, together with new (but subtle) relaxed-memory atomic primitives for high-performance concurrent code. However, the current draft standards, while the result of careful deliberation, are still rather far from clear and rigorous definitions.

In this paper we establish a mathematical (yet readable) semantics for C++ concurrency. We aim to capture the intent of the current draft as closely as possible, but discuss a number of points where this is not straightforward. We prove that a proposed x86 implementation of the concurrency primitives is correct with respect to the x86-TSO model, and describe our CPPMEM tool for exploring the semantics of examples, using code generated from our Isabelle/HOL definitions.

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

Categories and Subject Descriptors C.1.2 [*Multiple Data Stream Architectures (Multiprocessors)*]: Parallel processors; D.1.3 [*Concurrent Programming*]: Parallel programming; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]

General Terms Documentation, Languages, Reliability, Standardization, Theory, Verification

Keywords Relaxed Memory Models, Semantics

1. Introduction

Context Systems programming, of OS kernels, language runtimes, etc., commonly rests on shared-memory concurrency in C or C++. These languages are defined by informal-prose 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) is expected to follow suit [C1X].

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. [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 obviously the result of extensive and careful deliberation. However, when one looks more closely, it is still rather far from a 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 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.

Contributions In this paper we establish a mathematically rigorous semantics for C++ concurrency, described in Section 2 and with further examples in Section 3. It is *precise*, formalised in

Isabelle/HOL [Isa], and is *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.

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 4. The standards process for C++0x is ongoing: the current version is the 'final committee draft', leaving a small window for further improvements. That for C1X is at an earlier stage, though the two should be compatible.

As a theoretical test of our semantics, we prove a correctness result (§5) 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.

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 (§6) 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.

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. C++0x Concurrency, as Formalised

Here we describe C++ concurrency incrementally, starting with single-threaded programs and then adding threads and locks, SC atomics, and low-level atomics (release/acquire, relaxed, and release/consume). Our model also covers fences, but we omit the details here. In this section we do not distinguish between the C++ 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 4. Our memory model is expressed as a standalone Isabelle/HOL file and the complete model is available online [BOS]; here we give the main definitions, automatically typeset (and lightly hand-edited in a few cases) from the Isabelle/HOL source.

The semantics of a program p will be a set of allowed *executions* X. Some C++ programs are deemed to have *undefined behaviour*, meaning that an implementation is unconstrained, e.g. if any execution contains a data race. Accordingly, we define the semantics in two phases: first we calculate a set of pre-executions which are admitted by the operational semantics and are *consistent* (defined in the course of this section). Then, if there is a pre-execution in that set with a race of some kind, the semantics indicates undefined behaviour by giving NONE, otherwise it gives all the pre-executions. In more detail, a candidate execution X is a pair $(X_{\text{opsem}}, X_{\text{witness}})$, where the first component is given by the operational semantics and the second is an existential witness of some further data; we introduce the components of both as we go along. The top-level definition of the memory model, then, is:

