DRAFT: An operational semantics for C/C++11 concurrency

Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell

University of Cambridge

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 discuss the issues and remaining challenges involved in integrating this semantics with an operational semantics for sequential C (described elsewhere).

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, 26, 25]. Batty et al. contributed to this standardisation process, resulting in a mathematical model in close correspondence with the standard prose [2].

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 [2, 5], and on the other hand, the design provides a flexible range of syn-

chronisation 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 [14, 23, 3, 22, 20, 16].

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` [2] 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 difficulty is that, in the above naive enumeration process, one has to construct candidates with no knowledge of whether the choice of control-flow unfolding and memory read values are actually compatible with the concurrency model; the vast majority of them will not be.

Given this, for programs that go beyond litmus tests, one would at least want to be able to explore single executions, e.g. for testing or animating a concurrent data-structure algorithm w.r.t. the relaxed-memory semantics. But the axiomatic model structure does not support the incremental construction of single executions: its consistency predicate is only defined over candidate

complete executions. For the same reason, it also does not support proofs by induction on paths, or analysis or model-checking tools that are closely based on the model.

This is the problem we address here: how one can incrementally construct executions of C/C++11 concurrent programs. Our main contribution is an operational semantics for C/C++11 concurrency which is proved equivalent to the axiomatic model of Batty et al. [2] and the ISO standard; our proof is mechanised in Isabelle/HOL.

The challenge arises from the fact that the axiomatic model (intentionally) 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 the transitions of our operational semantics, which to be complete w.r.t. the axiomatic model must be able to generate those cycles, cannot simply follow all the above relations. We show that for C/C++11 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.

We continue with a preliminary investigation into what is required to integrate our operational concurrency model with a semantics for the sequential aspects of a substantial fragment of C. That sequential semantics, defined by a typed elaboration into a Core language, will be described in detail elsewhere (it is not itself part of the contribution of this paper). The initial integration permits litmus tests in Core or in C to be executed and, more importantly, reveals several important areas for future work. This is a step towards tools that let one explore the behaviour of larger concurrent C11 programs, that use more C features than the original `cppmem` tool [2] – which had only a threadwise semantics only for a small ad hoc fragment of C, and which was limited to exhaustive enumeration of the behaviours of tiny test cases.

Contributions

- We show that one cannot build an equivalent operational model for C/C++11 that simply follows program order, SC order, or the synchronises-with order (§3).
- We show that the axiomatic model *does* behave incrementally under a particular execution order, develop an operational concurrency model following that order, and prove this model equivalent to the axiomatic model of Batty et al. [2], with a mechanised Isabelle/HOL proof (§4–6).
- We discuss the issues involved in integrating our operational concurrency model with a sequential operational semantics for a Core language into which a substantial fragment of C can be elaborated (§7).

We do all this for the full C/C++11 model as formalised by Batty et al. [2], including nonatomic accesses, all the atomic memory orders (sequentially consistent, release/acquire, release/consume, and relaxed), read-modify-write operations, locks, and fences.

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 [15] 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/~kn307/c11/esop2016/esop_2016_40_supplementary_material.tar` which builds with Isabelle 2015.

Non-goals While our operational semantics is executable, it is not intended to be a single-path or state-space exploration tool that is usable on industrial grade code, as we discuss in §7, though it may contribute to such tools in future. Rather, our contribution is the mathematical one: the operational model and its correctness theorem, and the demonstration that the model integrates better with the rest of the C/C++ semantics than the axiomatic model does. Focussing on the need for incremental construction of executions gives new insights into the internal structure of the C/C++11 model, which we hope will inform future language-level concurrency model design. The incremental structure may also be useful for metatheory proofs and analysis tools, as it is for conventional sequential or SC operational semantics.

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 [18].

Our operational 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.

Lastly, 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 one can expect an equivalent abstract-machine model to be feasible.

2 Background: C/C++11 axiomatic concurrency model

We begin by recalling the C/C++11 concurrency primitives and axiomatic model, referring to previous work [8, 2, 6] 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. Atomics can be annotated with different *memory orders*:

- 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 [13]: 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 [19].

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

2.2 The C/C++11 axiomatic concurrency semantics

Pre-executions To compute the behaviour of a progam using the axiomatic model, one first calculates the set of all *pre-executions* using a *threadwise semantics* (this is a parameter of the concurrency model, not a part of it). Each pre-execution corresponds to a particular complete control-flow unfolding of the

program and an arbitrary choice of the values read from memory, with the values written to memory as determined by the threadwise semantics.

