An Operational Semantics for C/C++11 Concurrency

Kyndylan Nienhuis Kayvan Memarian Peter Sewell
University of Cambridge
United Kingdom
first.last@cl.cam.ac.uk

Abstract

The C/C++11 concurrency model balances two goals: it is relaxed enough to be efficiently implementable and (leaving aside the "thin-air" problem) it is strong enough to give useful guarantees to programmers. It is mathematically precise and has been used in verification research and compiler testing. However, the model is expressed in an axiomatic style, as predicates on complete candidate executions. This suffices for computing the set of allowed executions of a small litmus test, but it does not directly support the incremental construction of executions of larger programs. It is also at odds with conventional operational semantics, as used implicitly in the rest of the C/C++ standards.

Our main contribution is the development of an operational model for C/C++11 concurrency. This covers all the features of the previous formalised axiomatic model, and we have a mechanised proof that the two are equivalent, in Isabelle/HOL. We also integrate this semantics with an operational semantics for sequential C (described elsewhere); the combined semantics can incrementally execute programs in a small fragment of C.

Doing this uncovered several new aspects of the C/C++11 model: we show that one cannot build an equivalent operational model that simply follows program order, sequential consistent order, or the synchronises-with order. The first negative result is forced by hardware-observable behaviour, but the latter two are not, and so might be ameliorated by changing C/C++11. More generally, we hope that this work, with its focus on incremental construction of executions, will inform the future design of new concurrency models.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]

Keywords C/C++, Concurrency

1. Introduction

C and C++ have been used for concurrent programming for decades, and concurrency became an official part of the ISO language standards in C/C++11 [8, 28, 29][1]. Batty et al. contributed to this standardisation process, resulting in a mathematical model in close correspondence with the standard prose [6].

Extensionally, the C/C++11 design is broadly satisfactory, allowing the right observable behaviour for many programs. On the one hand, the semantics is relaxed enough to allow efficient implementation on all major hardware platforms [5, 6], and on the other hand, the design provides a flexible range of synchronisation primitives, with semantics strong enough to support both sequentially consistent (SC) programming and fine-grained concurrency. It has been used in research on compiler testing, optimisation, library abstraction, program logics, and model-checking [3, 17, 19, 23, 25, 26].

Intensionally, however, the C/C+11 model (in the ISO text and the formalisation) is in an "axiomatic" style, quite different from a conventional small-step operational semantics. A conventional operational semantics builds executions *incrementally*, starting from an initial state and following the permitted transitions of a transition relation. This incremental structure broadly mirrors the way in which conventional implementations produce executions. To calculate the semantically allowed behaviours of a program, one can calculate the set of all allowed behaviours by an exhaustive search of all paths (up to some depth if necessary), and one can find single paths (for testing) by making pseudorandom choices of which transition to take from each state. The incremental structure also supports proofs by induction on paths, as in typical type preservation proofs, and dynamic analysis and model-checking tools.

In contrast, an axiomatic concurrency model defines the set of all allowed behaviours of a program in a quite different and more global fashion: it defines a notion of *candidate execution*, the set of memory actions in a putative complete execution (together with various relations over them), and a

[1] C++14 concurrency [30] is essentially the same as C++11 concurrency. The most notable change is that the attempt to forbid "thin-air", previously known to be unsatisfactory, has been removed [31].

consistency predicate that picks out the candidate executions allowed by the concurrency model; the conjuncts of this are the axioms of the axiomatic model. Executions must also be permitted by the threadwise semantics of the program, though this is often left implicit in the relaxed-memory literature (for C/C++11, one additionally needs to check whether any consistent execution exhibits a race). With this structure, to calculate the set of all allowed behaviours of a program, in principle one first has to calculate the set of all its control-flow unfoldings, then for each of these consider all the possible choices of arbitrary values for each memory read (using the threadwise semantics to determine the resulting values of memory writes), and then consider all the possible arbitrary choices of the relations (the reads-from relation, coherence order, etc.). This gives a set of candidate executions which one can filter by the consistency predicate (and then apply a race check to each). This is viable for small litmus tests, and it is essentially what is done by the `cppmem` [6] and `herd` [1] tools. It intrinsically scales badly, however: the number of candidate executions increases rapidly with program size, and the fraction of consistent executions among them becomes vanishingly small.

The fundamental problem The fundamental difficulty with calculating behaviour in the axiomatic concurrency model is that one has to construct candidates with no knowledge of whether the choices of control-flow unfolding and memory read values are actually compatible with the concurrency model; the vast majority of them will not be.

Our approach To solve the above problem we construct an equivalent operational concurrency model, and incrementally generate executions by taking both this concurrency model and the threadwise semantics into account at each step.

First contribution: negative results Our first contribution is a negative result: we show that one cannot build an equivalent operational concurrency model for C/C++11 that simply follows program order, SC order, or the synchronises-with order (§3). The axiomatic model allows executions with certain cycles in the union of program order, the reads-from relation, coherence order, SC order and synchronises-with order (we recall these relations in §2). In a sequentially consistent semantics, each of the latter relations are consistent with program order: as one builds an execution path incrementally, each read is from a write that is earlier in the path, each write is a coherence-successor of a write that is earlier in the path, and so on. For a relaxed-memory semantics, that is not always the case, and so in order to be complete with respect to the axiomatic model the transitions of our operational semantics must be able to generate those cycles and can therefore not simply follow all the above relations.

The first negative result (one cannot build an equivalent operational model that follows program order) is not original, but the latter two (about SC order and synchronises-with order) are. Furthermore, the first negative result is forced by hardware-observable behaviour, but the latter two are not, and so might be ameliorated by changing C/C++11. The changes we propose (§3.1, §3.3 and §3.4) are original.

Main contribution: an equivalent operational concurrency model We show that the axiomatic model *does* behave incrementally under a particular execution order, we develop an operational concurrency model following that order, and prove this model equivalent to the axiomatic model of Batty et al. [6], with a mechanised Isabelle/HOL proof (§4–6). We do all this for the full C/C++11 model as formalised by Batty et al. [6], including non-atomic accesses, all the atomic memory orders (sequentially consistent, release/acquire, release/consume, and relaxed), read-modify-write operations, locks, and fences.

Our operational semantics is not in an "abstract machine" style—with an internal structure of buffers and suchlike—that has a very concrete operational intuition. That might be desirable in principle, but the C/C++11 model is an abstraction invented to be sound with respect to multiple quite different implementations, covering compiler and hardware optimisations; it is unclear whether an equivalent abstract-machine model is feasible. Instead, the operational semantics is defined using the axioms of the axiomatic model.

We are also deliberately not addressing the "thin-air" problem: the C/C++11 model permits certain executions that are widely agreed to be pathological, but which are hard to characterise [4]. Here we are aiming to be provably equivalent to that model, and those executions are therefore also permitted by our operational model. Instead we are solving an orthogonal problem: the cyclic executions presented in §3 that are the main reasons why developing an operational semantics is difficult are not out-of-thin-air executions. There may be scope for combining this work with proposals for thin-air-free models for the relaxed and non-atomic fragment of C/C++11 [21].

Third contribution: integration with a sequential semantics We integrate our operational concurrency model with a sequential operational semantics (§7). That sequential semantics, covering a substantial fragment of C, is described in detail elsewhere [16]; it is not itself a contribution of this paper.

The integration supports integers (of any kind), atomics, fences, conditional statements, loops, function calls, and parallel composition. Supporting non-scalar types such as arrays and structs is outside the scope of this work: the axiomatic concurrency model does not support them and the intention of the standard is not clear.

The integration is executable and can be used to pseudo-randomly explore single paths of programs. It is, however, not intended to be an efficient tool: the size of the state and the time to compute the next transition grow during execution. Rather, the integration solves the fundamental problem we described above: we can find out whether choices

of control-flow unfolding and memory read values are compatible with the concurrency model *during* the execution.

Mechanisation For such an intricate area, mechanisation has major advantages over hand proofs, but it also comes at a significant cost. The total development amounts to 7 305 lines of Isabelle/HOL script (excluding comments and whitespace), together with 2 676 lines of Isabelle/HOL script for the original axiomatic model. We use Lem [18] to generate the latter from its Lem source, which was previously used for HOL4 proof. In the paper we only state the most important theorems and definitions; the proofs and the rest of the theorems and definitions are available online at `http://www.cl.cam.ac.uk/~pes20/cpp_op/`.

2. The C/C++11 Axiomatic Concurrency Model

We begin by recalling the C/C++11 concurrency primitives and axiomatic model, referring to previous work [2, 6, 8] for the full details. In §2.8 we illustrate the fundamental problem with the axiomatic model that we aim to solve.

2.1 Atomics

We introduce atomics by contrasting them with normal *non-atomic* accesses using the example program below. The first thread of the program sets $data$ to a value and then signals the other thread by setting *flag* to 1; the second thread reads *flag* in a loop until it sees 1, and then uses $data$ for some other computation. The syntax {-{ T_1 ||| T_2 }-} is short for creating two threads that execute T_1 and T_2 and then joining them; it avoids the extra memory actions from library-based thread creation.

```
int main(void) {
  int data = 0;
  int flag = 0;
  int result;
  {-{ { data = 1;
        flag = 1; }
  ||| { while(flag != 1) {};
        result = data; }
  }-};
  return result;
}
```

Figure 1. The message passing program with non-atomics. This program has undefined behaviour.

