An operational semantics for C/C++11 concurrency

Kyndylan Nienhuis  Kayvan Memarian  Peter Sewell
University of Cambridge
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, SC 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.

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]. 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 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 forced by hardware-observable behaviour, but the latter two (about SC order and synchronises-with order) are not, and so might be ameliorated by changing C/C++11.

**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 nonatomic 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 optimisation; 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 nonatomic 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, will be 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 as anonymous supplemental material submitted with this paper.
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.

2.1 The language: C/C++11 concurrency primitives

C/C++11 provide concurrency primitives supporting a range of different programming idioms. First there are normal non-atomic accesses. Races on these give rise to undefined behaviour (to allow compiler optimisation to assume there are no races), and so concurrent use of them must be protected by conventional locks or other synchronisation. Then there are atomic accesses, which can be concurrently used without constituting undefined behaviour. Atomic accesses include memory reads, writes, and various read-modify-write operations, including atomic increments and compare-and-swap operations. There are also explicit memory fences.

The program in Fig. 1 uses atomics and a fence to synchronise two threads. The first thread non-atomically sets \(x\) to a value and then signals the other thread by atomically setting \(y\) to 1. The other threads reads \(y\) in a loop until it sees 1, and then uses \(x\) for some other computation.

Atoms can be annotated with different memory orders. The example uses a write-release, a read-acquire and a relaxed memory order.

- Sequentially consistent (SC) atomics are guaranteed to appear in a global total order, but their implementation on relaxed hardware requires relatively expensive synchronisation.
- Write-release and read-acquire atomics are cheaper but weaker: 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 difficult 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].

Certain combinations of release/acquire, relaxed, and read-modify-write atomics and fences also guarantee synchronisation (exploiting the force of the memory barriers used in write-release implementations). In the example in Fig. 1 the write-release and the acquire fence synchronise.

2.2 The C/C++11 threadwise 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. This factorisation works via pre-executions: the threadwise semantics determines the set of pre-executions of a program, and the concurrency model is defined in terms of pre-executions.

Pre-executions A pre-execution corresponds to a particular complete control-flow unfolding of the program and an arbitrary choice of the values read from memory. It 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 (the synchronisation strength of the action, not an order relation) is specified here: sequential consistent (sc), release (rel), acquire (acq), acquire-release (a/r), consume (con) or relaxed (rbl). Locks and unlocks do not have a memory order.
- \(x\), the location of the action. Fences do not have a loca-
- \(0\), 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 Fig. 1 we see one of the pre-executions of the message passing program. The program has infinitely many 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.

2.3 The C/C++11 axiomatic concurrency semantics

To determine the behaviour of a program given its set of pre-executions, we first extend each pre-execution with all possible execution witnesses; a pre-execution combined with one of its execution witnesses forms a candidate execution.


```c
#include <stdatomic.h>

int main(void) {
    int x = 0;
    Atomic(int) y = 0;
    int z;
    {- { x = 1;
        atomic.store.explicit(&y, 1, memory.order.release); }
    }|| | { while(atomic.load.explicit(&y, memory.order.relaxed) != 1) {};
        atomic.thread.fence(memory.order.acquire);
        z = x; }-};
    return z;
}
```

On the left: 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 pthread-style thread creation.

On the right: one of the pre-executions of the program. The condition of the loop is execute twice: actions \(f\) and \(g\) both correspond to the same instruction. The values \(\alpha\), \(\beta\), \(\gamma\), and \(\delta\) are read from memory are arbitrary, but do 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 (see §2.2).

![Figure 1: The message passing (MP) program and one of its pre-executions](image)

Then we use the axiomatic model to determine which of those candidate executions is consistent. Finally, we check whether any of the consistent executions contains a race.

**Execution witnesses** An execution witness 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 \(lo\). In principle every pre-execution can be extended by every execution witness to form a candidate execution. It is the next step (determining which candidate executions are consistent) that gives meaning to the relations defined above.

**Consistency** The axiomatic concurrency model defines a consistency predicate that determines whether a candidate execution is consistent or not. 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-executions, \(w\) 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.
- **atomic_write** requires that \(asw\) is a write, \(asw\) is a read from memory are arbitrary, but do 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 (see §2.2).

Other axioms define the more subtle properties that are the real substance of the C/C++11 model. These axioms typically use the following derived relations.

- The synchronises-with relation \(sw\) contains (among other things) the \(asw\) relation, synchronising unlock-lock pairs, and synchronising release-acquire pairs.
- The happens-before relation \(hb\). In the absence of the memory order consume, we have that \(hb = (sb \cup sw)^+\) where \(^+\) is the transitive closure.