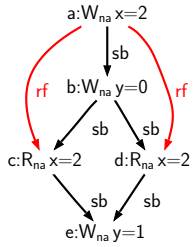
```
cpp_memory_model opsem (p : program) =
  let pre_executions = {(X_opsem, X_witness).
        opsem p X_opsem ∧
        consistent_execution (X_opsem, X_witness)} in
    if ∃X ∈ pre_executions.
        (indeterminate_reads X ≠ {}) ∨
        (unsequenced_races X ≠ {}) ∨
        (data_races X ≠ {})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

2.1 Single-threaded programs

We begin with the fragment of the model that deals with single-threaded programs, which serves to introduce the basic concepts and notation we use later.

As usual for a relaxed memory model, different threads can have quite different views of memory, so the semantics cannot be expressed in terms of changes to a monolithic memory (e.g. a function from locations to values). Instead, an execution consists of a set of *memory actions* and various relations over them, and the memory model axiomatises constraints on those. For example, consider the program on the left below.

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



../examples/t1.c

This has only one execution, shown on the right. There are five actions, labelled (a)–(e), all by the same thread (their thread ids are elided). 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. Evaluation order is expressed by the *sequenced-before* (sb) relation, a strict preorder over the actions, that here does not order (c) and (d). The two reads both *read from* the same write (a), indicated by the *rf* relation. The set of actions and sequenced-before relation are given by the operational semantics (so are part of the X_{opsem}); the *rf* relation is existentially quantified (part of the X_{witness}), as in general there may be many writes that each read might read from.

In a non-SC semantics, the constraint on reads cannot be simply that they read from the 'most recent' write, as there is no global linear time. Instead, they are constrained here using a *happens-before* relation, which in the single-threaded case coincides with sequenced-before. Non-atomic reads have to read from a *visible side effect*, a write to the same location that happens-before the read but is not happens-before-hidden, i.e., one for which there is no intervening write to the location in happens-before. We define the visible-side-effect relation below, writing it with an arrow. The auxiliary functions is_write and is_read pick out all actions (including atomic actions and read-modify-writes but not lock or unlock actions) that write or read memory.

$$
\begin{array}{l}
a \xrightarrow{\textit{visible-side-effect}} b = \\
\quad a \xrightarrow{\textit{happens-before}} b \wedge \\
\quad \mathrm{is\_write}\ a \wedge \mathrm{is\_read}\ b \wedge \mathrm{same\_location}\ a\ b \wedge \\
\quad \neg(\exists c.\ (c \neq a) \wedge (c \neq b) \wedge \\
\qquad \mathrm{is\_write}\ c \wedge \mathrm{same\_location}\ c\ b \wedge \\
\qquad a \xrightarrow{\textit{happens-before}} c \xrightarrow{\textit{happens-before}} b)
\end{array}
$$

The constraint on the values read by nonatomic reads is in two parts: the reads-from map must satisfy a well-formedness condition (not shown here), saying that reads cannot read from multiple writes, that they must be at the same location and have the same value as the write they read from, and so on. More interestingly, it must respect the visible side effects, in the following sense.

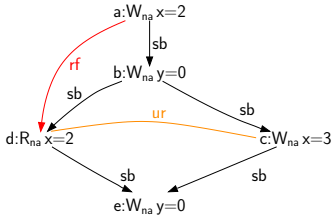
$$
\begin{array}{l}
\mathrm{consistent\_reads\_from\_mapping} = \\
\quad (\forall b.\ (\mathrm{is\_read}\ b \wedge \mathrm{is\_at\_non\_atomic\_location}\ b) \implies \\
\qquad (\textbf{if}\ \ (\exists a_{vse}.\ a_{vse} \xrightarrow{\textit{visible-side-effect}} b) \\
\qquad \textbf{then}\ \ (\exists a_{vse}.\ a_{vse} \xrightarrow{\textit{visible-side-effect}} b \wedge a_{vse} \xrightarrow{rf} b) \\
\qquad \textbf{else}\ \neg(\exists a.\ a \xrightarrow{rf} b))) \wedge \\
\quad [\ldots]
\end{array}
$$

If a read has no visible side effects (e.g. reading an uninitialised variable), there can be no *rf* edge. This is an *indeterminate read*, and the program is deemed to have undefined behaviour.

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

Programs also have undefined behaviour if they have a pre-execution in which there is a write and another access to the same location that are unsequenced, as in the example below, with an *unsequenced-race* (ur) edge shown.

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

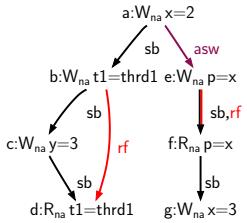


$$
\begin{array}{l}
\mathrm{unsequenced\_races} = \{(a, b). \\
\quad \mathrm{is\_load\_or\_store}\ a \wedge \mathrm{is\_load\_or\_store}\ b \wedge \\
\quad (a \neq b) \wedge \mathrm{same\_location}\ a\ b \wedge (\mathrm{is\_write}\ a \vee \mathrm{is\_write}\ b) \wedge \\
\quad \mathrm{same\_thread}\ a\ b \wedge \\
\quad \neg(a \xrightarrow{\textit{sequenced-before}} b \vee b \xrightarrow{\textit{sequenced-before}} a)\}
\end{array}
$$

2.2 Threads, Data Races, and Locks

We now integrate C++-0x threads into the model. The following 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; }
```

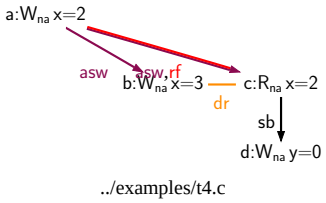


../examples/t3.c

The thread creation gives rise to *additional-synchronized-with* (asw) edges (here $a \xrightarrow{\mathrm{asw}} e$) from sequenced-before-maximal actions of the parent thread before the thread creation to sequenced-before-minimal edges of the child. As we shall see, these edges are also incorporated, indirectly, into happens-before. They are generated by the operational semantics, so are another component of an X_{opsem}.

Thread creation gives rise to many memory actions (for passing function arguments and writing and reading the thread id) which clutter examples, so for this paper we usually use a more concise parallel composition, written {{{ ... ||| ... }}}:

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



../examples/t4.c

This example exhibits a *data race* (dr): two actions at the same location, on different threads, not related by happens-before, at least one of which is a write.

$$
\begin{array}{l}
\mathrm{data\_races} = \{(a, b). \\
\quad (a \neq b) \wedge \mathrm{same\_location}\ a\ b \wedge (\mathrm{is\_write}\ a \vee \mathrm{is\_write}\ b) \wedge \\
\quad \neg\,\mathrm{same\_thread}\ a\ b \wedge \\
\quad \neg(\mathrm{is\_atomic\_action}\ a \wedge \mathrm{is\_atomic\_action}\ b) \wedge \\
\quad \neg(a \xrightarrow{\textit{happens-before}} b \vee b \xrightarrow{\textit{happens-before}} a)\}
\end{array}
$$

If there is a pre-execution of a program that has a data-race, then, as with unsequenced-races, that program has undefined behaviour.

Data races can be prevented by using mutexes, as usual. These give rise to *lock* and *unlock* memory actions on the mutex location, and a pre-execution has a relation, *sc*, as part of X_{witness} that totally orders such actions. A consistent_locks predicate checks that lock and unlock actions are appropriately alternating. Moreover, these actions on each mutex create *synchronizes-with* edges from every unlock to every lock that is ordered after it in *sc*. The synchronizes-with relation is a derived relation, calculated from a candidate execution, which contains mutex edges, the additional-synchronizes-with edges (e.g. from thread creation), and other edges that we will come to.

