# N4136 – C Concurrency Challenges Draft 2014-10-13

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon, Peter Sewell

This document was presented for discussion at the Redmond SG1 meeting on 2014-09-05. It is the draft of an academic paper whose examples raise issues with the C and C++ memory models.

# C Concurrency Challenges

Mark Batty Kayvan Memarian Kyndylan Nienhuis Jean Pichon Peter Sewell
University of Cambridge
first.last@cl.cam.ac.uk

## Abstract

In this paper we investigate the concurrency semantics of C and C++. We first give two positive results: we show that (for strongly finite programs) the existing axiomatic model for C/C++11 satisfies one of the core design goals, guaranteeing sequentially consistent semantics for race-free programs that do not use low-level atomics, and we develop an equivalent executable operational model. These results are mechanised, in HOL4 and Isabelle respectively.

We then investigate the longstanding problem of *thin-air executions*, which are known to be a challenge for the semantics of high-performance relaxed atomics. We first give a concise description of the problem and show that it cannot be solved (without restricting current compiler optimisations) with any per-candidate-execution condition. We then show that the problem essentially recurs when one attempts to integrate the concurrency model with more of C, mixing atomic and nonatomic accesses; it is not confined to programs that use relaxed atomics. We explore how far one can go with a semantics based on an explicit operational construction of out-of-order execution; this gives the desired behaviour for thin-air examples but exposes further difficulties.

We conclude by summarising the possible alternatives for the semantics of C and C++. The remarkable and disturbing fact is that, currently, there is no really satisfactory proposal for the semantics of any general-purpose shared-memory concurrent programming language.

## 1. Introduction

***Context*** The design of satisfactory semantics for shared-memory concurrent programming languages is a long-standing problem that is still not fully understood. The basic tension is between implementability and usability: on the one hand, such a semantics must admit the relaxed-memory behaviours that are permitted by multiprocessor architectures, and those that are introduced by compiler optimisations, otherwise it would not be efficiently implementable, but on the other hand, it must provide sufficiently strong guarantees for concurrent algorithms to work correctly. Then there are other important desiderata: the semantics must be mathematically rigorous, as this is an area where informal reasoning is particularly error-prone; it should be as intuitive as possible (though there will inevitably be subtleties); it should support testing of implementations and of concurrent algorithms, and it should support compositional reasoning.

There have been two major attempts to develop concurrency semantics for mainstream languages, for Java and C/C++. For Java, the original language specification [16] was shown by Pugh [25] to be flawed in both directions: too strong to be implementable and too weak for some concurrent programming idioms. A new specification [20] was developed in JSR-133, and incorporated into Java 5.0, but that too has been shown to be unsound with respect to standard compiler optimisations, by Cenciarelli et al. [12] and Ševčík and Aspinall [29]. This remains unresolved.

For C and C++, an effort as part of the C++0X standardisation process led to a specification incorporated into the C++11 and C11 standards [1, 6]. The basic design was outlined by Boehm and Adve [9], and Batty et al. [5] developed a formal semantics in the latter stages of the standardisation process, identifying various flaws in the draft standard and feeding back into the ratified standards and later defect reports. C/C++11 concurrency has been supported by GCC and Clang since versions 4.9 and 3.2 respectively, and the model by Batty et al. has been used for many purposes, including correctness proofs for compilation schemes to x86, by Batty et al. [5], and to IBM Power, by Batty et al. [4] and Sarkar et al. [27]; compiler testing via a theory of sound optimisations, by Morisset et al. [22]; model checking, by Norris and Demsky [24]; compositional library abstraction, by Batty et al. [3]; and program logics, by Vafeiadis and Narayan [31] and by Turon et al. [30]. Elements of the C/C++11 concurrency model have also been incorporated into OpenCL 2.0.

But despite all this, there remain major open questions, about the metatheory of the model and about what the semantics should be for "thin-air" executions. We resolve two of the former and discuss the latter, giving some precise constraints on possible solutions and demonstrating that it is more important and more challenging than previously recognised.

***Contributions*** We make five contributions. First (§2), we describe a machine-checked proof, in HOL4 [17], that (for programs without loops or recursion) the model of Batty et al. satisfies one of the core design goals for C/C++11 concurrency: programs that do not use the low-level atomics of the language, and that are race-free in a sequentially consistent (SC) semantics, only exhibit sequentially consistent behaviour. This *DRF-SC* property gives a relatively simple semantics for programmers using that fragment of the language. Batty et al. observed that this property was false in early drafts of the standard [5], and the description of the atomics in later drafts changed as a result [13, 21].