In Fig. 2 we see a consistent execution of the message passing program. There is only one choice for the arbitrary values \(\alpha\), \(\beta\), \(\gamma\), and \(\delta\) of the pre-execution. 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.

**Races** If one of the consistent executions contains a race, 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 are race free. In particular, the actions \(d\) and \(i\) in Fig. 2 do not race with each other because \(e\) and \(h\) synchronise. Without the acquire fence this would not be true, and the program would be undefined.
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). Each of these also suggests a possible change to future versions of the C/C++11 model.

3.1 Disappearing synchronisation

Most synchronisation is immune to 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 Fig. 3.

Recall that a release sequence is defined as follows [6, §2.6]. It starts at a write-release, and extends to all stores of the same thread and all RMWs (potentially by other threads) that immediately follow in modification order, regardless of their memory order annotation. The point of this is to provide at the C/C++11 level more of the force of the memory barrier used on some architectures to implement the write-release, just before the write.

Such a release sequence can be broken by executing a new action, of which we give an example below. 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.

3. Incrementalising the axiomatic model: the problems

In this section we consider the challenges of developing an equivalent operational concurrency model that generates executions one action at a time. Recall that the axiomatic model takes complete pre-executions as input, so in order to state the equivalence the operational concurrency model also has to take complete pre-executions as input. In other words, it would incrementally add execution witness data (new $rf$-pairs, etc.) to a pre-execution given up-front. We call adding execution witness data between an action $a$ and actions previously considered committing action $a$.

Another notion that we use is that of following or respecting a certain order. If we would commit the actions of Fig. 4 (left side) 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. Or formally: let $com$ be the commitment order (that is, $(a, b) \in com$ if $a$ has been committed before $b$) and $r$ a relation, we say that we follow $r$ if for all $(a, b) \in com$ we have $(b, a) \notin r$.

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 conceptually very strange and it has undesirable consequences, which

![Figure 2: A consistent execution of the message passing program (see Fig. 1). Note that actions $e$ and $h$ synchronise.](image)

![Figure 3: Disappearing synchronisation](image)
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 Fig. 4 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.

![Figure 4: On the left a consistent execution with a cycle in \(rf \cup sb\) and on the right one with a cycle in \(mo \cup sb\).](image)

For the second, observe that the execution on the right of Fig. 4 has a cycle in \(mo \cup sb\). As described in the previous subsection, we follow \(mo\), so the existence of this cycle also shows that we cannot follow program order. Here the corresponding hardware examples, after applying the standard mapping, are not architecturally allowed or observed on ARMv8 (2+2W+STLs) or POWER (2+2W+lsyncs), so one might conceivably strengthen C/C++11 to similarly forbid this behaviour.

### 3.3 Abandoning sequential-consistent order

Recall from §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 Fig. 5 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 the \(mo \cup rf \cup sc\) cycle.

![Figure 5: A consistent execution with a cycle in \(mo \cup rf \cup sc\) (omitting initialisation).](image)

### 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 \(consistent\nonatomic\rf\) requires (in case \(r\) is non-atomic) that \(r\) reads from a write that happens before it. 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 and only if there does not exist a write that happens before it, so the same problems apply 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 Fig. 6 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 it in C/C++11 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 constrast 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 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 was 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 \( \text{get}_\text{rel}, \) so \( ex = (\text{pre}, \text{wit}, \text{get}_\text{rel}(\text{pre}, \text{wit})) \).

We use the notation \( \text{pre}\_sb \) and \( \text{wit}\_rf \) to refer to parts of pre-executions and execution witnesses. For brevity, we abuse this notation by writing \( \text{ex}\_sb \) when we should actually write “let \( \text{ex} = (\text{pre}, \text{wit}, \text{rel}) \), consider \( \text{pre}\_sb \)” and likewise for the parts of the witness (such as \( \text{ex}\_rf \)) and derived relations (such as \( \text{ex}\_hb \)).

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). We cannot prevent new synchronisation appearing between previously committed actions, but by following \( \{(a, b) \in \text{hb} \mid \text{is}\_load(b)\} \) we can prevent it between previously committed \( \text{writes} \) and \( \text{loads} \). This is enough to prevent the situation described in §3.4 regarding the predicates \( \text{consistent\_non\_atomic}\_rf \) and \( \text{det}\_read \).

This order satisfies all the properties we would need to incrementise 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 \( \text{hb} \) as we can. Because we cannot follow program order (see §3.2) we know that we cannot include all of \( \text{hb} \).

We decided to leave out \( \text{hb} \) edges to atomic writes, and include all \( \text{hb} \) edges to other types of actions. (For locks and unlocks there is a choice whether to include \( \text{hb} \) 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 \( \text{hb} \) order.

