Mathematizing C++ Concurrency

Mark Batty  Scott Owens  Susmit Sarkar  Peter Sewell  Tjark Weber
University of Cambridge

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 not yet clear and rigorous definitions, and free substantial problems in their details.

In this paper we establish a mathematical (yet readable) semantics for C++ concurrency. We aim to capture the intent of the current (‘Final Committee’) Draft as closely as possible, but discuss changes that fix many of its problems. We prove that a proposed x86 implementation of the concurrency primitives is correct with respect to the x86-TO model, and describe our CPMEM tool for exploring the semantics of examples, using code generated from our Isabelle/HOL definitions.

Having already motivated changes to the draft standard, this work will aid discussion of any further changes, 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 processing: 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 [In02]. 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 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 explain 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, several substantial omissions, 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, SA08, 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 [Ils], 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 atoms, the various flavours of low-level atomic, 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 suggestions for changes. These are of various kinds, including editorial clarifications, substantive changes, 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 at the time of writing 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 prototype 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 [BN11]) to find executions more efficiently, albeit with less assurance. All 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 check-point, 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 atoms, and low-level atoms (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 (opsem, 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:

\[
\text{cpp} \text{memory model opsem \ p \ \ := \ \ let pre_executions \ = \ \{(X_{opsem}, X_{witness})\}, \ \ \text{opsem} \ p \ X_{opsem} \ \land \ \ \text{consistent execution} \ X_{opsem}, X_{witness} \ \text{in} \ \ \begin{cases} 3X \in \ pre_executions, \ (\text{indeterminate reads } X \neq \{\}) \lor (\text{unsequenced races } X \neq \{\}) \lor (\text{data races } X \neq \{\}) \to \text{NONE} \ \text{if} \ \text{else \ SOME \ pre_executions} \end{cases} \]

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. 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_{mn}) or writes (W_{mn}), 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.

\[
\begin{align*}
\text{int main() } \{ \\
\text{ int } x = 2; \\
\text{ int } y = 0; \\
\text{ y } = (x == x); \\
\text{ return } 0; \\
\}
\end{align*}
\]

The set of actions and the sequenced-before relation are given by the operational semantics (so are part of the \(X_{\text{opsem}}\)); the rf relation is existentially quantified (part of the \(X_{\text{witness}}\), 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.

\[
as \text{ visible-side-effect } b = \begin{cases} \text{ happens-before } a \rightarrow b \land & \text{is_write } a \land \text{is_read } b \land \text{same_location } a \land b \land \neg (\exists c, (c \neq a) \land (c \neq b)) \land \text{is_write } c \land \text{same_location } c \land b \land (\exists a, a \rightarrow c) \land (b \land a \rightarrow c) \land (\exists a, a \rightarrow c) \land (b \land a \rightarrow c) \end{cases}
\]

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.

\[
\text{consistent} \rightarrow \text{from_mapping} = (\forall b, (\text{is_read } b \land \text{is_at_nonatomic_location } b) \implies \left( (\exists a \rightarrow c, a \rightarrow c \text{ visible-side-effect } b) \land \left( (a \rightarrow c, a \rightarrow c \text{ happens-before } b \land a \land c \rightarrow b) \land \left( \vdots \right) \right) \right) \land (\exists a, a \rightarrow b) \right) \land (b \land a \rightarrow c) \land (\exists a, a \rightarrow c) \land (b \land a \rightarrow c) \end{cases}
\]

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.

\[
\text{indeterminate} \rightarrow \text{reads} = \{ b, \text{ is_read } b \land \neg (\exists a, a \rightarrow b) \}
\]

A pre-execution has an unsequenced-race if there is a write and another access to the same location, on which the two actions are unsequenced.

\[
\text{unsequenced} \rightarrow \text{races} = \{ (a, b), \}
\]

A pre-execution has a data race (dr) if there are two actions at the same location, on different threads, not related by happens-before, at least one of which is a write.

\[
\text{data} \rightarrow \text{races} = \{ (a, b), \}
\]