Below we show an example program (in a condensed syntax, with some common initialisation at the top and then two parallel threads) and one of its many pre-executions. The pre-execution is represented as a graph, whose nodes are

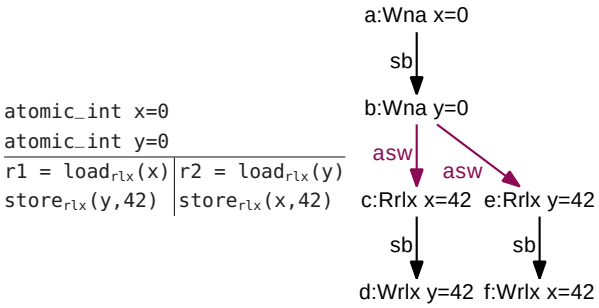


Fig. 1: Load buffering (LB)

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 (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.

To keep the diagrams simple we suppress the memory actions of thread-local variables ri. The *sb* "sequenced-before" edges capture program order, and the *asw* "additional synchronises-with" edges capture thread creation, both from the syntactic control-flow unfolding.

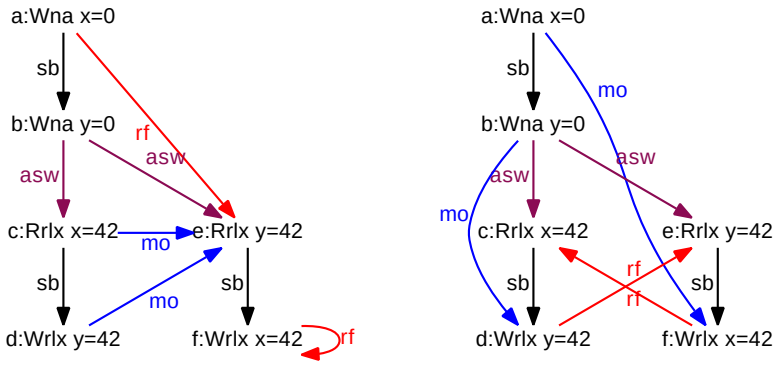
In general individual pre-executions may be infinitary, as may the set of all of them, but for programs without loops or recursion they will be finite, albeit perhaps extremely numerous. The threadwise semantics might calculate the set of all pre-executions of such programs inductively on program syntax (in that sense, this part of the semantics would be denotational, though it involves

no limit construction), or could involve exhaustive exploration of a threadwise labelled-transition operational semantics, with memory reads taking arbitrary values.

Execution witnesses For each pre-execution that has been computed, one enumerates all possible *execution witnesses*; a candidate execution is a pair of a pre-execution and an execution witness for it. An execution witness consists of the following relations over the actions of a pre-execution:

- The reads-from relation $\xrightarrow{rf}$ to relate each read to the write that it reads from.
- The coherence order $\xrightarrow{mo}$ is a total over atomic writes to the same location.
- The sequential consistent order $\xrightarrow{sc}$ is a total order over actions with a sequential consistent memory order.
- The lock order $\xrightarrow{lo}$ is a total order over locks and unlocks to the same location.

In Fig. 2 we see two witnesses over the pre-execution of Fig. 1. That on the left is not consistent: most of the $\xrightarrow{rf}$ and $\xrightarrow{mo}$ edges do not even relate events of the right kinds, and the reads-from edge from `a` to `e` relates events with different locations and values. It is the consistency predicate that imposes the intuitive meanings above, along with the more subtle properties that are the real substance of the C/C++11 model.



(a) An inconsistent witness (b) A consistent witness

Fig. 2: Execution witnesses over the pre-execution of Fig. 1

From a pre-execution and a witness the axiomatic model computes several relations that are referred to in the axioms. For example:

- The synchronises-with relation $\xrightarrow{sw}$ contains (among other things) synchronising unlock-lock pairs, and synchronising release-acquire pairs.
- The happens-before relation $\xrightarrow{hb}$ denotes which actions "happen before" which other actions.

The consistency predicate requires, among other things, that this derived happens-before relation is acyclic.