One might expect that this program always returns 1, but the standard gives the program *undefined behaviour*, which means that any outcome is allowed. The reason is that the program contains a *data race*: the second thread might read *flag* while the first thread writes to it (we define data races more precisely in §2.7). The fact that a data race leads to undefined behaviour allows for more compiler optimisations. For example, the writes of the first thread can be reordered, because if there are no concurrent accesses to $data$ and *flag* then the reordering cannot be observed, and if there are concurrent accesses then the program is undefined.

```
#include <stdatomic.h>
int main(void) {
  int data = 0;
  _Atomic(int) flag = ATOMIC_VAR_INIT(0);
  int result;
  {-{ { data = 1;
        atomic_store(&flag, 1); }
  ||| { while(atomic_load(&flag) != 1) {};
        result = data; }
  }-};
  return result;
}
```

Figure 2. The message passing program with atomics. This program has defined behaviour; it will always return 1.

To remove these data races one could protect the concurrent accesses by *locks*, but that might be undesirable for performance or progress reasons. As an alternative C/C++11 introduces *atomics*, which can provide synchronisation and which can be concurrently used without creating data races. In Figure 2 we see the message passing program that uses atomics (note that the type of *flag* and its accesses changed). This program does not have a data race: the accesses to *flag* do not race because they are atomic, and the accesses to $data$ do not race because the last read of *flag* in the loop of the second thread synchronises with the write to *flag* of the first thread (we define when actions synchronise in §2.5).

Besides the atomic store and atomic load seen in the program above, there are various read-modify-write (RMW) operations, including atomic increments and compare-and-swap operations.

2.2 Memory Orders

Atomic accesses without an explicit memory order annotation are sequentially consistent (SC), which means they are guaranteed to appear in a global total order. Because their implementation on relaxed hardware requires relatively expensive synchronisation, the standard introduces the following other memory orders.

- Write-release and read-acquire atomics are cheaper than SC but weaker: they do not appear in a global total order, but they guarantee (amongst other things) that if a write-release is read from by a read-acquire, then memory accesses program-order-after the latter are guaranteed to see those program-order-before the former.
- Read-consume is a still weaker variant of read-acquire, implementable on some relaxed hardware simply using the fact that those architectures guarantee that some dependencies are preserved. The status of read-consume is in flux, as McKenney et al. describe [15]: it is diffi-

```
#include <stdatomic.h>
int main(void) {
  int data = 0;
  _Atomic(int) flag = ATOMIC_VAR_INIT(0);
  int result;
  {-{ { data = 1;
        atomic_store_explicit(&flag, 1,
            memory_order_release); }
  ||| { while(atomic_load_explicit(&flag,
                  memory_order_relaxed) != 1) {};
        atomic_thread_fence(memory_order_acquire);
        result = data; }
  }-};
  return result;
}
```

Figure 3. The message passing program using weak memory orders. The program always returns 1.

cult to implement in full generality in existing compilers (where standard optimisations may remove source-code syntactic dependencies), but the basic facility it provides is widely used, e.g. in the Linux kernel. All this notwithstanding, our operational model captures its behaviour as specified in the formal C/C++11 axiomatic concurrency model.

- Relaxed atomics are the weakest of all, guaranteeing coherence but weak enough to require no hardware fences in their implementation on common architectures [22].

The program in Figure 2 can be optimised by changing the atomic SC store of the first thread by an atomic write-release, and the atomic SC load of the second thread by an atomic read-acquire. These memory orders are strong enough to ensure that the read of *flag* that reads 1 still synchronises with the write of *flag* of the first thread. We can transform this program further: the reads of *flag* that read 0 do not need to synchronise with anything, so they could be implemented as relaxed reads. We change the memory order of the read in the loop to relaxed, but now the read that reads 1 also does not synchronise anymore. To restore that, we add an acquire fence after the loop (in §2.5 we explain why this restores synchronisation). The resulting program is given in Figure 3.

2.3 The C/C++11 Semantics

The semantics of C/C++11 is factored into a *threadwise semantics* and a *concurrency semantics*. Broadly speaking, the concurrency semantics determines whether a program contains a race and which values can be read from memory, and the threadwise semantics determines everything else. With these two parts one can compute the behaviour of a program as follows.

- First the threadwise semantics generates the *pre-executions* of the program (§2.4). Each pre-execution corresponds to a particular complete control-flow unfolding of the program and an arbitrary choice of the values read from memory.
- Then, one extends each pre-execution with all possible *execution witnesses*. A pre-execution combined with one of its execution witnesses forms a *candidate execution* (§2.5).
- The axiomatic concurrency semantics defines a *consistency predicate* that can be used to determine which of the candidate executions are consistent (§2.6).
- The axiomatic concurrency semantics also defines a *race predicate* (§2.7). If any of the consistent executions satisfies the race predicate, then the behaviour of the program is undefined. Otherwise, the behaviour is the set of consistent executions.

To see that the concurrency semantics (and not the threadwise semantics) determines which values can be read from memory, recall that in pre-executions values read from memory are arbitrary, and that these values can be actually read if and only if there exists an execution witness that makes the pre-execution consistent.

2.4 Pre-executions

A pre-execution is represented as a graph, whose nodes are *memory actions*. A node label such as `a:Wna x=0` consists of:

- `a`, the identifier of the action, unique within the pre-execution.
- `W`, the type of the action, in this case a store. Other types are loads (`R`), read-modify-writes (`RMW`), fences (`F`), locks (`L`) and unlocks (`U`).
- `na`, specifying that this action is non-atomic. For atomic actions, the *memory order* (§2.2) is specified here: sequential consistent (`sc`), release (`rel`), acquire (`acq`), acquire-release (`a/r`), consume (`con`) or relaxed (`rlx`). Locks and unlocks do not have a memory order.
- `x`, the location of the action. Fences do not have a location.
- `0`, the value written (for stores). Load actions similarly contain the value read (recall that pre-execution contains arbitrary values for the return values of loads). For read-modify-writes a pair such as `2/3` specifies that 2 has been read, and 3 has been written.

The edges between the nodes denote various relations: the sequenced-before relation sb captures program order, and the additional synchronises-with relation asw captures thread creation and termination, both from the syntactic control-flow unfolding. In Table 1 we give an overview of the acronyms used.

In Figure 4 we see a pre-execution of the message passing program where the arbitrary values α, β, γ and δ are read from memory, and where the condition of the loop is executed twice: actions f and g both correspond to the same

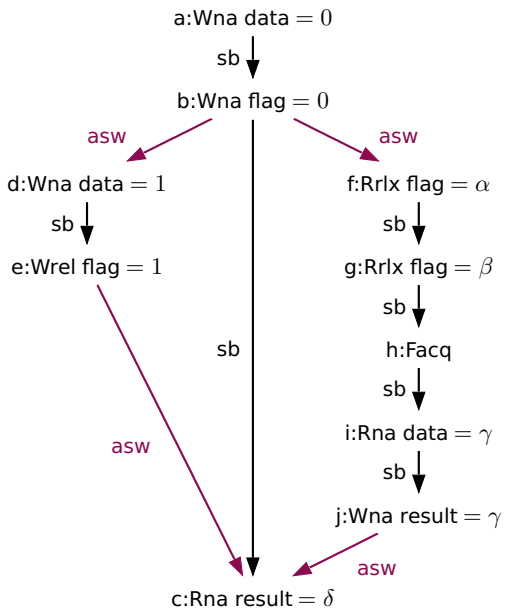

Figure 4. A pre-execution of the message passing program in Figure 3. The choice of control-flow unfolding constrains the values read: $\alpha \neq 1$ and $\beta = 1$. We omit transitive sb edges from all figures.

sb	Program order (or sequenced-before relation)
asw	Thread creation/termination (or additional sw)
rf	The reads-from relation
mo	The coherence (or modification) order
sc	The sequential consistent order
lo	The lock order
sw	The synchronises-with order
hb	The happens-before order
vse	The visible side effects order

Table 1. The acronyms used in execution graphs

instruction. The values read from memory have to agree with the choice of control flow in this pre-execution: since the condition of the loop was true the first time and false the second time, we have $\alpha \neq 1$ and $\beta = 1$; the values written to memory are determined by the threadwise semantics. The program has infinitely many other pre-executions: each time the condition of the loop is executed the value read is arbitrary, so the loop can be executed an indefinite number of times. The program also has pre-executions of infinite size where the loop is never exited.

2.5 Candidate Executions

To obtain a candidate execution from a pre-execution, we first extend it with an *execution witness*, which consists of the following relations over memory actions: the reads-from relation rf, the coherence order mo, the sequential consistent order sc, and the lock order lo. Then we derive the following relations from the pre-execution and execution witness.

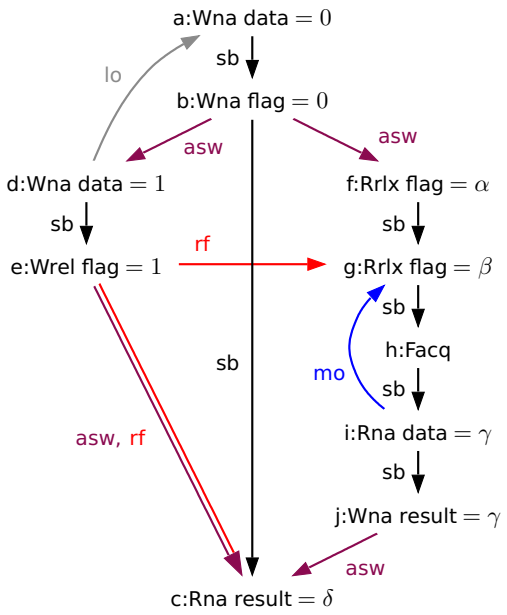

Figure 5. An (inconsistent) candidate execution that extends the pre-execution in Figure 4

- The synchronises-with relation sw. It contains thread synchronisation (the asw relation), synchronising unlock-lock pairs ($(a, b) \in lo$ with a an unlock and b a lock), synchronising release-acquire pairs ($(a, b) \in rf$ with a a write-release and b a read-acquire), and variations of the latter. These variations involve fences and *release sequences* which we introduce later.
- The happens-before relation hb. In the absence of the memory order consume, we have that $hb = (sb \cup sw)^+$ where $\cdot^+$ is the transitive closure.
- The visible side effects relation vse, with $(a, b) \in vse$ if a is a write and b a read to the same location, $(a, b) \in hb$, and there does not exists a write c to the same location that is between a and b in hb.