**Definition 5.1** (Commitment order). Let \( ex \) be a candidate execution. First define \( \text{ex}\_almost\_hb = \{(a, b) \in \text{ex}\_hb \mid \neg (\text{is}\_write(b) \land \text{is}\_atomic(b))\}. \) Then define \( \text{ex}\_\text{com} = (\text{ex}\_rf \cup \text{ex}\_mo \cup \text{ex}\_almost\_hb)^+ \).

**Theorem 5.2.** Let \( ex \) be consistent. Then the relation \( \text{ex}\_\text{com} \) 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 \( s\_\text{committed} \) denoting the actions that have been committed so far, and an execution witness \( s\_\text{wit} \) 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 \( \text{ex}\_\text{com} \) defined in the previous subsection. We want the states \( s_1, \ldots, s_n \) to reflect the witness build up so far, and an obvious thing to do is to define \( s_i\_\text{committed} \) to be the actions \( a_1, \ldots, a_i \) that are committed so far, and \( s_i\_\text{wit} \) as the restriction of \( ex\_\text{wit} \) to those actions. The initial state \( s_0 \) is always the same (regardless of the given pre-execution) because \( s_0\_\text{committed} = \emptyset \) and \( s_0\_\text{wit} \) the empty witness.
Definition 5.3. Let $pre$ be a pre-execution, and $S$ a set of actions. Then $\text{preRestrict}(pre, S)$ is defined by

$$\text{preRestrict}(pre, S).\text{actions} = pre.\text{actions} \cap S$$
$$\text{preRestrict}(pre, S).sb = pre.sb \cap S \times S$$
$$\text{preRestrict}(pre, S).asw = pre.asw \cap S \times S$$

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

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

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

$$pre' = \text{preRestrict}(pre, S)$$
$$wit' = \text{witRestrict}(wit, S)$$
$$\text{exRestrict}(ex, S) = \langle pre', wit', get_rel(pre', wit') \rangle$$

The partial executions $ex_i$ mentioned in the intro of this section are then given by $\text{exRestrict}(ex, A_i)$ 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 this: 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.

For example $\text{downclosed}(A, ex.mo)$ 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 $\text{downclosed}(A, ex.mo)$. Then $(\text{exRestrict}(ex, A)).hb \subseteq ex.hb$.

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 reads. We proved 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 $\text{downclosed}(A, ex.com)$. Then for all $(a, b) \in ex.hb$ with $b \in A$ and $b$ not an atomic write, we have that $(a, b) \in (\text{exRestrict}(ex, A)).hb$.

5.4 Consistency of prefixes

Now we know how $hb$ changes during incremental generation of executions, we can prove that the partial executions $\text{exRestrict}(ex, A_i)$ (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 $\text{downclosed}(A, ex.com)$. If $ex$ is a consistent execution, then $\text{exRestrict}(ex, A)$ is a consistent execution.

5.5 Transition relation

Given a consistent execution $ex$, an order $a_1, \ldots, a_n$, and the partial executions $ex_i = \text{exRestrict}(ex, \{a_1, \ldots, a_i\})$, 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 $\text{incrementalStep}(pre, s_1, s_2, a)$ 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.

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.

Definition 5.4. Let $r$ be a relation over actions, and $A$ a set of actions. Then $\text{downclosed}(A, r)$ holds if and only if for all $(a, b) \in r$ with $b \in A$ we have that $a \in A$. Figure 8: On the left $\text{exRestrict}(ex, \{a, b, d, e\})$ and on the right $\text{exRestrict}(ex, \{b, c, d, e\})$ where $ex_i$ and $ex_r$ are respectively the executions on the left and right of Fig. 4.
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 $s_0$ state 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. For the witness this is not immediately obvious, so 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 included in $mo$ 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 that adds an action $a$ at the end of the modification order of a state $s$.

Definition 6.1. Define $sameLocWrites(A, a)$ as \{$b \in A \mid is\_write(b) \land loc\_of(b) = loc\_of(a)$\}. 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. We explain the requirements afterwards.
Lemma 6.2. Let \( s \) be a state, \( ex \) an execution and \( a \) an action, for which the following holds.

\[
\begin{align*}
    a \notin s.committed \quad \text{(7)} \\
    ex.actions = s.committed \cup a \quad \text{(8)} \\
    \text{witRestrict}(ex.wit, s.committed) = s.wit \quad \text{(9)} \\
    \text{downclosed}(s.committed, ex.mo) \quad \text{(10)} \\
    \text{isConsistent}(ex) \quad \text{(11)}
\end{align*}
\]