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 \(X_{\text{witness}}\) 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. 

\[
\begin{align*}
\text{synchronizes-with} & : a \to b \Rightarrow a \to b \Rightarrow \\
\text{additional-synchronizes-with} & : a \to b \Rightarrow a \to b \lor
\end{align*}
\]

\( (*) \) \text{ additional synchronisation, from thread create etc. \( \sim *) \) 

\( (**) \text{ mutex synchronization} \sim ** \) 

\( \text{is_unlock a} \land \text{is_lock b} \land a \rightarrow b \lor \ldots ) \]

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 \textit{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.

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

Semantically, this is because SC atomic operations are totally ordered by \( sc \), 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 \( sc \) is a total order over all actions that \( \sim \) and values \( v \)). As we have seen, 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 an \( mo \) parameter). We often elide the thread ids.

### 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)

\[
\begin{align*}
\text{aid} & : \text{non-atomic read} \\
\text{aid} & : \text{non-atomic write} \\
\text{aid} & : \text{atomic read} \\
\text{aid} & : \text{atomic write} \\
\text{aid} & : \text{atomic read-modify-write} \\
\text{aid} & : \text{lock} \\
\text{aid} & : \text{fence}
\end{align*}
\]

The \( is_{\text{read}} \) predicate picks out non-atomic and atomic reads and atomic read-modify-writes; the \( is_{\text{write}} \) 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 \textit{location-kind} map. The atomic actions can only be performed on \textit{Atomic} locations.

### 2.6 Release/Acquire Synchronization

SC atomic actions are expensive to implement on most multiprocessors, e.g. with the suggested implementations for an SC atomic load being \textit{LOCK} XADD(0) on x86 [Ter08] and \textit{hasync}; id; cmp; bc; iasync on Power [MS10]; the \textit{LOCK}d instruction and the \textit{hasync} 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 \textit{memory order}, \( mo \), that specifies how much synchronisation and ordering is required. The strongest ordering is required for \textit{MO_SEQ_CST} actions (which is the default, as used above), and the weakest for \textit{MO_RELAXED} actions. In between there are \textit{MO_RELEASE/MO_ACQUIRE} and \textit{MO_RELEASE/MO_CONSUME} pairs, and \textit{MO_ACQ_REL} with both acquire and release semantics.

### 2.7 Low-level Atomics

Atomic actions (which is the default, as used above), and the weakest for \textit{MO_RELAXED} actions. In between there are \textit{MO_RELEASE/MO_ACQUIRE} and \textit{MO_RELEASE/MO_CONSUME} pairs, and \textit{MO_ACQ_REL} with both acquire and release semantics.

An atomic write or fence is a \textit{release} if it has the memory order \textit{MO_RELEASE}, \textit{MO_ACQ_REL} or \textit{MO_SEQ_CST}. Atomic reads or fences with order \textit{MO_ACQUIRE}, \textit{MO_ACQ_REL} or \textit{MO_SEQ_CST}, and fences with order \textit{MO_CONSUME}, are \textit{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
x = ... // while (0 == y);
y = 1;

// receiver
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 \texttt{MO\_RELEASE}, and an atomic load of \( y \) annotated \texttt{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 \texttt{synchronizes-with} edge, e.g. as on the left below (where the \( rf \) edges are suppressed).

For such programs (in fact for any program without release/consume atomic), happens-before is still the transitive closure of the union of the sequenced-before and synchronizes-with relations, so here \( a \) 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 \texttt{release sequence}, of writes that follow (in some sense) a write-acquire.

For example, in the fragment of an execution on the right above, 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.