$a \xrightarrow{\textit{synchronizes-with}} b =$
(* – additional synchronization, from thread create etc. – *)
$a \xrightarrow{\textit{additional-synchronized-with}} b \vee$

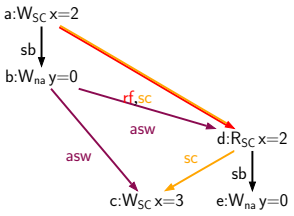
(same_location $a\ b \wedge a \in actions \wedge b \in actions \wedge$ (
(* – mutex synchronization – *)
(is_unlock $a \wedge$ is_lock $b \wedge a \xrightarrow{sc} b$) $\vee$
[. . .]))

For multi-threaded programs with locks but without atomics, happens-before is the transitive closure of the union of the sequenced-before and synchronizes-with relations. The definition of a visible side effect and the conditions on the reads-from relation are unchanged from the single-threaded case.

2.3 SC Atomics

For simple concurrent accesses to shared memory that are not protected by locks, C++0x provides *sequentially consistent atomics*. Altering the racy example from above to use an atomic object `x` and SC atomic operations, we have 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

Semantically, this is because SC atomic operations are totally ordered by *sc*, a total order over the sequentially-consistent operations, 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 the following section.

Initialisation of an atomic object is by non-atomic stores (to avoid the need for a hardware fence for every such initialisation), and those non-atomic stores *can* race with other actions at the location unless the program has some synchronisation. Non-initialization SC-atomic accesses are made with atomic read, write and read-modify-write actions that do not race with each other.

2.4 Low-level Atomics

SC atomics are expensive to implement on most multiprocessors, e.g. with the suggested implementations for an SC atomic load being `LOCK XADD(0)` on x86 [Ter08] and `hwsync; ld; cmp; bc; isync` on Power [MS10]; the LOCK'd instruction and the `hwsync` may take 100s of cycles. They also provide more synchronisation than needed for many concurrent idioms. Accordingly, C++0x includes several weaker variants: atomic actions are parametrised by a *memory order* mo that specifies how much synchronisation and ordering is required. The strongest ordering is required for MO_SEQ_CST actions (which is the default, as used above), and the weakest for MO_RELAXED actions. In between there are MO_RELEASE/MO_ACQUIRE and MO_RELEASE/MO_CONSUME pairs, and MO_ACQ_REL with both acquire and release semantics.

2.5 Types and Relations

Before giving the semantics of low-level atomics, we summarise the types and relations of the model. There are base types of action ids aid, thread ids tid, locations l, and values v. As we have seen already, actions can be non-atomic reads or writes, or mutex locks or unlocks. Additionally, there are atomic reads, writes, and read-modify-writes (with a memory order parameter mo) and fences (also with a mo parameter). We often elide the thread ids.

action =	
$aid, tid{:}\mathrm{R_{na}}\, l = v$	non-atomic read
$aid, tid{:}\mathrm{W_{na}}\, l = v$	non-atomic write
$aid, tid{:}\mathrm{R}_{mo}\, l = v$	atomic read
$aid, tid{:}\mathrm{W}_{mo}\, l = v$	atomic write
$aid, tid{:}\mathrm{RMW}_{mo}\, l = v_1/v_2$	atomic read-modify-write
$aid, tid{:}\mathrm{L}\ l$	lock
$aid, tid{:}\mathrm{U}\ l$	unlock
$aid, tid{:}\mathrm{F}_{mo}$	fence

The is_read predicate picks out non-atomic and atomic reads and atomic read-modify-writes; the is_write predicate picks out non-atomic and atomic writes and atomic read-modify-writes.

Locations are subject to a very weak type system: each location stores a particular kind of object, as determined by a *location-kind* map. 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. These are enforced (among other sanity properties) by a well_formed_threads predicate; we elide the details here.

The X_{opsem} part of a candidate execution X consists of a set of thread ids, a set of actions, a location typing, and four binary relations over its actions: *sequenced-before* (sb), *additional-synchronized-with* (asw), *data-dependency* (dd), and *control-dependency* (cd). We have already seen the first two: *sequenced-before* contains the intra-thread edges imposed by the C++ evaluation order, and *additional-synchronized-with* contains additional edges from thread creation and thread join (among others). Data dependence will be used for require/consume atomics (in §2.8) and control dependence will be used only in discussion, not in the current model. These are all relations that are decided by the syntactic structure of the source code and the path of control flow, and so the set of possible choices for an X_{opsem} can be calculated by the operational semantics without reference to the memory model (with reads taking unconstrained values).

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). The *rf* reads-from map is a relation containing edges to read actions from the write actions whose values they take, and edges to each lock action from the last unlock of its mutex. The sequentially consistent order *sc* is a total order over all actions that are MO_SEQ_CST and all mutex actions. The modification order (mo) is a total order over all writes at each atomic location (leaving writes at different locations unrelated), which will be used to express a coherence condition. These relations are existentially quantified in the definition of cpp_memory_model, and for each X_{opsem} admitted by the operational semantics there may be many choices of an X_{witness} that give a consistent execution (each of which may or may not have a data race, unsequenced race, or indeterminate read).

The happens-before relation, along with several others, are derived from those in X_{opsem} and X_{witness}.

2.6 Release/Acquire Synchronization

An atomic write or fence is a *release* if it has the memory order MO_RELEASE, MO_ACQ_REL or MO_SEQ_CST. Atomic reads or fences with order MO_ACQUIRE, MO_ACQ_REL or MO_SEQ_CST, and fences with order MO_CONSUME, are *acquire* actions.