In principle every pre-execution can be extended by every execution witness; it is the next step (determining which candidate executions are consistent) that gives meaning to the relations defined above. To illustrate this, consider the execution in Figure 5. This is a candidate execution, despite the fact that there are writes (namely a and d) in the lock order lo.

To illustrate the derived relations, consider Figure 6. The sw edge (e, h) arises from a variant of a release-acquire pair: the relaxed read g reads from the release write e (see Figure 5), and because the acquire fence h is sb after g, we have that e and h synchronise. The other sw edges arise from

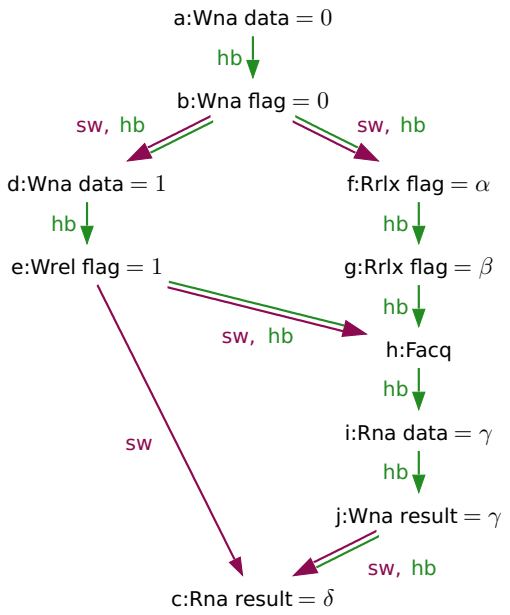

Figure 6. The relations sw and hb derived from the candidate execution in Figure 5. Transitive hb edges are not shown (for example (b, c) and (e, c)).

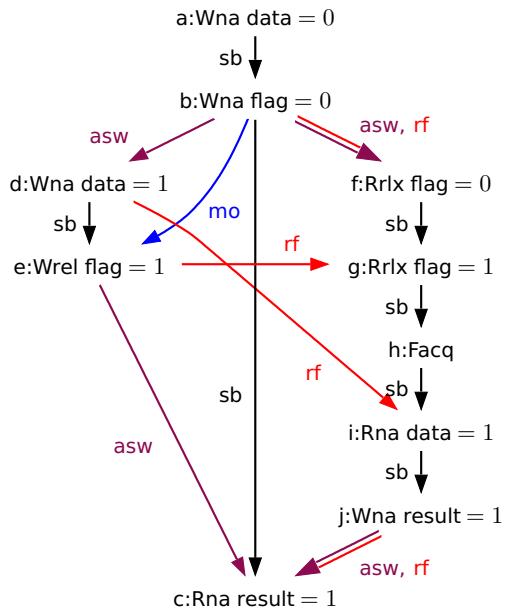

Figure 7. A consistent execution that extends the pre-execution in Figure 4 of the message passing program in Figure 3

asw edges present in the pre-execution. Since there are no consume memory orders, the hb relation is then given by $(sb \cup sw)^+$.

2.6 Consistency

In order to determine whether a candidate execution is consistent or not, the axiomatic concurrency model defines a *consistency predicate*. The consistency predicate consists of several conjuncts, which are called the axioms of the model. Some of those axioms give the relations of the execution witness their intuitive meaning:

- *well_formed_rf* requires (amongst other things) that for each $(w, r) \in rf$ we have that w and r are actions of the pre-execution, w is a write, and r a read to the same location that reads the value written by w.
- *consistent_mo* requires that mo is a total order over all atomic writes to the same location.
- *consistent_sc* requires that sc is a total order over all actions with a sequential consistent memory order.
- *consistent_lo* requires that lo is a total order over all locks and unlocks to the same location.

The other axioms define the more subtle properties that are the real substance of the C/C++11 model.

Before we introduce the other axioms, observe that the candidate execution in Figure 5 does not satisfy *consistent_lo* and is therefore inconsistent. The candidate execution in Figure 7, on the other hand, is consistent because it does satisfy all axioms. Note that there is only one choice for the arbitrary values α, β, γ and δ of the pre-execution that makes the candidate execution consistent. The whole set of consistent executions of the program consists of one execution where the loop executes n times for every n, and one infinite execution where the loop is never exited.

In the rest of the paper we refer to the following two axioms that determine whether and where non-atomic loads can read from.

- *det_read* determines whether a load r should read from somewhere: it requires that $\exists w.(w, r) \in rf$ if and only if $\exists w'.(w', r) \in vse$.
- *consistent_non_atomic_rf* requires that if a *non-atomic* load r reads from a write w we must have $(w, r) \in vse$.

The following other axioms are needed to understand why the examples in §3 are consistent and to understand the equivalence proof (which is included in the supplementary material). It is however not necessary to understand these axioms in order to understand how the operational semantics works and why it solves the fundamental problem mentioned in the introduction.

- *consistent_atomic_rf* forbids *atomic* loads to read from hb later writes.
- *rmw_atomicity* requires for each RMW r that it reads from a write w if and only if w is an immediate mo predecessor of r. Note that in consistent executions there can be at most one immediate mo predecessor, because *consistent_mo* requires mo to be a total order over atomic writes.

- *sc_reads_restricted* further restricts sequentially consistent (SC) loads and RMWs r: if r reads from an SC write w, then we must have $(w, r) \in sc$ and there cannot be a write w' to the same location that is between w and r in sc. If r reads from a non-SC write w, then there cannot be a write w' to the same location with $(w, w') \in hb$ and $(w', r) \in sc$.
- *coherent_memory_use* forbids certain coherence shapes, which force mo in a certain direction: if $(w, w') \in hb$ then $(w', w) \notin mo$; or which restrict rf: if $(w', w) \in mo$ and $(w, r) \in hb$, then $(w', r) \notin rf$; if $(r, w) \in hb$ and $(w, w') \in mo$, then $(w', r) \notin rf$; and if $(w, r) \in rf$, $(w', w) \in mo$ and $(r, r') \in hb$, then $(w', r') \notin rf$.
- *consistent_hb* requires that hb is acyclic and has finite pre-fixes.
- *locks_only_consistent_locks* requires that for every two successful locks l and l' with $(l, l') \in lo$ there exists an unlock u which is between l and l' in lo.
- *sc_fenced_sc_fences_heeded* forbids certain shapes involving SC fences, such as $(w', f) \in sb$, $(f, w) \in sc$ and $(w, w') \in mo$ with f a fence.
- *well_formed_threads* and *assumptions* require certain basic properties such as sb and asw are relations over the actions of the pre-execution, sb relates actions of the same thread and asw of different threads, all relations of the pre-execution and witness have finite pre-fixes, etcetera.

2.7 Races

Besides a consistency predicate the axiomatic concurrency model defines a *race* predicate. If one of the consistent executions contains a race according to this predicate, the whole program is undefined; otherwise the program is defined and the behaviour is the set of consistent executions. An example of a race is a *data race*: two actions, at least one a write and at least one non-atomic, that are of different threads, not happens-before related, but to the same location.

All the consistent executions of the message passing program in Figure 3 are race free. In particular, the actions d and i in Figure 7 do not race with each other because e and h synchronise (for the same reason why the actions e and h in Figure 6 synchronise). Without the acquire fence this would not be true, and the program would be undefined.

2.8 The Fundamental Problem

Having introduced the axiomatic model we can now illustrate the fundamental problem we mentioned in the introduction. The program in Figure 8 uses message passing to send the value of $data$ to the other thread. Suppose we are generating a pre-execution of this program where the first thread wrote x to $data$ and the second thread reads α from $data$. With the axiomatic model we only find out that α has to be x after we have generated the complete pre-execution.

```
int data = 0;
_Atomic(int) flag = ATOMIC_VAR_INIT(0);

{-{ { // Some code that does not change 'flag'
      // Set 'data' to some value
      atomic_store_explicit(&flag, 1,
        memory_order_release); }
||| { while(atomic_load_explicit(&flag,
                memory_order_relaxed) != 1) {};
      atomic_thread_fence(memory_order_acquire);
      // Read 'data' and use it
      // Some more code
    }
}-};
```

Figure 8. Message passing within a larger program

This means we have to explore all the paths where $\alpha \neq x$ although none of them are consistent.

3. Incrementalising the Axiomatic Model: The Problems

We develop our operational semantics in stages: in the first two stages we develop an operational *concurrency model* that assumes a complete pre-execution given up-front (just like the axiomatic model), and incrementally generates execution witnesses; in the third stage we also incrementally generate the pre-execution (see §4 for a complete description of the stages). In this section we consider the challenges of the first stages, namely how to develop an operational concurrency model that is equivalent to the axiomatic model.

Before we discuss those challenges we introduce some terminology. With *committing* action a we mean adding execution witness relations (rf, mo, lo or sc; see Table 1 for an explanation of the acronyms) between previously committed actions and a. With *following* or *respecting* a certain order r we mean that we commit actions in a way that agrees with r: let com be the commitment order (that is, $(a, b) \in com$ if a has been committed before b), we say that we follow r if for all $(a, b) \in com$ we have $(b, a) \notin r$. For example, if we would commit the actions of the left side of Figure 10 in the order $a, b, c, \ldots, f$ then we would not respect rf because the edge $(f, c) \in rf$ goes against this order.