3 Incrementalising the axiomatic model: the problems

Recall that our ultimate goal is to incrementally generate executions in such a way that every consistent execution can be generated. In this section we consider a part of the problem, namely how to incrementally generate *witnesses*, given a complete pre-execution up-front, in such a way that all consistent witnesses over that pre-execution can be generated. We call a model that does that a *concurrency model*, in later sections we also construct the pre-execution incrementally. Our goal is to generate witnesses one action at the time: each step we add execution witness data (new rf-pairs, etc.) between a new action a and actions previously considered. We call such a step *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. 2b 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 we discuss in Section 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 Section 3.2 we show that we cannot follow sb (the program order), in Section 3.3 that we cannot follow sc (the sequential consistent order) and in Section 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 [2, §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.

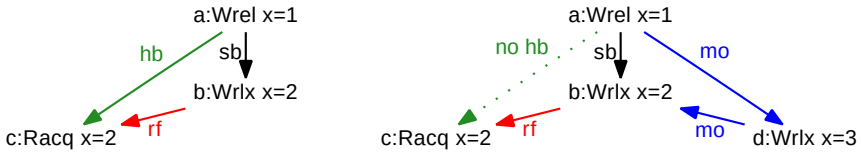


Fig. 3: Disappearing synchronisation

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 2.

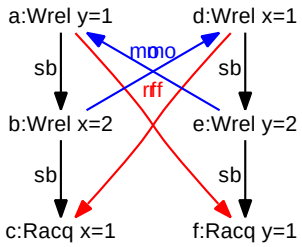
Another solution would be to change the axiomatic model (and the C/C++ ISO standards) by allowing the release sequence to extend to *sb*-later writes in the same thread irrespective of whether the write is immediately following in *mo* order. We believe that this matches hardware behaviour, so this change 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 in Fig. 2b (the well-known LB example) 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 below 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+lwsyncs), so one might conceivably strengthen C/C++11 to similarly forbid this behaviour.

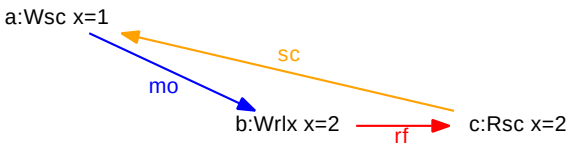


A consistent execution with a cycle in $mo \cup sb$

3.3 Abandoning sequential-consistent order

Recall from Section 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 [6, 4]). It is therefore surprising that the concurrency model cannot follow the sc relation when the SC memory order is mixed with other memory orders.

Our argument is as follows. The execution below 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.



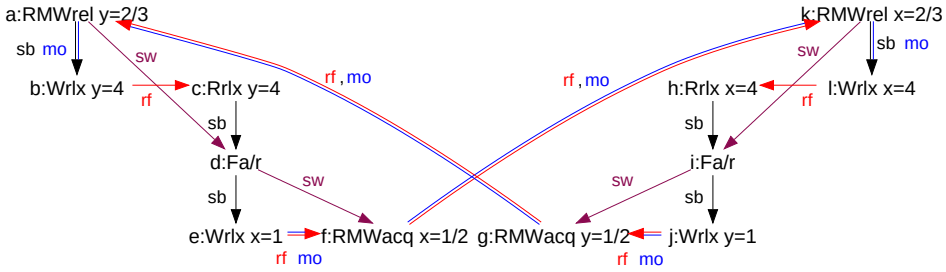
A consistent execution with a cycle in $mo \cup rf \cup sc$ (omitting initialisation)

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 (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 exists a write that happens before it, 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 below 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.



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

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 Section 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 Section 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 transtions.

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 2 and Theorem 3); we leave this, which is really 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 $3n$ 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 Section 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.

5 The incremental model

In the light of the non-approaches of Section 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 Section 5.1 and prove that it is acyclic, in contrast to the candidate orders considered in Section 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 Section 5.2. In Section 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 Section 3.1 and Section 3.4 we discussed why we need those properties).

Then in Section 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 Section 5.5. Finally, we define the incremental model in Section 5.6 and prove equivalence with the axiomatic model for finite executions.

Notation 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 $ex.rf$. With $get\_rel(pre, wit)$ we mean the list of relations that are calculated from the pre-execution and witness. Here we use the same shorthand: with $ex.hb$ we mean "let $ex = (pre, wit, rel)$ and $rel = [hb, \ldots]$, consider 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 $\{(a, b) \in hb \mid is\_load(b)\}$ we can prevent it between previously committed *writes* and *loads*. This is enough to prevent the situation described in Section 3.4 regarding the predicates consistent_non_atomic_rf and det_read.