The modification order of a candidate execution (here \( b \rightarrow c \rightarrow d \rightarrow e \) modification-order) 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.

\[
\text{consistent\_modification\_order} = \left( \forall l \in \text{locations\_of\_actions}, \forall a, b \in \text{actions} \mid \text{case location-kind } l \mid \text{atomic} \rightarrow \left( \begin{array}{l}
\text{let } \text{actions\_at}\_l = \{ a \mid \text{is\_domain}\_store\_a \lor \text{is\_atomic\_load}\_a \} \\
\text{strict\_total\_order \over \text{actions\_at}\_l} \\
\text{(*) happens-before at the writes of } l \text{ is a subset of mo for } l \} \right) \right)
\]

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.

\[
\text{rs\_element } rs\_head \ a = \begin{cases} 
\text{same\_thread } a \ rs\_head \lor \text{is\_atomic\_rmw } a \\
\text{is\_atomic\_location } b \land \\
\text{is\_release } a \land (b = a) \\
\text{modification-order } b \land \\
\text{(rs\_element } a \land \text{modification-order } b \land \\
\text{rs\_element } a \ c))
\end{cases}
\]

A write-release \texttt{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 \( b \rightarrow \text{release-sequence} \rightarrow d \), so we have \( b \overset{rf}{\rightarrow} d \). The definition below covers mutexes and thread creation (in additional-synchronizes-with) but elides the effects of fences.

\[
a \text{ synchronizes-with } b = \begin{cases} 
\text{(*) additional synchronization, from thread create etc. } & \text{–*} \\
\text{additional-synchronizes-with } & b \lor \\
\text{(same\_location } a \land a \in \text{actions} \land b \in \text{actions} \land ( & b \land \text{mutex synchronization } & \text{–*}) \\
\text{is\_unlock } a \land a \land a \text{ by } b \lor \\
\text{(release/acquire synchronization } & \text{–*}) \\
\text{is\_release } a \land a \text{ by } b \land \text{same\_thread } a \land b \land ( & \text{release-sequence } c \overset{rf}{\rightarrow} b) \lor \\
\text{(\exists c. a} & \text{release-sequence } c \overset{rf}{\rightarrow} b))
\end{cases}
\]

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

\[
\text{consistent\_sc\_order} = \begin{cases} 
\text{let } \text{happens\_before} = \text{happens\_before} \mid \text{all\_sc\_actions} \in \text{in} \\
\text{let } \text{sc\_mod\_order} = \text{modification\_order} \mid \text{all\_sc\_actions} \in \text{in} \\
\text{strict\_total\_order} \over \text{all\_sc\_actions} \mid \text{in} \\
\text{happens\_before} \subseteq \text{in} \land \\
\text{sc\_mod\_order} \subseteq \text{in} 
\end{cases}
\]

### 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 shown on the right in the previous subsection, showing a transitive reduction of happens-before that coincides with its constituent orderings.

An atomic action must read a write that is in one of its \textit{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).

\[
\text{visible_sequence_of_side_effects}_\text{tail} = \{ \text{c, vse_head} \mid \text{modification-order} \rightarrow \text{c} \}
\]

\[
\neg(b \text{ happens-before} \text{ c}) \land (\forall a. \text{vse_head} \text{ modification-order} \rightarrow a \text{ happens-before} \text{ c})
\]

We define \( \text{visible-sequences-of-side-effects} \) 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.

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, and a constraint ensuring read-modify-write atomicity.

\[
\text{consistent_reads_from_mapping} = \left( \forall b. (\text{is_read} b \land \text{is_at_non_atomic_location} b) \implies \left( (\exists a_{vse} \land a_{vse} \text{ visible-side-effect} b) \implies \right. \right. \left. \right.
\]

\[
\left. \left. \left. \left. \left( \neg(\exists b_{vse} \land b_{vse} \text{ visible-side-effect} a) \lor \neg(\exists c_{vse} \land c_{vse} \text{ modification-order} b) \right) \lor \right. \right. \right. \left. \right.
\]

\[
\left. \left. \neg(\exists a_{vse} \land a_{vse} \text{ visible-side-effect} b) \land \right) \lor \right)
\]

\[
(\forall a, b \in d_r, \text{atomic_rmw} b \implies a \text{ modification-order} b) \land
\]

A candidate execution is also required to be free of the following four execution fragments. This property is called coherence.