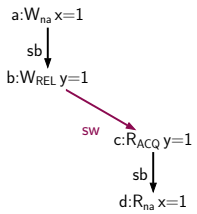
Pairs of a write-release and a read-acquire support the following programming idiom. Here one thread writes some data `x` (perhaps

spanning multiple words) and then sets a flag `y` while the other spins until the flag is set and then reads the data.

```
// sender                 | // receiver
x = ...                   | while (0 == y);
y = 1;                    | r = x;
```

The desired guarantee here is that the receiver must see the data writes of the sender (in more detail, that the receiver cannot see any values of data that precede those writes in modification order). This can be achieved with an atomic store of `y`, annotated MO_RELEASE, and an atomic load of `y` annotated MO_ACQUIRE. The reads and writes of `x` can be nonatomic.

In the model, any instance of a read-acquire that reads from a write-release gives rise to a *synchronizes-with* edge, e.g. as below (where the *rf* edges are suppressed).

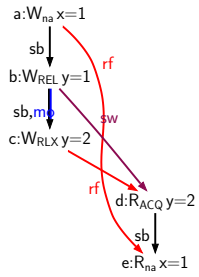


../examples/t15.c

For such programs (in fact for any program without release/consume atomics), happens-before is still the transitive closure of the union of the sequenced-before and synchronizes-with relations, so here $a \xrightarrow{\textit{happens-before}} d$ and (d) is obliged to read from (a).

In this case, the read-acquire synchronizes with the write-release that it reads from. More generally, the read-acquire can synchronize with a write-release (to the same location) that is before the write that it reads from. To define this precisely, we need to use the modification order of a candidate execution and to introduce the derived notion of a *release sequence*, of writes that follow (in some sense) a write-acquire.

For example, in the fragment of an execution below, the read-acquire (d) synchronizes with the write-release (b) by virtue of the fact that (d) reads from another write to the same location, (c), and (b) precedes (c) in the modification order (mo) for that location.



../examples/t8a.c

The modification order of a candidate execution (here $b \xrightarrow{\textit{modification-order}} c$) totally orders all of the write actions on each atomic location, in this case `y`. It must also be consistent with happens-before, in the sense below.

$$
\begin{array}{l}
\text{consistent\_modification\_order} = \\
\quad (\forall a.\ \forall b.\ a \xrightarrow{\textit{modification-order}} b \implies \text{same\_location}\ a\ b) \wedge \\
\quad (\forall l \in \text{locations\_of}\ \textit{actions}.\ \textbf{case}\ \textit{location-kind}\ l\ \textbf{of} \\
\qquad \text{ATOMIC} \rightarrow (\\
\qquad\quad \textbf{let}\ \textit{actions\_at\_l} = \{a.\ (\text{location}\ a = \text{SOME}\ l)\}\ \textbf{in} \\
\qquad\quad \textbf{let}\ \textit{writes\_at\_l} = \{a \in \textit{actions\_at\_l}.\ (\text{is\_store}\ a \vee \\
\qquad\qquad \text{is\_atomic\_store}\ a \vee \text{is\_atomic\_rmw}\ a)\}\ \textbf{in} \\
\qquad\qquad \text{strict\_total\_order\_over}\ \textit{writes\_at\_l} \\
\qquad\qquad (\xrightarrow{\textit{modification-order}}|_{\textit{actions\_at\_l}}) \wedge \\
\qquad\qquad \text{(* happens-before at the writes of } l \text{ is a subset of mo for } l \text{ *)} \\
\qquad\qquad \xrightarrow{\textit{happens-before}}|_{\textit{writes\_at\_l}} \subseteq \xrightarrow{\textit{modification-order}} \wedge \\
\qquad\qquad [\ldots]) \\
\quad \| \_ \rightarrow (\\
\qquad\quad \textbf{let}\ \textit{actions\_at\_l} = \{a.\ (\text{location}\ a = \text{SOME}\ l)\}\ \textbf{in} \\
\qquad\quad (\xrightarrow{\textit{modification-order}}|_{\textit{actions\_at\_l}}) = \{\}))
\end{array}
$$

In the example, the release action (b) has a release sequence [(b),(c)], a contiguous sub-sequence of modification order on the location of the write-release. The release sequence is headed by the release and can be followed by writes from the same thread or read-modify-writes from any thread; other writes by other threads break the sequence. We represent a release sequence not by the list of actions but by a relation from the head to all the elements, as the order is given by modification order. In figures we usually suppress the reflexive edge from the head to itself.

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

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

A write-release *synchronizes-with* a read-acquire if both act on the same location and the release sequence of the release contains the write that the acquire reads from. In the example $b \xrightarrow{\textit{release-sequence}} c \xrightarrow{\textit{rf}} d$, so we have $b \xrightarrow{\textit{synchronizes-with}} d$. The definition below covers mutexes and thread creation (in additional-synchronized-with) but elides the effects of fences.

$$
\begin{array}{l}
a \xrightarrow{\textit{synchronizes-with}} b = \\
\quad \text{(* – additional synchronization, from thread create etc. – *)} \\
\quad a \xrightarrow{\textit{additional-synchronized-with}} b \vee \\
\\
\quad (\text{same\_location}\ a\ b \wedge a \in \textit{actions} \wedge b \in \textit{actions} \wedge (\\
\qquad \text{(* – mutex synchronization – *)} \\
\qquad (\text{is\_unlock}\ a \wedge \text{is\_lock}\ b \wedge a \xrightarrow{sc} b) \vee \\
\\
\qquad \text{(* – release/acquire synchronization – *)} \\
\qquad (\text{is\_release}\ a \wedge \text{is\_acquire}\ b \wedge \neg \text{same\_thread}\ a\ b \wedge \\
\qquad (\exists c.\ a \xrightarrow{\textit{release-sequence}} c \xrightarrow{\textit{rf}} b)) \vee \\
\qquad [\ldots]))
\end{array}
$$