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 Section 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 1 (Commitment order). *Let ex be a candidate execution. First define $ex.almost\_hb = \{(a, b) \in ex.hb \mid \neg (is\_write(b) \wedge is\_atomic(b))\}$.*

Then define the order

$$ex.com = (ex.rf \cup ex.mo \cup ex.almost\_hb)^+ .$$

Theorem 1. *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 we assumed that it was given and therefore we only need to incrementally build the witness.

Let ex be the execution that we want to generate incrementally, 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 build 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 (where $i = 0$) is always the same (regardless of the given pre-execution) because the set of committed actions is empty, and the witness contains only empty relations.

Definition 2. *Let pre be a pre-execution, and S a set of actions. Then $preRestrict(pre, S)$ is defined by*

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

Similary, 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.

5.3 Properties of happens before

In Section 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 3. *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 $\mathit{downclosed}(A, \mathit{ex}.\mathit{mo})$ 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 2. *Let ex be an execution (not necessarily consistent). Let A be a set of actions such that* $\mathit{downclosed}(A, \mathit{ex}.\mathit{mo})$. *Then*

$$(\mathit{exRestrict}(\mathit{ex}, A)).\mathit{hb} \subseteq \mathit{ex}.\mathit{hb}$$

Recall that in Section 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 3. *Let ex be a consistent execution. Let A be a set of actions such that* $\mathit{downclosed}(A, \mathit{ex}.\mathit{com})$. *Then for all* $(a, b) \in \mathit{ex}.\mathit{hb}$ *with* $b \in A$ *and b not an atomic write, we have that* $(a, b) \in (\mathit{exRestrict}(\mathit{ex}, A)).\mathit{hb}$.

5.4 Consistency of prefixes

Now we know how hb changes during incremental generation of executions, we can prove that the partial executions $\mathit{exRestrict}(\mathit{ex}, A_i)$ (as defined in Section 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 4. *Let A be a set of actions such that* $\mathit{downclosed}(A, \mathit{ex}.\mathit{com})$. *If ex is a consistent execution, then* $\mathit{exRestrict}(\mathit{ex}, A)$ *is a consistent execution.*

5.5 Transition relation

Given a consistent execution ex, an order $a_1, \ldots, a_n$, and the partial executions $\mathit{ex}_i = \mathit{exRestrict}(\mathit{ex}, \{a_1, \ldots, a_i\})$, 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 $\mathit{incrementalStep}(\mathit{pre}, s_1, s_2, a)$ 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 4. *The relation incrementalStep*(pre, s_1, s_2, a) *is defined as*

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

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) ensure 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.

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. *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*, such that*

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

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. *Let $ex = (pre, wit, rel)$ be a candidate execution. We define*

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

Theorem 6 (Equivalence). *Let ex be a candidate execution with*

$$ex = (pre, wit, \mathit{get\_rel}(pre, wit)).$$

Then $\mathit{incrementalConsistent}(ex)$ holds if and only if ex is finite and 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 $\mathit{incrementalStep}(pre, s_i, s_{i+1}, a_i)$ 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 $\mathit{incrementalStep}(pre, s_i, s_{i+1}, a_i)$ holds. For the set of committed actions this is clear: $s_{i+1}.\mathit{committed} = s_i.\mathit{committed} \cup \{a\}$ since this is directly required by *incrementalStep*. For the witness this is not immediately obvious, so investigate this in the following sections: in Section 6.1 we consider the *mo* relation, in Section 6.2 the *rf* relation, and in Section 6.3 the *sc* and *lo* relations. Then in Section 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 the previous formally, we define a function that adds an action a at the end of the modificition order of a state s.

Definition 6. *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. We explain the requirements afterwards.

Lemma 1. *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* $ex.mo = addToMo(ex.pre, a, s)$ *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 $addToMo$, 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 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 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.

Note that in the formal definition we use a non-deterministic monad every time we say "non-deterministically choose or pick". With this monad we can later randomly or exhaustively explore the possibilities.

Definition 7. *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_atomcity* 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 8. *Define* $addToRfRmw(a, s)$ *as follows. If the set* $sameLocWrites(s.committed, a)$ *is empty, return* $s.wit.rf$. *Otherwise, there is a 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 1. Note that the functions *addToRfLoad* and *addToRfRmw* are non-deterministic, so they return a set of possible new rf relations.