CoRR Two reads ordered by happens-before may not read two writes that are modification-ordered in the other direction.

CoWR It is forbidden to read from a write that is happens-before-hidden by a later write in modification order.

CoWW Happens-before and modification-order may not disagree.

CoRW The union of the reads-from map, happens-before and modification-order must be acyclic.

Finally, we restrict SC reads. If there is no preceding write in sc order, then there is no extra restriction. Otherwise, they must read from the last prior write in sc order, from a non-atomic write that follows it in modification order, or from any non-SC atomic write.

2.8 Release/Consume Atomics

On multiprocessors with weak memory orders, notably Power, release/consume pairs are cheaper to implement than sequentially consistent atoms but still significantly more expensive than plain stores and loads. For example, the proposed Power implementation of load-acquire, ld; cmp; be; iasync, involves an iasync [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 iasync sequence unnecessary. As we understand it, this is the motivation for introducing a read-consume variant of read-acquire atomicity. 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
data = ... // receiver
data = p; p = &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 \( \text{MO}_{\text{CONSUME}} \):

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

As we saw in §2.6, the semantics of release/acquire pairs introduced synchronizes-with edges, and happens-before includes the transitive closure of synchronizes-with and sequenced-before — for a release/acquire version of this example, we would have the edges on the left below, and hence \( \text{a happens-before} \rightarrow d. \)

For release/consume, the key fact is that there is a data dependency (dd) from (c) to (d), as shown on the right. The (dd) edge is provided by the operational semantics and gives rise to a carries-a-dependency-to (cad) edge, which extends data dependency with thread-local reads from relationships:

\[
a \rightarrow \text{carries-a-dependency-to} \rightarrow b \implies a \rightarrow \text{sequence-before} \lor \text{data-dependency} + b
\]

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\}).
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:

\[
\text{inter-thread-happens-before} = \text{synchronizes-with} \cup \text{dependency-ordered-before} \cup (\text{synchronizes-with} \cup \text{sequenced-before})^{+}
\]

In any execution, this must be acyclic:
\[
\text{consistent}\_\text{inter-thread-happens-before} = \text{irreflexive}(\text{inter-thread-happens-before})
\]

Happens-before (which is thereby also acyclic) is then just the union with sequenced-before:
\[
\text{happens-before} = \text{sequenced-before} \cup \text{inter-thread-happens-before}
\]

2.10 Putting it together

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

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

The definition of consistent\textunderscore execution used at the start of Section 2 is then simply the conjunction of the predicates we have defined:

\[
\text{consistent}\_\text{execution} = \text{well_formed_threads} \land \text{consistent}\_\text{locks} \land (\text{§2.5, defn. elided}) \land \text{consistent}\_\text{inter-thread-happens-before} \land (\text{§2.2, defn. elided}) \land \text{consistent}\_\text{ac-order} \land (\text{§2.6}) \land \text{consistent}\_\text{modification-order} \land (\text{§2.6}) \land \text{well_formed}\_\text{reads_from}\_\text{mapping} \land (\text{§2.1, defn. elided}) \land \text{consistent}\_\text{reads_from}\_\text{mapping} \land (\text{§2.1, §2.7})
\]

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. This behaviour is not allowed for the same program using sequentially consistent consistent atomics (with non-atomic initialisation).

**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 the does not happen-before the read and so is not a visible side effect).

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 dual 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. In C++0x this behaviour is ruled out as that does not happen-before the read and so is not a visible side effect.

**Write-to-Read Causality (WRC)** Here the first thread writes to \( x \); the second reads from that and then (w.r.t. sequenced-before) reads \( y \); 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.

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 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.

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, N3092) 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 they will be incorporated.

We begin with two straightforward drafting issues, easily fixed. Then there are three substantial semantic problems in N3092 where we have proposed solutions. Finally, there is an outstanding question that warrants further investigation.

‘Subsequent’ in visible sequences of side effects N3092 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.

Additional happens-before edges 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 instead add them to the synchronises-with relation; for those within our C++ fragment, our operational semantics introduces them into additional-synchronizes-with.