A requirement that follows from later sections is that we should follow rf. In a complete pre-execution all the reads have a concrete value (that is arbitrarily chosen), but later we want the concurrency model to determine which value is read. Since rf relates reads to the write they read from, this means that the concurrency model has to establish an rf-edge to the read when it commits the read; in other words it has to follow rf.

The first problem we face is that hb edges (happens-before edges) between previously committed actions might disappear when committing new actions. This is conceptu-

ally very strange and it has undesirable consequences, which we discuss in §3.1. In the same section we show that if we follow mo then this problem does not occur.

The other problems follow from the existence of consistent executions with particular cycles. In §3.2 we show that we cannot follow sb (the program order), in §3.3 that we cannot follow sc (the sequential consistent order) and in §3.4 that we cannot follow sw (the synchronises-with order). In §3.1, §3.3 and §3.4 we suggest possible changes to future versions of the C/C++11 model.

3.1 Disappearing Synchronisation

Most kinds of synchronisation are not affected by adding new actions. For example, a synchronising release-acquire pair will be synchronised no matter which or how many new actions are added to the execution, and similarly for a synchronising unlock-lock pair. However, this is not true for types of synchronisations that depend on *release sequences*, as can be seen in Figure 9.

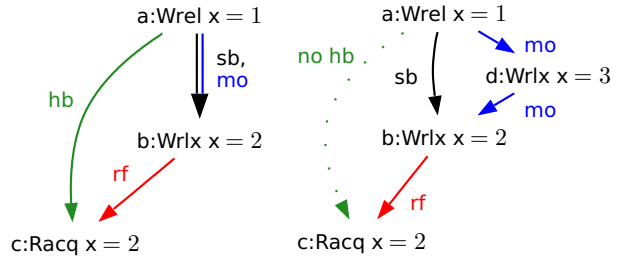

Figure 9. Disappearing synchronisation. On the left side a and b form a release sequence, and because c reads from a part of the release sequence it synchronises with a. On the right side a and b no longer form a release sequence, and therefore c does not synchronise with a anymore.

The rationale behind release sequences is that write-releases are typically implemented by a memory barrier just before the machine write. Independent of whether the read c on the left side of Figure 9 reads from a or b, the barrier of the write-release a needs to be propagated to c's thread first, which means c will synchronise with a in either case. Release sequences are defined to capture this without referring to implementations: a release sequence starts at a write-release w and extends to all mo later writes w' such that w' is either a RWM or to the same thread as w, and such that all writes between w and w' in mo are also either RMWs or to the same thread as w. Then the synchronises-with relation (§2.5) includes pairs (w, r) with r a read-acquire that reads from a write w' in the release sequence of w.

Such a release sequence can be broken by executing a new action, as illustrated in Figure 9. In the execution on the left, the writes a and b are part of a release sequence, and because the read c reads from a write in this sequence, it synchronises with the first write in the sequence. In the second execution, however, a new write d is inserted in modification order between the existing writes a and b, which breaks the release sequence. Therefore, there is no synchronisation between the read c and write a anymore.

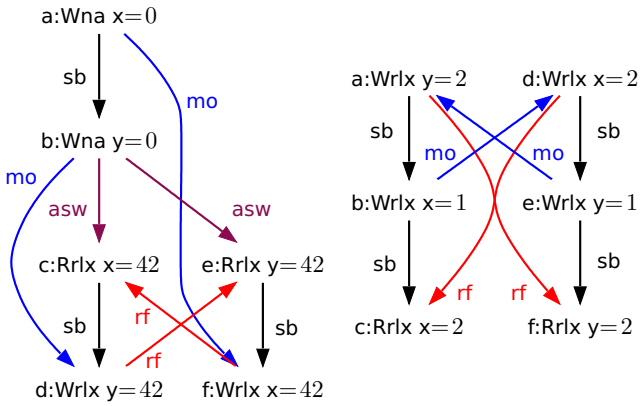

Figure 10. On the left a consistent execution with a cycle in $rf \cup sb$ and on the right one with a cycle in $mo \cup sb$

Such disappearing hb edges make it difficult to construct an operational concurrency model that generates all consistent executions. An hb edge restricts consistent executions in many ways: for example, it restricts the set of writes that a read can read from, and it forces modification order in certain directions. If the concurrency model took those restrictions into consideration but at a later step the hb edge disappeared, the concurrency model would have to reconsider all earlier steps. If on the other hand the concurrency model already took into account that an hb edge might disappear when it encounters an hb edge, the number of possibilities would blow up, and furthermore many executions would turn out to be inconsistent when the hb edge does not disappear after all.

Our solution to prevent disappearing synchronisation is to follow mo when committing actions. We prove that this suffices in a later section, in Theorem 5.5. Another solution would be to change the axiomatic model (and the C/C++ ISO standards) by allowing the release sequence to extend to sb-later writes in the same thread irrespective of whether the write is immediately following in mo order. We believe that this matches hardware behaviour, so this change would not invalidate current implementations of C/C++11.

3.2 Abandoning Program Order

There are two kinds of cycles that show that we cannot follow program order. For the first, recall that the operational concurrency model has to follow rf to determine the return values of reads. Then the cycle in $rf \cup sb$ in the execution on the left of Figure 10 shows that we cannot follow program order (sb) at the same time. This execution has to be allowed in C/C++ because it is allowed on POWER and ARM, and observable on current ARM hardware.

For the second, observe that the execution on the right of Figure 10 has a cycle in $mo \cup sb$. As described in the previous subsection, we follow mo, so the existence of this cycle is another reason we cannot follow program order. This execution is also allowed on POWER and ARM so it has to be allowed in C/C++.

3.3 Abandoning Sequential-Consistent Order

Recall from §2.2 that C/C++11 introduces sequential consistent atomics that are guaranteed to appear in a global total order. When all accesses to atomics have this SC memory order annotation, programs that have no non-atomic races behave as if memory is sequentially consistent (Batty [2, 4]). It is therefore surprising that the concurrency model cannot follow the sc relation when other memory orders are present.

Our argument is as follows. The execution in Figure 11 contains a cycle in $mo \cup rf \cup sc$, so we cannot follow all three relations together. We saw before that we have to follow both rf and mo, hence we cannot follow sc. To the best of our knowledge, this execution is not observable on POWER/ARM, so this suggests another possible strengthening of C/C++11, which would allow an operational model to follow sc by disallowing $mo \cup rf \cup sc$ cycles.

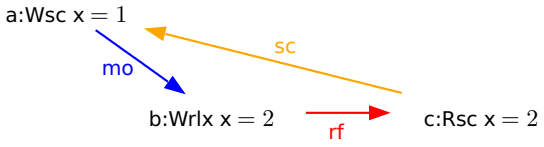

Figure 11. A consistent execution with a cycle in $mo \cup rf \cup sc$ (omitting initialisation)

3.4 Abandoning Synchronises-with Order

Just as disappearing synchronisation makes it hard to develop an operational semantics, new synchronisation to previously committed actions makes it equally hard.

To see this consider the situation where there was no hb edge between a write w and a load r when the load was committed, but committing a new action a creates a hb edge between w and r. The consistency predicate *consistent_non_atomic_rf* requires that a non-atomic read r reads from a write that happens before it[2] (§2.6, observing that $vse \subseteq hb$). When committing r we either have to consider w and discard the execution when there never appears a hb edge, or we do not consider it, but then we have to reconsider the execution of r as soon as there does appear a hb edge. Similarly, the consistency predicate *det_read* requires that r (regardless of whether it is atomic or not) is indeterminate if there does not happen a write before it (§2.6, again observing that $vse \subseteq hb$), so the same problems applies here.

The hb relation is a superset of the synchronises-with (sw) relation, that arises from thread creation, synchronising locks and synchronising release-acquire atomics or fences. If we would have been able to follow sw, it would have been easier to prevent new synchronisation between previously committed actions. However, the execution in Figure 12 has a cycle in $sw \cup rf$, and since we follow rf we can therefore not follow sw. This execution is not observable on POWER/ARM, so again one might conceivably forbid $sw \cup rf$ cycles to allow the operational semantics to follow the sw order.

4. Constructing an Operational Model: Overview

In the rest of the paper we construct the operational semantics in the following three stages.

Stage 1 The incremental concurrency model In §5 we present an order r that can be used to incrementally generate all consistent executions, in contrast to the orders presented in the previous section. The crucial property of the order r is the following: *an r-prefix of a consistent execution is again a consistent execution.*

We use this order to define the *incremental concurrency model* in the following way. We assume for now that a complete pre-execution is given (in a later stage we remove this assumption). We define a notion of state that contains a partially generated execution witness, and we allow a transition from state s_1 to s_2 if s_2 extends s_1 with one action, and s_2 is consistent.

To prove completeness (for finite executions), we exploit that consistency is closed under r-prefixes: let ex be a consistent execution with n actions, define the states $s_0, \ldots, s_n$ where s_i is the r-prefix of ex with i actions. Then the incremental model can transition from s_i to s_{i+1} and therefore it can incrementally generate the consistent execution ex.