Lemma 2. *Let* s *be a state,* ex *an execution and* a *an action for which* $a \notin s.committed$, $ex.actions = s.committed \cup a$, $witRestrict(ex.wit, s.committed) = s.wit$, $downclosed(s.committed, ex.mo)$, $downclosed(s.committed, ex.rf)$, *and* $isConsistent(ex)$.
(1) If a *is a load, we have* $ex.rf \in addToRfLoad(ex.pre, a, s)$.
(2) If a *is a RMW, we have* $ex.rf \in addToRfRmw(ex.pre, a, s)$.
(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 *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 the usual lemmas that show that this construction suffices. For the explanation of the assumptions we refer to the explanation given after Lemma 1.

Lemma 3. *Let* s *be a state,* ex *an execution and* a *an action for which* $a \notin s.committed$, $ex.actions = s.committed \cup a$, $witRestrict(ex.wit, s.committed) = s.wit$, *and* $isConsistent(ex)$.

If a has a sequential consistent memory order (which is possible for loads, stores, RMWs and fences), we have $ex.sc \in addToSc(ex.pre, a, s)$ and otherwise we have $ex.sc = s.wit.sc$.

If a is a lock or an unlock, we have $ex.lo \in addToLo(ex.pre, a, s)$ and otherwise we have $ex.lo = s.wit.lo$.

6.4 The transition function

With the results of Section 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. We have summarised this in the table below, defining $performAction(s, a)$ by describing how each of the relations of $s.wit$ change based on the type of a.

	mo	*rf*	*sc*	*lo*
Loads	Unchanged	*addToRfLoad*	If memory order is SC then *addToSc* else unchanged	Unchanged
Stores	If non-atomic then unchanged else *addToMo*	Unchanged	Same as loads	Unchanged
RMWs	*addToMo*	*addToRfRmw*	Same as loads	Unchanged
Locks, unlocks	Unchanged	Unchanged	Unchanged	*addToLo*
Fences	Unchanged	Unchanged	Same as loads	Unchanged

Now we define the transition function.

Definition 9. *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 an consistent execution by discarding states that do not satisfy Requirement (5) *or Requirement* (6) *of Definition 4. For each of the non-discarded options, the function returns the pair (s_2, a).*

Theorem 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), but then using *executableStep* instead of *incrementalStep*. From the previous theorem and from Theorem 6 it then follows that the executable model is equivalent to the axiomatic model for finite executions:

Corollary 1. *Let ex be a candidate execution with $ex = (pre, wit, get\_rel(pre, wit))$. Then $executableConsistent(ex)$ holds if and only if ex is finite and 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 to give semantics of the C programming language; 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 Section 7.1. The symbolic execution has significant drawbacks and one might hope that it is only needed for atomics, but in Section 7.2 we show that it is also necessary for non-atomics. Then in Section 7.3 what the implementation of the combined semantics supports and 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, all within a nondeterminism monad.

The Core dynamics is a step function: from a given Core program state it returns the set of memory operations (and the resulting Core program state should that operation be performed) that can be performed at this point by the program. These operations (object creation, load, store) 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, for each load operation, 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 (Core has a single if-then-else construct) 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 symbolic execution can also be used to compute complete pre-executions, e.g. to test variants of the axiomatic model, by executing a program completely symbolically, without any steps of the concurrency semantics. In this mode symbolic values cannot be resolved in the course of the execution, obviously.

7.2 Symbolic execution unavoidable for non-atomics

Symbolic execution has significant downsides here: some paths are followed that later turn out to be inconsistent, and 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 of §3.4, 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 Section 3.4) and the threadwise semantics has to use a symbol as its return value.

7.3 Integration: implementation and outstanding issues

The correctness of the concurrency model is guaranteed by the equivalence theorem. Our semantics is also executable, and we have exercised it on some classic litmus test programs. It can be run in two modes: pseudorandom mode, exploring a single execution path, with the concurrency model and threadwise semantics tightly interleaved, and an exhaustive mode that calculates all pre-executions up-front. In principle one could also do an exhaustive search of the tightly interleaved semantics, but we expect the combinatorics would be prohibitive. Each test can be written in multiple forms, of increasing complexity: the pseudocode

one typically sees in papers, as in Fig. 1; hand-written Core, which makes intra-thread sequencing and variable creation explicit; C extended with explicit parallel composition, adding memory actions for thread-local accesses; and actual C, adding explicit pthread thread creation and join. For example, running a release-acquire message-passing test exhaustively (MP+na+rel+acq+na, shown in the first three forms in Fig. 4), the Core version has 1350 executions, while the C-with-explicit-parallel version has 8451, taking 0.2s and 25s respectively. The performance advantage of the former arises from the fact that in a hand-written Core test one can use pure value `let`s that do not give rise to memory actions, while in C one cannot. Finding a single execution in random mode takes negligible time (0.02s and 0.05s), it usually results in a state with a satisfiable constraint, and it does sometimes result in the relaxed-behaviour outcome. For these and the other tests we tried (store buffering, load buffering, and cycles in $mo \cup sb$), random mode always returned allowed outcomes and exhaustive mode returned the set of *all* allowed outcomes.