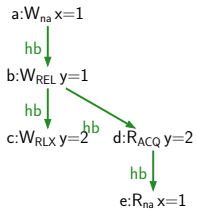
The modification order and the sc order we saw earlier must also be consistent, in the following sense:

$$
\begin{array}{l}
\text{consistent\_sc\_order} = \\
\quad \textbf{let}\ \text{sc\_happens\_before} = \xrightarrow{\textit{happens-before}}|_{\text{all\_sc\_actions}}\ \textbf{in} \\
\quad \textbf{let}\ \text{sc\_mod\_order} = \xrightarrow{\textit{modification-order}}|_{\text{all\_sc\_actions}}\ \textbf{in} \\
\quad \text{strict\_total\_order\_over}\ \text{all\_sc\_actions}\ (\xrightarrow{sc}) \wedge \\
\quad \xrightarrow{\text{sc\_happens\_before}} \subseteq \xrightarrow{sc} \wedge \\
\quad \xrightarrow{\text{sc\_mod\_order}} \subseteq \xrightarrow{sc}
\end{array}
$$

2.7 Constraining Atomic Read Values

The values that can be read by an atomic action depend on happens-before, derived from sequenced-before and synchronizes-with. We

return to the execution fragment from the previous section, showing a transitive reduction of happens-before that coincides with its constituent orderings.



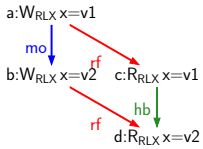
../examples/t8b.c

An atomic action must read a write that is in one of its *visible sequences of side effects*, in this case (d) either reads (b) or (c). A visible sequence of side effects of a read is a contiguous subsequence of modification order, headed by a visible side effect of the read, where the read does not happen before any member of the sequence. We represent a visible sequence of side effects not as a list but as a set of actions in the tail of the sequence (we are not concerned with their order).

$$\begin{aligned}&\text{visible\_sequence\_of\_side\_effects\_tail } \mathit{vsse\_head}\ b = \\ &\quad \{c.\ \mathit{vsse\_head} \xrightarrow{\textit{modification-order}} c \wedge \\ &\qquad \neg(b \xrightarrow{\textit{happens-before}} c) \wedge \\ &\qquad (\forall a.\ \mathit{vsse\_head} \xrightarrow{\textit{modification-order}} a \xrightarrow{\textit{modification-order}} c \\ &\qquad\quad \implies \neg(b \xrightarrow{\textit{happens-before}} a))\}\end{aligned}$$

We define *visible-sequences-of-side-effects* to be the binary relation relating atomic reads to their visible-side-effect sets (now including the visible side effects themselves). The atomic read must read from a write in one of these sets.

Atomic read actions also obey the following coherence condition: If one read happens-before another on the same location, then the writes that they read are either the same, or the write to the earlier read is modification-ordered before the later one. We show this below in the relaxed case, but it applies to all actions to atomic locations.



../examples/t39-CoRR.exc

We can now extend the previous definition of the consistent reads-from predicate to be the conjunction of the read-restrictions on nonatomic and atomic actions, the coherence condition, and a constraint ensuring read-modify-write atomicity. In the definition below, we write $a \xrightarrow{R}_P b$ to mean that $a \xrightarrow{R} b$, predicate P holds for a and there is no R-intervening z satisfying P.

$$\begin{aligned}&\text{consistent\_reads\_from\_mapping} = \\ &\quad (\forall b.\ (\text{is\_read}\ b \wedge \text{is\_at\_non\_atomic\_location}\ b) \implies \\ &\qquad (\textbf{if}\ (\exists a_{vse}.\ a_{vse} \xrightarrow{\textit{visible-side-effect}} b) \\ &\qquad\ \textbf{then}\ (\exists a_{vse}.\ a_{vse} \xrightarrow{\textit{visible-side-effect}} b \wedge a_{vse} \xrightarrow{rf} b) \\ &\qquad\ \textbf{else}\ \neg(\exists a.\ a \xrightarrow{rf} b))) \wedge \\ &\quad (\forall b.\ (\text{is\_read}\ b \wedge \text{is\_at\_atomic\_location}\ b) \implies \\ &\qquad (\textbf{if}\ (\exists(b', \mathit{vsse}) \in \textit{visible-sequences-of-side-effects}.\ (b' = b)) \\ &\qquad\ \textbf{then}\ (\exists(b', \mathit{vsse}) \in \textit{visible-sequences-of-side-effects}. \\ &\qquad\qquad (b' = b) \wedge (\exists c \in \mathit{vsse}.\ c \xrightarrow{rf} b)) \\ &\qquad\ \textbf{else}\ \neg(\exists a.\ a \xrightarrow{rf} b))) \wedge\end{aligned}$$