Limitations To actually compute a next state s_2 from a state s_1 one would have to enumerate all possible execution witnesses and filter them according to the criteria "s_2 extends s_1 with one action, and s_2 is consistent". Computing behaviour this way is even less efficient than with the axiomatic model itself, since there one would only need to

[2] The requirement that non-atomic reads can only read from happens-before writes holds for all consistent executions, so also for racy executions. This seems counterintuitive: it could potentially make all racy executions of a program inconsistent, which would mean that the program does not have a race according to C11. In fact, there are some out-of-thin-air programs where that happens (Vafeiadis et al. [26] use it to prove certain compiler optimisations unsound), but the good news is that the predicate does not remove any races that occur in mainstream hardware: the proof that the compilation scheme to POWER is sound [5] explains how one can construct a consistent racy C11 execution from a racy POWER execution, and we expect that the soundness proofs of the compilation schemes to ARM and x86 do the same. However, all this is essentially irrelevant for our semantics: C11 requires consistent executions to satisfy *consistent_non_atomic_rf*, and defines races only for those, and therefore we require the same. Because exhaustive exploration with our operational semantics generates all consistent executions, exhaustive exploration will find all C11 races.

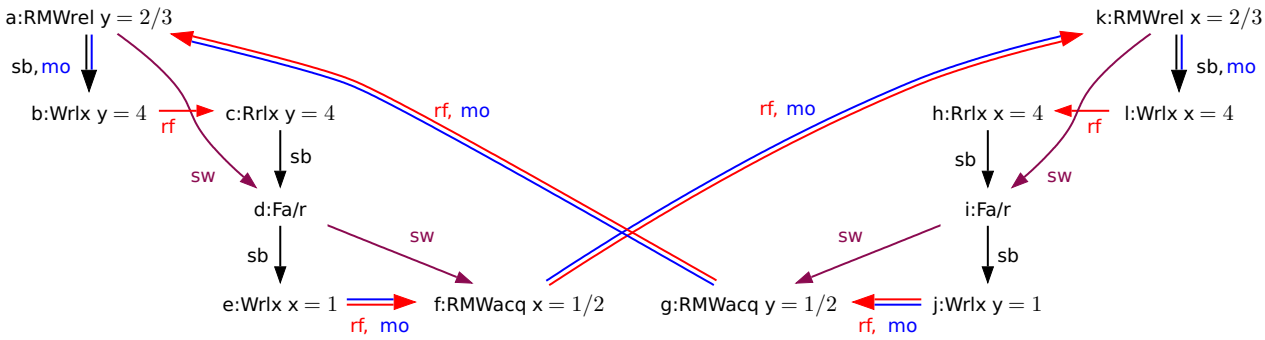

Figure 12. A consistent execution with a cycle in $sw \cup rf$ (omitting initialisation)

enumerate the witnesses once while here for every transition. This limitation is precisely what we solve in the next stage.

Stage 2 The executable concurrency model In §6 we present the *executable concurrency model*. This is similar to the incremental model: it also assumes a complete pre-execution, it has the same notion of states, and it can transition from a state s_1 to s_2 if and only if the incremental model can. The difference is that the executable model defines transitions using a function that given a state s_1 returns the set of all states where s_1 can transition to. This makes it feasible to compute transitions.

We develop this transition function by examining how the relations rf, mo, sc and lo (that together form the execution witness) can change during a transition of the incremental model.

Limitations The transition function internally still enumerates some candidates and filters them using some of the conjuncts of the axiomatic consistency predicate. We believe that the set of a priori possible candidates can be further reduced when we know exactly how hb changes during a transition (instead of the general results stated in Theorem 5.5 and Theorem 5.6); we leave this, which is an implementation optimisation, for future work. The point is that we have to enumerate significantly fewer candidates than in the incremental model: the executable model enumerates at most n^2 candidates where n is the number of actions in the partial witness, while the incremental model enumerates all possibilities for four partial orders over n actions.

The remaining limitation is that the executable model still assumes a complete pre-execution given up-front. This is what we solve in the next stage.

Stage 3 The operational semantics In §7 we integrate the executable concurrency model with an operational model for the sequential aspects of a substantial fragment of C. Here the latter incrementally builds a pre-execution while the concurrency model incrementally builds a witness, synchronising between the two as necessary.

The main obstacle we had to overcome was the fact that the executable concurrency model cannot follow program order (as explained in §3), but the sequential semantics does. Our solution is to allow the sequential semantics and the concurrency model to transition independently of each other: the former *generates* actions in program order, and at every step the concurrency model *commits* zero, one or more of the generated actions.

A consequence of the independent transitions is that when the sequential semantics generates a read, the concurrency semantics might not immediately commit that read and return the value. In that case the sequential semantics has to be able to continue its execution without the return value. Our solution is to make the sequential semantics symbolic: for all reads we use fresh symbols for the return values, and whenever the concurrency model commits a read we resolve the symbol with the value actually read.

When a control operator with a symbolic condition is encountered the sequential semantics non-deterministically explores both branches, adding the corresponding constraints to a constraint set. In some cases the semantics explores a path that leads to an inconsistent constraint set, in which case the execution is terminated. A production tool would need to backtrack or explore a different path at such points, and it would be critical to resolve constraints as early as possible.

The semantics can detect C/C++11 races on the path it explores, but, as for any non-exhaustive semantics, it cannot detect races on other paths.

5. The Incremental Model

In the light of the non-approaches of §3, we now show how one can, given a complete pre-execution (with concrete values for all the reads), incrementally generate witnesses in such a way that every consistent witness over the pre-execution can be generated.

Let ex be a finite consistent execution whose witness we want to incrementally generate. The first step is to find an order $a_1, \ldots, a_n$ of the actions of ex in which we plan the generate the witness; we define this order in §5.1 and

prove that it is acyclic, in contrast to the candidate orders considered in §3.

Then we define the partial executions $ex_1, \ldots, ex_n$ we plan to generate when committing the actions $a_1, \ldots, a_n$, see §5.2. In §5.3 we prove that hb edges do not disappear during a transition from ex_i to ex_{i+1}, and neither do there appear new hb edges between previously committed writes and reads (in respectively §3.1 and §3.4 we discussed why we need those properties).

Then in §5.4 we prove that the partial executions $ex_1, \ldots, ex_n$ are all consistent if ex is consistent, and, based on that, we define a transition relation in §5.5. Finally, we define the incremental model in §5.6 and prove equivalence with the axiomatic model for finite executions.

Notation Recall from §2 that an execution consists of a pre-execution, an execution witness and derived relations. The function that derives those relation is get_rel, so $ex = (pre, wit, get\_rel(pre, wit))$.

We use the notation $pre.sb$ and $wit.rf$ to refer to parts of pre-executions and execution witnesses. For brevity, we abuse this notation by writing $ex.sb$ when we should actually write "let $ex = (pre, wit, rel)$, consider $pre.sb$" and likewise for the parts of the witness (such as $ex.rf$) and derived relations (such as $ex.hb$).

5.1 The Commitment Order

Recall that the operational concurrency model has to follow rf to determine the return values of reads, and it has to follow mo in order to preserve earlier synchronisation (see §3.1). To prevent the situation described in §3.4 regarding the predicates *consistent_non_atomic_rf* and *det_read*, it is enough to prevent new synchronisation appearing between previously committed *writes* and *loads*. To achieve this, we also follow $\{(a, b) \in hb \mid is\_load(b)\}$.

This order satisfies all the properties we would need to incrementalise the axiomatic model, but it leaves many actions unordered, which means that the transition relation would be very non-deterministic. To reduce this non-determinism as much as possible, we include as much of hb as we can. Because we cannot follow program order (see §3.2) we know that we cannot include all of hb.

We decided to leave out hb edges to atomic writes, and include all hb edges to other types of actions. (For locks and unlocks there is a choice whether to include hb edges to locks and unlocks, or to follow the lock-order lo, but one cannot include both since there can be a cycle in their union. We did not see any compelling argument in favour of either of the two, and we chose to follow the former.) In other words, this order allows us to speculate writes, and forces us to commit all other actions in hb order.

Definition 5.1 (Commitment order)**.** Let ex be a candidate execution. First define $ex.almost\_hb =$

$$\{(a, b) \in ex.hb \mid \neg (is\_write(b) \wedge is\_atomic(b))\}.$$

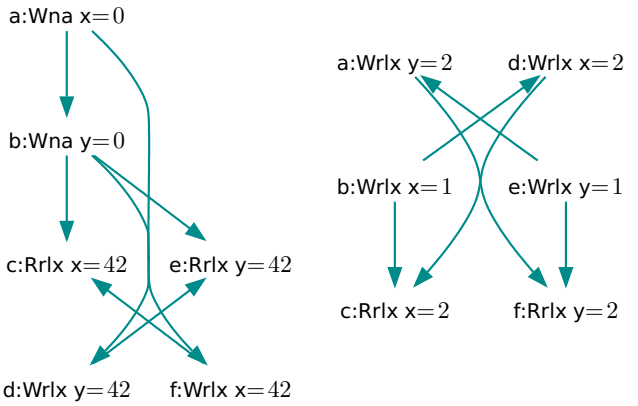

Figure 13. The commitment orders of the executions in Figure 10. Note that these are strict partial orders and do not contain any cycles.

Then define $ex.com = (ex.rf \cup ex.mo \cup ex.almost\_hb)^+$ where $\cdot^+$ is the transitive closure.

Theorem 5.2. *Let ex be consistent. Then the relation $ex.com$ defined above is a strict partial order.*

The proof, like all our work, has been mechanised in Isabelle/HOL and is included in the supplementary material.

5.2 States

A state s consists of a set of actions $s.committed$ denoting the actions that have been committed so far, and an execution witness $s.wit$ denoting the execution witness built up so far. Note that the pre-execution is not part of the state, since it is given up-front.