Acyclicity of happens-before 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 §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.

Coherence requirements The draft standard enforced only two of the four coherence requirements presented in §2.7, CoRR and CoWW. In the absence of CoRW and CoWR, the following executions were allowed.

The execution on the left violates CoRW by containing a cycle of happens-before and modification order edges, allowed only due to the lack of transitivity of happens-before. The execution on the right violates CoWR by having a read from a write (the read-modify-write (a)) that is sequenced-before-hidden by (c). Actions (b) and (c) are shown as SC atomics for emphasis.

Furthermore, the draft standard refers to ‘the’ visible sequence of side-effects, suggesting uniqueness. Nevertheless, it allows valid executions that have more than one, relying on the lack of transitivity of happens-before as in the CoRW execution above.

These behaviours are surprising and were not intended by the designers.

Sequential consistency for SC atomics The promise of sequential consistency to the non-expert programmer is a central design choice of C++0x and is stated directly by N3092: memory_order<_seq_cst> ensures sequential consistency [...] for a program that is free of data races and uses exclusively memory_order_seq_cst operations. Unfortunately N3092 allows the following non-sequentially consistent execution of the SB example with SC atomics (initialisation writes, such as (a) and (b), are non-atomic so that they need not be compiled with memory fences):

We devised a stronger restriction on the values that may be read by SC atoms, stated in §2.7, that does provide sequential consistency here.

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.

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

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. N3092 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 follows, 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.

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

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 x86 hardware. We also ignore sequentially consistent fences, 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 an assembly program by creating an x86 event structure $E_{\text{x86}}$ (analogous to $X_{\text{opsem}}$) comprising a set of events and 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 $X_{\text{witness}}$, there are x86 execution witnesses $X_{\text{x86}}$ which comprise 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 $\text{action}_{\text{comp}}$ 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 $X_{\text{opsem}}$ which is allowed by its operational semantics. We regard an abstract compiler $\text{evt}_{\text{comp}}$ as taking such an $X_{\text{opsem}}$ and giving an x86 event structure $E_{\text{x86}}$, respecting the $\text{action}_{\text{comp}}$ 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 $E_{\text{x86}}$ implies the existence of a consistent C++ execution witness $X_{\text{witness}}$ for the original actions $X_{\text{opsem}}$. We prove this by lifting such an x86 execution witness to a C++ consistent execution, as illustrated below.

\[ E_{\text{x86}} \xrightarrow{\text{valid}_{\text{exec}}} X_{\text{x86}} \]

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

In more detail, we use two existentially quantified helper functions $\text{locn}_{\text{comp}}$ and $\text{tid}_{\text{comp}}$ 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, $\text{locn}_{\text{comp}}$ 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 $\text{locn}_{\text{comp}}$ 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 \texttt{evt\_comp} relation specifies valid translations, ap-
plying \textit{action\_comp} with a well-formed \textit{locn\_comp} and also con-
straining how events from different actions relate: no single x86
instruction instance can be used by multiple C++ actions, and the
x86 \textit{program-order} relation must respect C++'s \textit{sequenced-before}.

The detailed definitions, and the proof of the following theorem,
are available online \cite{BOS}.

\textbf{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 \( \text{actions} \), \textit{sequenced-before}, and \textit{location-kind} are
members of the \( X_{\text{opsem}} \) part of a candidate execution resulting from
the operational semantics of \( p \), then the following holds:

\[ \forall \text{comp locn} \quad \text{X}_{\text{x86}}\cdot \text{evt\_comp locn\_comp tdi\_comp X}_{\text{x86}}\cdot \text{valid\_execution} (\cup e \in \text{actions}(\text{comp} \ a)) \quad \text{X}_{\text{x86}} \Rightarrow \\
\exists \text{witness}, \ \text{consistent\_execution} (\text{X}_{\text{opsem}}, \text{X}_{\text{witness}}) \]

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

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

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 \( X_{\text{witness}} \) is a 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 \( \exists \text{happens-before} \ b \), then a \textit{visible-side-effect} b, we
note that if there were a write c to the same location such that
\( \exists \text{happens-before} \ c \), then the property from 1,
there is an x86 write event in \textit{comp} c that would come between the
events of \textit{comp} a and \textit{comp} b in \( X_{\text{x86}} \), thus meaning that they
would not be in \( X_{\text{x86}} \)'s reads-from map, contradict the construc-
tion of \( X_{\text{witness}} \)'s reads-from map.