$$\begin{aligned}&(\forall(x, a) \in \xrightarrow{rf}. \\ &\quad \forall(y, b) \in \xrightarrow{rf}. \\ &\qquad a \xrightarrow{\textit{happens-before}} b \wedge \\ &\qquad\quad \text{same\_location}\ a\ b \wedge \text{is\_at\_atomic\_location}\ b \\ &\qquad\qquad \implies (x = y) \vee x \xrightarrow{\textit{modification-order}} y) \wedge \\ &(\forall(a, b) \in \xrightarrow{rf}.\ \text{is\_atomic\_rmw}\ b \\ &\quad \implies a \xrightarrow{\textit{modification-order}} b) \wedge \\ &(\forall(a, b) \in \xrightarrow{rf}.\ \text{is\_seq\_cst}\ b \\ &\quad \implies \neg\,\text{is\_seq\_cst}\ a \vee \\ &\qquad a \xrightarrow{sc}_{\lambda c.\ \text{is\_write}\ c \wedge \text{same\_location}\ b\ c} b) \wedge \\ &[\ldots]\end{aligned}$$

2.8 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.

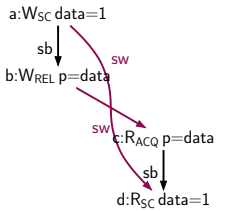
Read-consume enables efficient implementations of algorithms that use pointer reassignment for commits of their data, e.g. read-copy-update [MW]. For example, suppose one thread writes some data (perhaps spanning multiple words) then writes the address of that data to a shared atomic pointer, while the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