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

The state \( s \) should be thought of as the current state, and \( ex \) as the execution we try to transition to. The requirements say that we should be able to transition to \( ex \): requirements (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 \( s \); requirement (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 \( \text{addToMo} \), 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 the first case, the consistency predicate \( \text{det_read} \) describes when there should be a new \( rf \) edge: if there exists a write that happens before \( a \) there should, otherwise there 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 \( \text{det_read} \) 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 \( \text{addToRfLoad}(a, s) \) 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 \( \text{sameLocWrites}(s, \text{committed}, a) \) for which we have \( \text{value_written_by}(w) = \text{value_read_by}(a) \) and returning \( s.wit.rf \cup \{(w, a)\} \).

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

Definition 6.4. Define \( \text{addToRfRmw}(a, s) \) as follows. If the set \( \text{sameLocWrites}(s, \text{committed}, a) \) is empty, return \( s.wit.rf \). Otherwise, there is a \( mo \)-maximal element \( w \) of that set. We check whether \( \text{value_written_by}(w) = \text{value_read_by}(a) \) 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. Note that the functions \( \text{addToRfLoad} \) and \( \text{addToRfRmw} \) are non-deterministic, so they return a set of possible new \( rf \) relations.

Lemma 6.5. Let \( s \) be a state, \( ex \) an execution and \( a \) an action for which \( a \notin s.committed \), \( ex.actions = s.committed \cup a \), \( \text{witRestrict}(ex.wit, s.committed) = s.wit \), \( \text{downclosed}(s.committed, ex.mo) \), and \( \text{isConsistent}(ex) \).

1. If \( a \) is a load, then \( ex.rf \in \text{addToRfLoad}(a, s) \).
2. If \( a \) is a RMW, then \( ex.rf \in \text{addToRfRmw}(a, s) \).
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 a 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 \( \text{addToSc} \) and \( \text{addToLo} \) 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 \( ex.sc \in \text{addToSc}(a, s) \) and otherwise we have \( ex.sc = s.wit.sc \); if \( a \) is a lock or an unlock, we have \( ex.lo \in \text{addToLo}(a, s) \) 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 \( \text{performAction}(s, a) \) that returns an execution witness based on the type of \( a \).

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

Definition 6.6. Define \( \text{executableStep}(pre, s) \) 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 \( \text{performAction}(s, a) \). Define the
new state \(s_2\) with \(s_2,\text{committed} = s,\text{committed} \cup \{a\}\) and \(s_2,\text{wit} = \text{wit}\). Finally, check whether our choice followed the commitment order and resulted in a consistent execution by discarding states that do not satisfy Requirement (5) or Requirement (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 \((s_2, a) \in \text{executableStep}(\text{pre}, s_1)\) if and only if \(\text{incrementalStep}(\text{pre}, s_1, s_2, a)\).

Define \(\text{executableTrace}\) and \(\text{executableConsistent}\) in the same way as in the incremental model (Definition 5.10), but then using \(\text{executableStep}\) instead of \(\text{incrementalStep}\). 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 \(ex = (\text{pre}, \text{wit}, \text{get.rel}(\text{pre}, \text{wit}))\). Then \(\text{executableConsistent}(ex)\) 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 does not need to 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 Fig. 1)
- **Load buffering:** a version with relaxed atomics (that allows the cycle given on the left of Fig. 4), 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 Fig. 4).**
- **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 only allowed outcomes). For the
relaxed version of LB and for the mo-sb-cycle program the outcomes with cycles in respectively rf ∪ sb and mo ∪ sb happened rarely; only approximately 1 out of a 1000 executions exhibited them. 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 Fig. 6 and imagine a non-atomic write w1 to a new location (say z1) that is sb-before action a, and similarly a new write w2 that is sb-before action k; and imagine a non-atomic read r1 of z1 that is sb-between actions d and e, and similarly a read r2 that is sb-between actions i and j. Suppose without loss of generality that when r1 is generated by the threadwise semantics, r2 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 w1 and r1 does not exist yet, and therefore we do not know where r1 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 nonatomic accesses, and with just a single kind of fence. They also identify that the execution presented in §3.2 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 Demskey [19]. This is focussed on efficiency, but attempts neither to be sound nor complete 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 ∪ 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. naïve 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 (∪) cycles in sc ∪ 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 isolation. Leaving such integration for future work may lead to a memory model that makes it remarkably involved. 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 (∪) cycles in sc ∪ 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 isolation. Leaving such integration for future work may lead to a memory model that makes it remarkably involved.

Acknowledgements We thank Mark Batty for discussions. 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