Let ex be the execution that we want to incrementally generate, and $a_1, \ldots, a_n$ the actions of that execution in some order that agrees with $ex.com$ defined in the previous subsection. We want the states $s_1, \ldots, s_n$ to reflect the witness built up so far, and an obvious thing to do is to define $s_i.committed$ to be the actions $a_1, \ldots, a_i$ that are committed so far, and $s_i.wit$ as the restriction of $ex.wit$ to those actions. The initial state s_0 is always the same (regardless of the given pre-execution) because $s_0.committed = \emptyset$ and $s_0.wit$ the empty witness.

Definition 5.3. Let pre be a pre-execution, and S a set of actions. Then $preRestrict(pre, S)$ is defined by

$$\begin{aligned} preRestrict(pre, S).actions &= pre.actions \cap S \\ preRestrict(pre, S).sb &= pre.sb \cap S \times S \\ preRestrict(pre, S).asw &= pre.asw \cap S \times S \end{aligned}$$

Similarly, with wit an execution witness, $witRestrict$ is defined by restricting rf, mo, sc and lo to $S \times S$, as in

$$witRestrict(wit, S).rf = wit.rf \cap S \times S$$

And finally, with $ex = (pre, wit, rel)$ an execution, $exRestrict$ is defined by

$$\begin{aligned} pre' &= preRestrict(pre, S) \\ wit' &= witRestrict(wit, S) \\ exRestrict(ex, S) &= (pre', wit', get\_rel(pre', wit')) \end{aligned}$$

The partial executions ex_i mentioned in the intro of this section are then given by $exRestrict(ex, A_i)$ where $A_i = \{a_1, \ldots, a_i\}$. Note that we have also restricted the pre-execution to the set of actions committed, although the complete pre-execution is fixed during the generation of the witness. We have two reasons for that: one is that otherwise the partial execution would be inconsistent (since the actions in the pre-execution that have not been committed yet have no mo, rf, etc. edges to and from them, while this is in some cases required to be consistent). And the second reason is that when we integrate with the operational threadwise semantics, the pre-execution is no longer fixed.

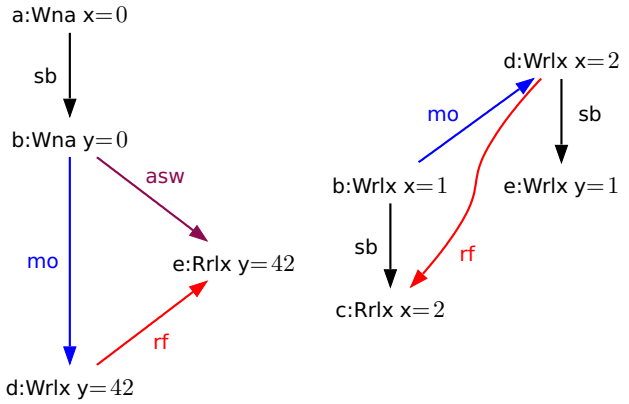

Figure 14. On the left $exRestrict(ex_\ell, \{a, b, d, e\})$ and on the right $exRestrict(ex_r, \{b, c, d, e\})$ where ex_ℓ and ex_r are respectively the executions on the left and right of Figure 10

5.3 Properties of Happens-before

In §3.1 we explained that synchronisation could disappear when mo is not followed. Since we have included mo in the commitment order, the counterexample does not apply anymore, and we can prove that hb grows monotonically. We use the following auxiliary definition for that.

Definition 5.4. Let r be a relation over actions, and A a set of actions. Then $downclosed(A, r)$ holds if and only if for all $(a, b) \in r$ with $b \in A$ we have that $a \in A$.

For example $downclosed(A, ex.mo)$ means that there are no mo edges from outside A into A. Now the following monotonicity theorem states that if that is true for A, then the restriction of ex to A does not contain any hb edges that are not in ex, or in other words none of the hb edges disappeared.

Theorem 5.5. *Let ex be an execution. Let A be a set of actions with $downclosed(A, ex.mo)$. Then $(exRestrict(ex, A)).hb \subseteq ex.hb$.*

Recall that in §3.4 we mentioned another desirable property of how hb changes: there should not appear new synchronisation between previously committed writes and loads. We prove a slightly stronger result: there does not appear new synchronisation between any type of action to an action that is not an atomic write.

Theorem 5.6. *Let ex be a consistent execution. Let A be a set of actions such that $downclosed(A, ex.com)$. Then for all $(a, b) \in ex.hb$ with $b \in A$ and b not an atomic write, we have that $(a, b) \in (exRestrict(ex, A)).hb$.*

5.4 Consistency of Prefixes

Now we know how hb changes during incremental generation of executions, we can prove that the partial executions $exRestrict(ex, A_i)$ (as defined in §5.2) are consistent, where A_i is the set of actions committed so far. This means that every consistent execution can be build incrementally while being consistent at every step.

Theorem 5.7. *Let A be a set of actions such that $downclosed(A, ex.com)$. If ex is a consistent execution, then $exRestrict(ex, A)$ is a consistent execution.*

5.5 Transition Relation

Given a consistent execution ex, an order $a_1, \ldots, a_n$, and the partial executions $ex_i = exRestrict(ex, \{a_1, \ldots, a_i\})$, we now define a transition relation that allows the transition between ex_i and ex_{i+1}. This ensures completeness: if we use this transition relation to follow paths from the initial state (containing an empty witness) we know that we will generate all consistent executions.

The transition relation $incrementalStep(pre, s_1, s_2, a)$ is intended to hold if committing a in state s_1 can result in state s_2, given the pre-execution pre (recall that we still assume to be given a complete pre-execution). The transition relation has several conjuncts, which we describe after giving the definition.

Definition 5.8.
The relation $incrementalStep(pre, s_1, s_2, a)$ is defined as

$$a \in pre.actions \wedge \quad (1)$$
$$a \notin s_1.committed \wedge \quad (2)$$
$$s_2.committed = s_1.committed \cup \{a\} \wedge \quad (3)$$
$$witRestrict(s_2.wit, s_1.committed) = s_1.wit \wedge \quad (4)$$
$$\big[\forall b \in pre.actions.$$
$$(b \in s_1.committed \rightarrow (a, b) \notin ex.com) \wedge$$
$$((b, a) \in ex.com \rightarrow b \in s_1.committed)\big] \wedge \quad (5)$$
$$isConsistent(ex_{prefix}) \quad (6)$$

where ex and ex_{prefix} are defined by

$$\begin{aligned} ex &= (pre, s_2.wit, get\_rel(pre, s_2.wit)) \\ pre_{prefix} &= preRestrict(pre, s_2.committed) \\ ex_{prefix} &= (pre_{prefix}, s_2.wit, get\_rel(pre_{prefix}, s_2.wit)) \end{aligned}$$

Conjunct (1) makes sure that an action of the pre-execution is committed (and not an arbitrary action), Conjunct (2) that the action a has not been committed yet, and Conjunct (3) that the set of committed actions is updated correctly during the transition. Conjunct (4) ensures that all the changes to the witness involve the new action a; in other words, the execution witness restricted to the old set of committed actions is still the same. Conjunct (5) ensures that actions are committed according to the commitment order, and finally Conjunct (6) ensures that the generated partial execution is consistent ($isConsistent$ is the axiomatic consistency predicate).

We define that $incrementalTrace(pre, s)$ holds if s is reachable from the initial state following $incrementalStep$. The following states that all consistent executions are reachable.

Theorem 5.9. *Let ex be a consistent, finite execution. Let A be a set of actions with $A \subseteq ex.actions$ and $downclosed(A, ex.com)$. Then there exists a state s with*

$$\begin{aligned} s.committed &= A \\ s.wit &= witRestrict(ex.wit, A) \end{aligned}$$

such that $incrementalTrace(pre, s)$.

5.6 The Incremental Model

We now define a new notion of consistency that uses $incrementalTrace$, which is equivalent to the axiomatic consistency predicate for finite executions.

Definition 5.10. Let $ex = (pre, wit, get\_rel(pre, wit))$ be a candidate execution. We define

$$\begin{aligned} &incrementalConsistent(ex) = \\ &\exists s.\big(incrementalTrace(pre, s) \wedge \\ &\quad s.wit = wit \wedge s.committed = pre.actions\big) \end{aligned}$$

Theorem 5.11 (Equivalence)**.** *Let ex be a finite candidate execution with $ex = (pre, wit, get\_rel(pre, wit))$. Then $incrementalConsistent(ex)$ holds if and only if ex is consistent according to the axiomatic model.*

6. An Executable Model

In the previous section we saw that all finite consistent witnesses can be generated incrementally: starting from the initial state s_0 we follow $incrementalStep(pre, s_i, s_{i+1}, a_i)$ to generate the states $s_1, \ldots, s_n$ until we have committed all the actions of the pre-execution. The problem is that $incrementalStep$ is a relation, so to actually compute a state s_{i+1} from the state s_i we have to enumerate states until one of them satisfies $incrementalStep$.

In this section we define a step function $executableStep$ that given a state and a pre-execution, returns the set of possible next states, which makes it feasible to compute executions incrementally.

To find out how we should define the step function we investigate how s_{i+1} differs from s_i when $incrementalStep(pre, s_i, s_{i+1}, a_i)$ holds. For the set of committed actions this is clear: $s_{i+1}.committed = s_i.committed \cup \{a\}$ since this is directly required by $incrementalStep$ (see Definition 5.8). For the witness this is not immediately obvious, so we investigate this in the following sections: in §6.1 we consider the mo relation, in §6.2 the rf relation, and in §6.3 the sc and lo relations. Then in §6.4 we define the step function.

6.1 Modification Order

We consider how mo can change from s_i to s_{i+1} when action a is committed. In consistent executions, mo is an order over atomic writes that is total over the writes of the same location. We therefore expect mo to remain the same if a is not an atomic write, and a to be added to mo otherwise. Since the modification order is included in the commitment order, we expect that a can only be added to the end of the existing mo order. To state that formally, we define a function $addToMo$ that adds an action a at the end of the modification order of a state s.