3) In some cases, some of the properties required for 2 might be
false. For example, in showing that \( \exists \text{happens-before} \ b \), we need to show that a \textit{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 \( X_{\text{x86}} \), again using techniques from our previous work \cite{Owe10}.
Next we can remove the violation, resulting in a consistent \( X_{\text{witness}} \)
for a prefix of the execution, then we add the read 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.

\end{proof}

\textbf{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. In
practice, placing the fence after the sc writes is expected to yield
higher performance.

This optimisation is a direct result of using triangular-race free-
edom (TRF) \cite{Owe10} to construct the \textit{sc} ordering in proving The-
orem 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 \( X_{\text{x86}} \) can be linearised such that each read from the
most recent preceding write.

Although the entirety of an execution witness \( X_{\text{x86}} \) might con-
tain triangular races and therefore not be linearisable, by restricting
attention to only sc reads and writes we get a subset of the execu-
tion 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 \( X_{\text{witness}} \)'s \textit{sc} ordering: if \( a \) and \( b \) are
sequentially consistent atomics and \( a \text{ \textit{happens-before} } b \), then \( a \) immediately
precedes \( b \) in \textit{sc} restricted to that address.

\textbf{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.

\[ p \quad \text{w.f. threads} \quad \text{X}_{\text{opsem}} \quad \text{consistent\_execution} \quad \text{X}_{\text{witness}} \quad \text{compiler} \quad \text{writes to x86} \quad \text{valid\_execution} \quad X_{\text{x86}} \]

If, once we use \( f \), we can then apply \textit{evt\_comp} to get the same
execution \( f(E) \), then The-
orem 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 \textit{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 relatively complex axiomatic memory model, as we pre-
sented in Section 2, it is often hard to immediately see the con-
sequences of the axioms, or what behaviour they allow for partic-
ular programs. Our \texttt{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 $X_{\text{opsem}}$ parts of the candidate executions $X$ of a program; a search procedure to enumerate the possible $X_{\text{without}}$ for each of those; and a checking procedure to calculate the derived relations and predicates of the model for each $(X_{\text{opsem}}, X_{\text{without}})$ 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 formulæ 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 selectively disable some of the checks of the model; and to de-clutter 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:

![Diagram of candidate executions]

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 $X_{\text{opsem}}$ 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 $X_{\text{opsem}}$ part of a finite candidate execution, the $X_{\text{without}}$ 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 $O\left(m^{n+1} \cdot n! \cdot (m + n)!\right)$ 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 CPSE 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 $X_{\text{opsem}}$ part of a candidate execution, the space of possible $X_{\text{without}}$ parts which will lead to valid executions can be explored by tools for model generation. We reused the operational semantics above to produce a $X_{\text{opsem}}$ 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 formulæ to first-order formulæ 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 that no execution exists, while other examples take around 5 seconds. Of course, Nitpick can also validate an execution $X$ with both parts $X_{\text{opsem}}$ and $X_{\text{without}}$ 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, MPAO5, CKS07, SA08, 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.

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.

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 [SSZN09, OSS09, SSO10, 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.

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 c++-threads mailing list, including Hans Boehm, Lawrence Crowl, Peter Dimov, Doug Lea, Nick Maclaren, Paul McKenzie, Clark Nelson, and Anthony Williams. Jasmin Blanchette assisted us with the Nitpick tool. We acknowledge funding from EPSRC grants EP/F036345, EP/H005633, EP/H027351, and EP/F067909.

References