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

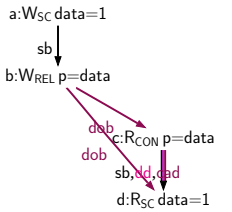
Here there is a dependency at the receiver from the read of `p` to the read of `data`. This can be expressed using a write-release and an atomic load of `p` annotated MO_CONSUME:

```
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 §2.6, the semantics of release/acquire pairs introduced 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 $a \xrightarrow{\textit{happens-before}} d$.



../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 on the right. This is data given by the operational semantics. The (dd) edge gives rise to a *carries-a-dependency-to* (cad) edge, which extends data dependency with thread-local reads-from relationships:

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

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)]):

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

2.9 Happens-before

Finally, we can define the complete happens-before relation. To accommodate MO_CONSUME, and specifically the fact that release/consume pairs only introduce happens-before relations to dependency-successors of the consume, *not* to all actions that are sequenced-after it, the definition is in two steps. First, we define *inter-thread-happens-before*, which combines synchronizes-with and dependency-ordered-before, allowing transitivity with sequenced-before on the left for both and on the right only for synchronizes-with:

$$\begin{array}{l} \xrightarrow{\textit{inter-thread-happens-before}} = \\ \quad \textbf{let}\ \ r = \xrightarrow{\textit{synchronizes-with}} \cup \\ \qquad\qquad \xrightarrow{\textit{dependency-ordered-before}} \cup \\ \qquad\qquad (\xrightarrow{\textit{synchronizes-with}} \circ \xrightarrow{\textit{sequenced-before}})\ \textbf{in} \\ (\xrightarrow{r} \cup\ (\xrightarrow{\textit{sequenced-before}} \circ \xrightarrow{r}))^+ \end{array}$$

In any execution, this must be acyclic:

$$\begin{array}{l} \mathrm{consistent\_inter\_thread\_happens\_before} = \\ \quad \mathrm{irreflexive}\ (\xrightarrow{\textit{inter-thread-happens-before}}) \end{array}$$

Happens-before (which is thereby also acyclic) is then just the union with sequenced-before:

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

2.10 Putting it together

Given a candidate execution $X = (X_{\mathrm{opsem}}, X_{\mathrm{witness}})$, we can now calculate the derived relations:

release-sequence (§2.6), *hypothetical-release-sequence* (a variant of *release-sequence* used in the fence semantics), *synchronizes-with* (§2.2, §2.6), *carries-a-dependency-to* (§2.8), *dependency-ordered-before* (§2.8), *inter-thread-happens-before* (§2.8), *happens-before* (§2.1, §2.2, §2.3, §2.8), *visible-side-effect* (§2.1), and *visible-sequences-of-side-effects* (§2.7).

The definition of consistent_execution used at the start of Section 2 is then simply the conjunction of the predicates we have defined:

consistent_execution =	
well_formed_threads ∧	(§2.5, defn. elided)
consistent_locks ∧	(§2.2, defn. elided)
consistent_inter_thread_happens_before ∧	(§2.8)
consistent_sc_order ∧	(§2.6)
consistent_modification_order ∧	(§2.6)
well_formed_reads_from_mapping ∧	(§2.1, defn. elided)
consistent_reads_from_mapping	(§2.1, §2.7)

The acyclicity check on inter-thread-happens-before, and the subtlety of the non-transitive happens-before relation, are needed only for release/consume pairs:

Theorem 1. *For an execution with no consume operations, the* consistent_inter_thread_happens_before *condition of* consistent_execution *is redundant.*

Theorem 2. *If a consistent execution has no consume operations, happens-before is transitive.*

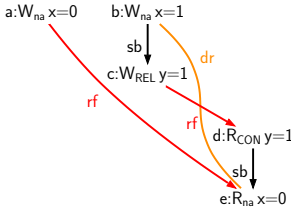
The proofs are by case analysis and induction on the size of possible cycles.

3. Examples

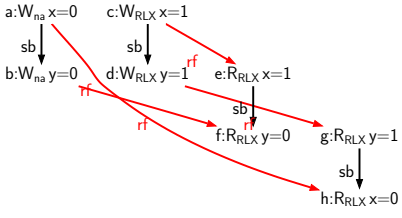
We now illustrate the varying strength of the different memory orders by showing the semantics of some 'classic' examples. In all cases, variants of the examples with SC atomics do not have the weak-memory behaviour. As in our other diagrams, to avoid clutter we only show selected edges, and we omit the C++ sources for these examples, which are available on-line [BOS].

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.

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. As we saw in §2.6, with a release/acquire pair it is. A release/consume pair gives the same guarantee iff there is a dependency between the reads, otherwise there is a consistent execution (on the left) 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).



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

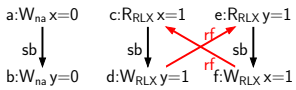


../examples/irdw-relaxed.c

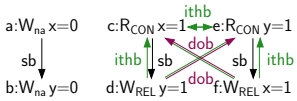
The same holds with relaxed flag operations.

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 on the right above) — consistent with what one might see on Power, for example.

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 on the left:



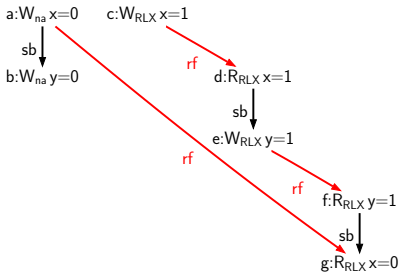
../examples/t30-lb-relaxed.c



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

but with release/consumes (with dependencies) it is not (as on the right above), 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), because of the cyclic inter-thread-happens-before and for other reasons.

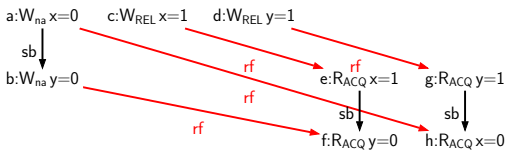
Write-to-Read Causality (WRC) Here the first (non-initialisation) thread writes to x; the second reads from that and then (w.r.t. sequenced-before) writes to y; the third reads from that and then (w.r.t. sequenced-before) reads x. The question is whether it is guaranteed to see the first thread's write.



../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 second two threads can see those writes in different orders. With relaxed, release/acquire, or release/consume atomics, they can.



../examples/t9.c

4. From standard to formalisation and back

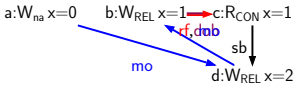
We developed the model presented in Section 2 by a lengthy iterative process: building formalisations of various drafts of the standard, and of Boehm and Adve's model without low-level atomics [BA08]; considering the behaviour of examples, both by hand and with our tool; trying to prove properties of 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 current draft (the final committee draft, N3090) of the standard, we describe a selection of debatable issues. This also serves to bring out the delicacy of the standard, and the pitfalls of prose specification, even when carried out with great care. We have made suggestions for technical or editorial changes to the draft for many of these points and it seems likely that several of them will be incorporated.

Acyclicity of happens-before N3090 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 §3 has a cyclic inter-thread-happens-before, as shown there, but is otherwise a consistent execution. 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 There are 6 places where N3090 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 instead add them to the synchronized-with relation; for those within our C++ fragment, our operational semantics introduces them into additional-synchronized-with.