Definition 6.1. Define $sameLocWrites(A, a)$ as

$$\{b \in A \mid is\_write(b) \wedge loc\_of(b) = loc\_of(a)\}.$$

Then define $addToMo(a, s)$ as $s.wit.mo \cup \{(b, a) \mid b \in sameLocWrites(s.committed, a)\}$.

We now formally state our expectations of how mo changes. The state s should be thought of as the current state, and ex as the execution we try to transition to. We explain the assumptions afterwards.

Lemma 6.2. *Let s be a state, ex an execution and a an action, for which the following holds.*

$$a \notin s.committed \tag{7}$$
$$ex.actions = s.committed \cup a \tag{8}$$
$$witRestrict(ex.wit, s.committed) = s.wit \tag{9}$$
$$downclosed(s.committed, ex.mo) \tag{10}$$
$$isConsistent(ex) \tag{11}$$

If a is an atomic write, we have $ex.mo = addToMo(a, s)$ and otherwise we have $ex.mo = s.wit.mo$.

Assumptions (7) and (8) together state that there is one new action in ex. Then (9) states that the witnesses of ex and s agree on the part that is already committed in s; assumption (10) states that so far, the execution has followed mo; and

finally, (11) states that ex is consistent. The conclusion of the lemma then says that if a is an atomic write, the modification order of s changes according to $addToMo$, and otherwise it does not change.

6.2 Reads-from Relation

We consider how rf can change from s_i to s_{i+1} when action a is committed. In consistent executions, rf is a relation from writes to reads. Because rf is included in the commitment order, we only expect new rf edges *to* the new action a and not from a. Hence, how rf changes depends on whether a is a load, an RMW, or neither.

In case a is a load, the first observation is that there could be multiple writes where a could read from. To account for this we let $addToRfLoad$ non-deterministically return a new rf relation (which is mathematically modelled as a function that returns a set of rf relations). There is also the possibility that a does not read from anywhere. The consistency predicate *det_read* describes when this happens: if there exists a write that happens before a then a should read from somewhere, otherwise it should not. This could be self-satisfying: if there is no write that happens before a, creating a rf edge might create hb edge from a write to a which would then make *det_read* true. Hence, we non-deterministically choose to create a rf edge or not, and when the new hb relation is known, we check whether there should have been an edge or not.

Definition 6.3. Define $addToRfLoad(a, s)$ as follows. First, non-deterministically choose between returning $s.wit.rf$ (meaning no new edge is added), or non-deterministically picking a write w from the set $sameLocWrites(s.committed, a)$ for which we have $value\_written\_by(w) = value\_read\_by(a)$ and returning $s.wit.rf \cup \{(w, a)\}$.

In the second case (where a is an RMW), the consistency predicate *rmw_atomicity* requires that a reads from its immediate mo-predecessor if there is one, and otherwise it should be indeterminate (not reading from any write).

Definition 6.4. Define $addToRfRmw(a, s)$ as follows. If the set $sameLocWrites(s.committed, a)$ is empty, return $s.wit.rf$. Otherwise, there is a mo-maximal element w of that set. We check whether $value\_written\_by(w) = value\_read\_by(a)$ holds, and if so, we return $s.wit.rf \cup \{(w, a)\}$.

We can now formally state our expectations about how rf changes during a transition. For the explanation of the assumptions we refer to the explanation given after Lemma 6.2.

Lemma 6.5. *Let s be a state, ex an execution and a an action for which*

$$
\begin{aligned}
&a \notin s.committed \\
&ex.actions = s.committed \cup a \\
&witRestrict(ex.wit, s.committed) = s.wit \\
&downclosed(s.committed, ex.mo) \\
&downclosed(s.committed, ex.rf) \\
&isConsistent(ex)
\end{aligned}
$$

(1) If a is a load, then $ex.rf \in addToRfLoad(a, s)$.
(2) If a is a RMW, then $ex.rf \in addToRfRmw(a, s)$.
(3) Otherwise we have $ex.rf = s.wit.rf$.

6.3 SC and Lock Order

In consistent executions, sc is a total order over all actions with an SC memory order, and lo is an order over locks and unlocks that is total per location. Because there exist cycles in $sc \cup com$ and in $lo \cup com$, we have to allow the new action a to be inserted before already committed actions in either order. Our approach is to define the functions $addToSc$ and $addToLo$ that non-deterministically insert a anywhere in respectively sc or lo, and later filter the possibilities that became inconsistent.

Then we prove a lemma similar to Lemma 6.2 and Lemma 6.5 that shows that this construction suffices: if a has a sequential consistent memory order, we have $ex.sc \in addToSc(a, s)$ and otherwise we have $ex.sc = s.wit.sc$; if a is a lock or an unlock, we have $ex.lo \in addToLo(a, s)$ and otherwise we have $ex.lo = s.wit.lo$.

6.4 The Transition Function

With the results of §6.1, 6.2 and 6.3 it is now straightforward to define a non-deterministic function $performAction(s, a)$ that returns an execution witness based on the type of a.

- Loads: we change rf with $addToRfLoad$. If the memory order of a is SC, we change the sc relation with $addToSc$.
- Stores: if a is atomic we change mo with $addToMo$. If the memory order is SC we change sc with $addToSc$.
- RMWs: we change rf with $addToRfRmw$, mo with $addToMo$, and if the memory order is SC then sc with $addToSc$.
- Locks and unlocks: we change lo with $addToLo$.
- Fences: if the memory order is SC we change sc with $addToSc$.

Definition 6.6. Define $executableStep(pre, s)$ as follows. First non-deterministically pick an action $a \in pre.actions$ with $a \notin s.committed$. Then, non-deterministically generate a witness wit using $performAction(s, a)$. Define the new state s_2 with $s_2.committed = s.committed \cup \{a\}$ and $s_2.wit = wit$. Finally, check whether our choice followed the commitment order and resulted in a consistent execution

by discarding states that do not satisfy Conjunct (5) or Conjunct (6) of Definition 5.8. For each of the non-discarded options, the function returns the pair (s_2, a).

Theorem 6.7. *We have* $(s_2, a) \in executableStep(pre, s_1)$ *if and only if* $incrementalStep(pre, s_1, s_2, a)$.

Define $executableTrace$ and $executableConsistent$ in the same way as in the incremental model (Definition 5.10), but then using $executableStep$ instead of $incrementalStep$. From the previous theorem and from Theorem 5.11 it then follows that the executable model is equivalent to the axiomatic model for finite executions:

Corollary 6.8. *Let* ex *be a finite candidate execution with* $ex = (pre, wit, get\_rel(pre, wit))$. *Then* $executableConsistent(ex)$ *holds if and only if* ex *is consistent according to the axiomatic model.*

7. Integration with the Threadwise Model

In the previous section we defined an executable transition function, but we still assumed that we are given a complete pre-execution with concrete values for all the reads. We now integrate that executable model with an operational threadwise semantics that builds pre-executions incrementally.

As the front-end language, we use a small functional programming language with explicit memory operations (Core). This is developed as an intermediate language in a broader project [16] to give semantics of C; as such, any C program can be elaborated to a Core program.

The challenge here is that the operational semantics of Core follows program order, while the executable concurrency model does not. Our solution is to let the two models take transitions independently of each other, so the former can follow program order, while the latter follows the commitment order. A consequence of this is that the concurrency model does not always immediately commit a read when the threadwise semantics has generated it, which means that the threadwise semantics does not know the return value, but at the same time it has to be able to continue the execution. Our solution is to continue the execution symbolically.

We describe the interaction between the operational semantics of Core and the executable concurrency model in §7.1 and the validation in §7.2. The symbolic execution has significant drawbacks and one might hope that it is only needed for atomics, but in §7.3 we show that it is also necessary for non-atomics. Then in §7.4 we discuss what remains necessary to produce a more generally usable tool.

7.1 The Interaction with the Threadwise Model

The integrated semantics starts with an empty pre-execution, and then goes on to alternate between performing one step of the Core dynamics and zero or more steps of the concurrency model.

The Core dynamics is a step function: from a given Core program state it returns the set of memory actions (and the resulting Core program state should that operation be performed) that can be performed at this point by the program. These actions are communicated to the concurrency model by adding them to the pre-execution. For load operations, the resulting Core program state needs a read value. Since the concurrency model may choose not to provide a value immediately, we introduce a symbolic name for the value read, and use it to build the resulting Core state.

As a result all values in Core programs must be symbolic. This means in particular that the execution of control operators is done symbolically. When a control point is reached, the threadwise semantics non-deterministically explores both branches, under corresponding symbolic constraints for each branch.

When the concurrency model does give an answer for a read, at some later point in the execution, the set of constraints is updated by asserting an equality between the symbolic name created earlier for the read and the actual value. In the case of execution branches that should not have been taken, the constraint therefore becomes unsatisfiable and the execution path is killed. Our C semantics elaborates the many C integral numeric types into Core operations on mathematical integers, so all constraints are simply over those.

This solves the fundamental problem we stated in the introduction: although the concurrency model cannot always immediately determine the value of a read, it does so *during* the generation of the pre-execution which avoids exploring many incompatible control-flow unfoldings.