Extending this 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. [2], 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. [11], Owens et al. [17], 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 only closely related work that we are aware of is the work in press by Lahav et al. [12], that we were made aware of while preparing this submission. 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 $hb \cup mo$ 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,

```
int x=0
atomic_int y=0
-----------------------------
x = 1          | r1 = load_acq(y)
store_rel(y,1) | r2 = x
```

(a) The release-acquire message-passing test, MP+na+rel+acq+na, in pseudocode

```
proc main () : eff integer :=
  let strong x = create(<alignof>("signed␣int"), "signed␣int") in
  let strong _ = store("signed␣int", x, 0) in
  let strong y = create(<alignof>("_Atomic(signed␣int)"), "_Atomic(signed␣int)") in
  let strong _ = store("signed␣int", y, 0) in
  let strong (_, a2) =
    par(
       let strong _ = store("signed␣int", x, 1) in
       let strong _ = store("_Atomic(signed␣int)", y, 1, release) in
       return (unit)
       end end
       ,
       let strong a1 = load("_Atomic(signed␣int)", y, acquire) in
       if a1 = 1 then
         let strong ret = load("signed␣int", x) in
         return(ret)
         end
       else
         return (2)
       end end) in
  let strong _ = kill(x) in
  let strong _ = kill(y) in
  return(a2)
  end end end end end end end
```

(b) The MP+na+rel+acq+na test in Core, making object lifetime explicit with create and kill, and sequencing explicit with let strong, but using Core value lets to record the results of memory loads.

```
#include <stdatomic.h>
int main(void) {
  int x = 0;
  _Atomic int y = 0;
  int z1, z2;
  {{{ { x = 1;
        atomic_store_explicit(&y, 1, memory_order_release); }
  ||| { z1 = atomic_load_explicit(&y, memory_order_acquire);
        if (z1 == 1)
          z2 = x;
        else
          z2 = 2; }  }}};
  return z2;
}
```

(c) The MP+na+rel+acq+na test in C extended with an explicit parallel composition ({{{·|||·}}}). This version creates memory actions for the accesses to `z1` and `z2`, but the explicit parallel avoids the extra memory actions from pthread-style thread creation.

Fig. 4: The MP+na+rel+acq+na litmus test in three forms

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 other most closely related work we are aware of is the model-checker of Norris and Demsky [16]. 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. [24], and the K semantics of Ellison [10, §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.

Finally, 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. [21]. 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.

9 Conclusion

We have presented an operational concurrency model that covers the full formalisation [2] 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 explored preliminary integrated of the concurrency model with a sequential operational semantics for a Core language into which a substantial fragment of C can be elaborated.

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 (Section 3.3), cycles in $sw \cup rf$ (Section 3.4) and we suggested changing the definition of release-sequences (Section 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 integratable (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. 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.

Bibliography

[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] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proc. POPL*, 2011.

[3] Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 235–248, New York, NY, USA, 2013. ACM.

[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. Springer, 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 *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 509–520, New York, NY, USA, 2012. ACM.

[6] Mark John 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`.

[7] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. Nitpicking C++ concurrency. In Peter Schneider-Kamp and Michael Hanus, editors, *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, pages 113–124. ACM, 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, Berlin, Heidelberg, 2007. Springer-Verlag.

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

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

[12] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency, 2016. To appear in POPL 2016.

[13] 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.

[14] R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proc. PLDI*, 2013.

[15] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In *Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 175–188, 2014.

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

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

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

[19] 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.

[20] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 110–120, 2015.

[21] Emina Torlak, Mandana Vaziri, and Julian Dolby. Memsat: Checking axiomatic specifications of memory models. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 341–350, New York, NY, USA, 2010. ACM.

[22] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 691–707, 2014.

[23] 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 *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 209–220. ACM, 2015.

[24] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), June 2013.

[25] WG14. ISO/IEC 14882:2011.

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