'Subsequent' in visible sequences of side effects N3090 defines: *The visible sequence of side effects on an atomic object M, with respect to a value computation B of M, is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first side effect is visible with respect to B, and for every subsequent side effect, it is not the case that B happens before it.* However, if every element in a vsse happens-before a read, the read should not take the value of the visible side effect. Following discussion, we formalise this without the *subsequent*.

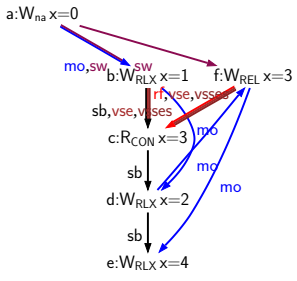
Release/consume example The draft standard requires modification-order to agree with happens-before, but as happens-before is not always transitive the standard (and our model) permits counter-intuitive executions such as that below.



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

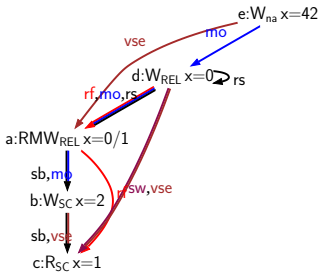
Here there is a dependency-ordered-before edge for a write-release/read-consume pair (a),(b), with (b) reading from (a), but the write (a) *follows* (in modification order) another write (c) that *follows* the read in sequenced-before. We believe that a natural Power implementation of C++ would not permit this behaviour, which can be seen as arising from the lack of a coherence condition in C++0x.

Non-unique vsses The draft standard refers to *'the'* visible sequence of side-effects, suggesting uniqueness. Nevertheless, the following valid execution has more than one, relying on the lack of transitivity of happens-before as in the previous example. This is not necessarily a problem, but may be confusing.



../examples/t24-4m2vsses.c

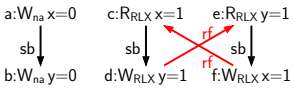
Reading a hidden write In the example below, the read-acquire (c) synchronizes-with the write-release (d) by virtue of reading from the RMW (a), which is in the release sequence headed by (d). Counter-intuitively, this can happen despite the presence of an intervening write (b) in sequenced-before. This relies on the inclusion of read-modify-writes from any thread in the definition of release-sequence, but seems to arise from the lack of another coherence condition.



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

Overlapping executions and thin-air reads In a C++0x program that gives rise to the relaxed LB example in §3, the written value 1 might have been concrete in the program source. Alternatively, one might imagine a *thin-air read*: the program below has the same execution, and here there is *no* occurrence of 1 in the program source.

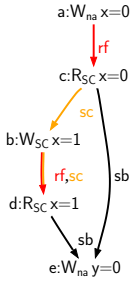
```
int main() {
  int r1, r2;
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { r1 = x.load(mo_relaxed));
        y.store(r1,mo_relaxed); }
  ||| { r2 = y.load(mo_relaxed));
        x.store(r2,mo_relaxed); }
  }}}
  return 0; }
```



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

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. C++0x also 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

5. 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. Events 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

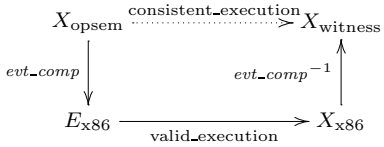
reads and writes that is total on the writes. The remainder of the axiomatisations are very different: x86-TSO has no concept of release, acquire, visible side effect, etc.

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.

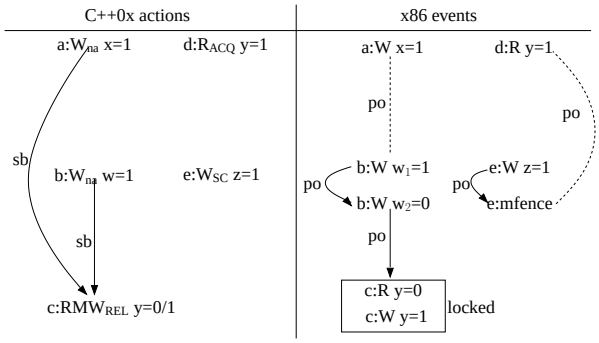
First, we lift the mapping between instructions to a nondeterministic translation $action\_comp$ from C++ actions to small x86 event structures, e.g. relating an atomic read-modify-write action to the events of the corresponding x86 LOCK'd instruction.

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.



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 3. *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:*

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

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 $\exists(e_1 \in comp\ a)(e_2 \in comp\ b).\ e_1 \xrightarrow{\text{x86-rf}} e_2$). 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 $a \xrightarrow{\text{happens-before}} b$ 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 $\text{consistent\_execution}$. 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 $a \xrightarrow{\text{visible-side-effect}} b$, we note that if there were a write c to the same location such that $a \xrightarrow{\text{happens-before}} c \xrightarrow{\text{happens-before}} b$, 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 $a \xrightarrow{\text{visible-side-effect}} b$, we need to show that $a \xrightarrow{\text{happens-before}} b$. 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

[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.

creates a data race, and allow the program to complete in any way. The details of this part are by pencil-and-paper proof.

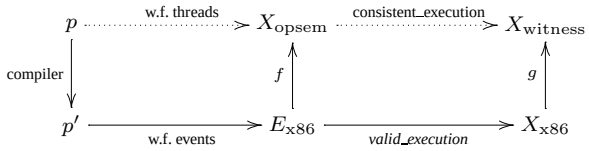
□

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 3. 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 $a \xrightarrow{rf} b$, then a immediately precedes b in *sc* restricted to that address.

Compiler correctness Although we translate executions instead of source code, Theorem 3 could be applied to full source-to-assembly compilers that follow the prototype implementation. The following diagram presents the overall correctness property.



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 3 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.

6. Tool support for exploring the model

Given a a relatively complex axiomatic memory model, as we presented in Section 2, 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.

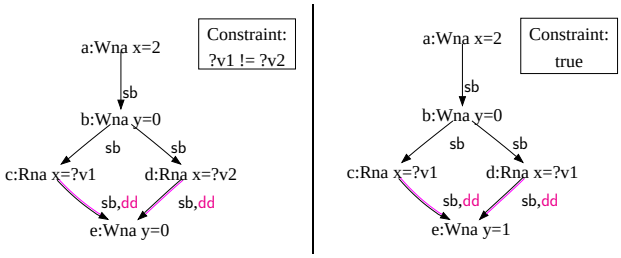
The tool has three main components: an executable symbolic operational semantics to build the X_{opsem} parts of the candidate executions X of a program; a search procedure to enumerate the possible X_{witness} for each of those; and a checking procedure to calculate the derived relations and predicates 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 §2.1. 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.

6.1 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.

7. Related work

The starting points for this paper were the draft standard itself and the work of Boehm and Adve [BA08], who 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.

8. Conclusion

We have put the semantics of C++ and C concurrency on a mathematically sound footing, following the current final committee draft standard as far as possible, except as we describe in §4. This should support future improvements to the standard and the development of semantics, analysis, and reasoning tools for concurrent systems code.

Having done so, the obvious question is the extent to which the formal model could be incorporated as a *normative* part of a future standard. The memory model is subtle but it uses only simple mathematical machinery, of various binary relations over a fixed set of concrete actions, that can be visualised graphically. There is a notational problem: one would probably have to translate (automatically or by hand) the syntax of first-order logic into natural language, to make it sufficiently widely accessible. But given that, we suspect that the formal model would be clearer than the current 'standardsese' for all purposes, not only for semantics and analysis.

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, Doug Lea, Nick Maclaren, Paul McKenney, Clark Nelson, and Anthony Williams. 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. *Working Draft, Standard for Programming Language C++*. March 2010. N3090=10-0080.

[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.