7.2 Validation

The correctness of the concurrency model is guaranteed by the equivalence theorem. To validate the integration we have run the semantics on the following classic litmus test programs (these tests are available in the supplementary material):

- Message passing: a version with a write-release, a relaxed read in a loop, and an acquire fence (see Figure 3)
- Load buffering: a version with relaxed atomics (that allows the cycle given on the left of Figure 10), a version with release/acquire atomics, and a version with SC atomics.
- Store buffering: a version with relaxed atomics, a version with release/acquire atomics, and a version with SC atomics.
- A program that allows a cycle in $mo \cup sb$ (see Figure 10).
- WRC: a version with a write-releases, load-acquires in loops, and a relaxed read.

For each test, pseudo-random exploration revealed all the allowed outcomes (and never forbidden outcomes). For the relaxed version of LB and for the mo-sb-cycle program the outcomes with cycles in respectively $rf \cup sb$ and $mo \cup sb$ happened rarely: only approximately 1 out of a 1000 runs

exhibited them (we include runs that result in a dead-end in this number). For all the other tests all the allowed outcomes where generated in the order of 10 runs.

7.3 Symbolic Execution Unavoidable for Non-atomics

One drawback of the symbolic execution is that we lose completeness if the constraint generation and solver cannot handle the full generality of constraints (e.g. for memory accesses from pointers computed in complex ways). One might hope to only need symbolic execution for atomics, and that one could always immediately return a concrete value for non-atomics, but unfortunately the following shows that this is not the case.

Consider the execution in Figure 12 and imagine a non-atomic write w_1 to a new location (say z_1) that is sb-before action a, and similarly a new write w_2 that is sb-before action k; and imagine a non-atomic read r_1 of z_1 that is sb-between actions d and e, and similarly a read r_2 that is sb-between actions i and j. Suppose without loss of generality that when r_1 is generated by the threadwise semantics, r_2 has not yet been generated. The latter means that j cannot have been generated (since the threadwise semantics follows program order), and therefore that g, a, b and c have not been committed by the concurrency model (because the concurrency model follows rf and mo). Hence, the hb edge between w_1 and r_1 does not exist yet, and therefore we do not know where r_1 can read from at this time (see also §3.4) and the threadwise semantics has to use a symbol as its return value.

7.4 Outstanding Issues

Extending the operational semantics to support random-mode execution of more realistic C programs requires at least three significant advances. First, the C/C++11 concurrency model, in both axiomatic and operational forms, must be extended to support aspects of C neglected by Batty et al. [6], including general array, struct, and mixed-size accesses, object lifetime, and dynamic errors. Second, the implementation of constraints must support those that arise from realistic pointer arithmetic (ideally including bitwise operations). Third, there will need to be performance optimisation, as at present the state size (and transition compute time) grows with trace length, but in principle "sufficiently old" information can be garbage-collected.

8. Related Work

There is a long history of equivalence or inclusion results between operational and axiomatic relaxed memory models, e.g. Higham et al. [12], Owens et al. [20], Alglave et al. [1], and Cenciarelli et al. [9], but very little that relates to the C/C++11 model issues that we address here (the first three of those address hardware models, where concrete operational models provide a usable order; the last is in the rather different JMM context).

The most closely related work that we are aware of is the work by Lahav et al. [13]. The authors study the fragment of C/C++11 in which all read, write, and read-modify-write accesses have release/acquire memory orders, without relaxed, consume, SC, or non-atomic accesses, and with just a single kind of fence. They identify that the execution presented in §3.2 with read-acquires and write-releases instead of relaxed accesses is not observable in implementations, and go on to prove that the existing compilation schemes to POWER and x86-TSO can still be used when forbidding hb-mo-cycles. For this stronger release/acquire semantics (where those cycles are forbidden) they give a concrete operational semantics in terms of ordered message buffers and memory local to processors, and their results are largely also mechanised (in Coq). However, the release/acquire fragment of C/C++11 is considerably simpler than the full model we deal with here. For example, in that fragment the sb-rf and sc-mo-rf cycles that we address do not occur. They also work with a small calculus rather than integrating their model with a larger C semantics.

The operational semantics by Turon et al. [25] covers non-atomics, SC-atomics and release/acquire atomics, but not relaxed or consume atomics. It is precisely these memory orders that make developing an equivalent operational semantics hard. Furthermore, their semantics simplifies some of the concepts of the axiomatic model to give a cleaner semantics, at the expense of completeness. For their purposes this is not a problem, since they are developing a sound program logic, but our goal is to develop an equivalent model.

The other most closely related work we are aware of is the model-checker of Norris and Demsky [19]. This is focused on efficiency, but does not attempt to be equivalent with respect to the C/C++11 model. Our operational model may inform future work on C/C++11 model-checking.

More peripherally, two lines of work have integrated a TSO memory model with a semantics for significant fragments of C: the CompCertTSO verified compiler of Ševčík et al. [27], and the K semantics of Ellison [11, §4.2.6]. TSO is much stronger and simpler than C/C++11, and there cannot be cycles in $hb \cup rf$, so the concurrency impacts much less on the sequential semantics. Moreover, mainstream C compilers do not implement TSO, so the significance of such a semantics for concurrent C/C++11 programs is unclear.

Then there is work using SAT solvers for axiomatic models, for C/C++11 by Blanchette et al. [7] and for the JMM by Torlak et al. [24]. For litmus tests these offer performance improvements w.r.t. naive enumeration of candidate executions, but finding single paths of larger programs seems likely to be challenging, as does integration with a more substantial C semantics.

Finally, there are also a number of less closely related proposals for other language-level memory models [10, 14].

9. Conclusion

We have presented an operational concurrency model that covers the full formalisation [6] of C/C++11 concurrency including locks, fences, read-modify-writes, non-atomics and atomics with all memory orders, including consume. We have proved the equivalence of our model with that formalisation and mechanised the proof in Isabelle/HOL. We have also integrated the operational concurrency model with a sequential operational semantics [16] (the sequential semantics is not our contribution); the combined semantics can incrementally execute programs in a small fragment of C.

The challenge in defining the operational model was the fact that many obvious approaches such as following program order or the sequential consistency order do not work, because C/C++11 allows cycles in various orders. These cycles are not always observed on current hardware, and in these cases we suggested strengthening the C/C++11 model: we suggested to forbid coherence shapes that involve sc (§3.3), cycles in $sw \cup rf$ (§3.4) and we suggested changing the definition of release-sequences (§3.1).

More generally, we highlight two so-far underappreciated qualities that a programming language concurrency semantics should have. It should be incrementally executable, and it should be integrable (better yet, integrated) with the semantics for the rest of the language, not just a memory model in isolation. Leaving such integration for future work may lead to a memory model that makes it remarkably involved. Since the sequential part of most languages are defined in an operational style (including C/C++) these requirements can be best satisfied by developing an equivalent operational concurrency semantics early in the process.

Acknowledgements

We thank Mark Batty for discussions and the anonymous referees for their comments. This work was partly funded by a Gates studentship (Nienhuis) and by the EPSRC Programme Grant *REMS: Rigorous Engineering for Mainstream Systems*, EP/K008528/1.

References

[1] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM TOPLAS*, 36(2):7:1–7:74, 2014.

[2] Mark Batty. *The C11 and C++11 Concurrency Model*. PhD thesis, University of Cambridge, 2015. `https://www.cs.kent.ac.uk/people/staff/mjb211/toc.pdf`.

[3] Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. In *Proc. POPL*, pages 235–248, 2013.

[4] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In *Proc. ESOP*, pages 283–307. 2015.

[5] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Proc. POPL*, pages 509–520, 2012.

[6] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *Proc. POPL*, 2011.

[7] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. Nitpicking C++ concurrency. In Peter Schneider-Kamp and Michael Hanus, editors, *Proc. PPDP*, pages 113–124, 2011.

[8] Hans-J Boehm and Sarita V Adve. Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008.

[9] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *Proc. ESOP*, pages 331–346, 2007.

[10] Karl Crary and Michael J. Sullivan. A calculus for relaxed memory. In *Proc. POPL*, pages 623–636, 2015.

[11] Chucky Ellison. *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois, July 2012.

[12] Lisa Higham, Lillanne Jackson, and Jalal Kawash. Specifying memory consistency of write buffer multiprocessors. *ACM TOPLAS*, 25(1), February 2007.

[13] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In *Proc. POPL*, pages 649–662, 2016.

[14] Jeremy Manson, William Pugh, and Sarita V Adve. *The Java memory model*, volume 40. ACM, 2005.

[15] Paul E. McKenney, Torvald Riegel, Jeff Preshing, Hans Boehm, Clark Nelson, and Olivier Giroux. N4321: Towards implementation and use of memory order consume. WG21 working note, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4321.pdf`, October 2014.

[16] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N.M. Watson, and Peter Sewell. Into the depths of C: elaborating the de facto standards. In *Proc. PLDI*, 2016.

[17] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proc. PLDI*, 2013.

[18] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In *Proc. ICFP*, pages 175–188, 2014.

[19] Brian Norris and Brian Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Proc. OOPSLA*, 2013.

[20] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Theorem Proving in Higher Order Logics*, pages 391–407. 2009.

[21] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proc. POPL*, 2016.

[22] Jaroslav Ševčík and Peter Sewell. C/C++11 mappings to processors. `http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`. Accessed 2015-07-08.

[23] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *Proc. PLDI*, pages 110–120, 2015.

[24] Emina Torlak, Mandana Vaziri, and Julian Dolby. Memsat: Checking axiomatic specifications of memory models. In *Proc. PLDI*, pages 341–350, 2010.

[25] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *Proc. OOPSLA*, pages 691–707, 2014.

[26] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proc. POPL*, pages 209–220, 2015.

[27] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), June 2013.

[28] WG14. ISO/IEC 9899:2011.

[29] WG14 and WG21. ISO/IEC 14882:2011.

[30] WG14 and WG21. ISO/IEC 14882:2014.

[31] WG21. N3786.