Second (§3), we give a provably equivalent *operational* model. The standard and the earlier mathematical model [5] are *axiomatic* memory models (not to be confused with the traditional program-logic notion of axiomatic semantics). In such a semantics one first calculates the set of all *candidate executions* of a program: the sets of memory access actions (and various partial orders over them) that could be obtained in a complete run in which the values read from memory are unconstrained. The memory model filters the candidate executions, defining the *consistent executions*, and defines which of those are *race-free*. The semantics of a program is either the set of all consistent executions, if they are all race-free, or undefined, otherwise. This style of semantics is both good and bad. For very small litmus-test programs one can execute the semantics to calculate the set of all their consistent executions, and to identify any races, but it does not support exploring (interactively or randomly) *some* executions of a larger program. The consistency and race predicates are in some sense mathematically simple (albeit intricate), expressed just with propositional logic and quan-

tification over the actions in a complete candidate execution, but this is very different to conventional operational or denotational semantics, and one cannot reason about it using simple inductions on trace length, operational derivation, or program syntax. We present an executable operational model (covering C/C++11 memory accesses, fences, locks, and all memory orders except consume) that (for finite executions) is proved equivalent to the formal semantics of Batty et al., with a machine-checked proof in Isabelle/HOL [18]. A key property of this operational model is that it must execute actions out of program order, but must be integrated with a thread-local semantics that does respect program order.

Third (§4), we consider *thin-air* reads. This is a long-standing open problem in the design of the semantics for C/C++11 relaxed atomics: accesses for which races are permitted but where one does not wish to pay the cost of any barriers or other hardware instructions beyond normal reads and writes. The question is how one can define an envelope that permits current compiler optimisations and hardware behaviour, while excluding particular example executions that it is agreed should be forbidden: those with self-satisfying conditional cycles or values appearing out of thin air (this is also closely related to the difficulties with Java). Here we give an instructive negative result: thin-air executions cannot be forbidden in a per-execution way, by any adaptation of the C/C++11 definition of consistent execution that uses the same notion of candidate execution.

Fourth (§5), we identify a new problem that arises when one tries to integrate C/C++11 concurrency with a semantics for more of the C language. Thin-air executions have been thought to be a problem only for programs using the relaxed atomics (intended only for expert use) of C/C++11, but that turns out not to be the case. The model of Batty et al. presupposes an up-front distinction between atomic and non-atomic locations, but that is not present in C, where (for example) one should be able to reuse `malloc`'d regions to store atomics and then nonatomics, or use `char` pointers to read the representation bytes of an atomic. We show that the thin-air problem essentially recurs in this setting, even in the absence of relaxed atomics.

Finally (§6) we explore an out-of-order operational semantics construction; this gives the desired behaviour for the thin-air examples of §4 but highlights other difficulties.

We conclude (§7) by summarising possible approaches for the semantics of a shared-memory concurrent language. Details of our formal developments are available in the supplementary material, including definitions in Lem [23] and HOL4 and Isabelle proof scripts. We introduce aspects of the C/C++11 model as required, but for a full description we refer to [5].

## 2. DRF-SC: sequential consistency for race-free programs

DRF-SC presents a simple concurrency model to programmers that use a fragment of C/C++11: if a program uses no low-level atomics, and all executions of the program in an SC semantics are race-free, then the program exhibits no relaxed behaviour. The property was introduced simultaneously by Adve and Hill [2] and Gharachorloo et al. [15], and is explicitly described by Boehm and Adve [9] and the C11 and C++11 standards [1, §5.1.2.4p26], [6, §1.10p21]. Here, we formally establish DRF-SC for the full C/C++11 concurrency model [5], mechanised in HOL4. Previous results along similar lines have been given by Boehm and Adve [9], with a hand proof for a preliminary model, and Batty et al. [4, Thm. 5], with a hand proof based on an earlier version of Batty et al.'s formal model, and using that model's notion of races for the SC semantics rather than the more straightforward SC notion of race based on identifying two conflicting adjacent actions that we use here. The latter exposes the user of the SC model to the relaxed model in the calculation of races, negating much of the simplification aimed for by DRF-SC. The intricacy of the semantics and significance of the result make this a prime target for mechanised proof, to give high confidence in the result. The proof script (approx. 23k lines, including additional model equivalence results) is included in the supplementary material.

To state DRF-SC, we first define a memory model for C/C++ executions, the *total model*, that is manifestly sequentially consistent. While in C/C++11 candidate executions describe the dynamic behaviour of memory with many partial orders (modification order, lock order and SC order), the total model has only a single total order over all memory accesses in the pre-execution. Reads must read from the immediately preceding write to the same location in the total order, and two accesses race if they access the same location, at least one is a write, they are not both atomic, and they are adjacent in the total order.

The theorem requires that the program ensures that atomic initialisation happens before all atomic accesses for each location. To simplify the proof, we also restrict its statement to programs that satisfy a strong finiteness condition: there must be a finite bound on the size of the pre-executions allowed by the threadwise semantics (this lets us use a simple form of induction). This means it does not apply to programs with recursion or loops. However, intuitively those are orthogonal to the concurrency semantics; we do not know of any reason why including them might affect the truth of the theorem.

**Theorem 1.** *For programs whose pre-executions (i) use only mutex, non-atomic and SC-atomic accesses, (ii) have atomic initialisations ordered by sequenced-before and parent-to-child thread synchronisation before all atomic accesses at the same location, and (iii) are bounded in size by some $N$, either both the C/C++11 model and the total model give undefined behaviour, or the sets of consistent executions in each, projected down to the pre-execution and the reads-from relation, are equal.*

PROOF OUTLINE The proof first involves several steps of simplifying the C/C++11 model for programs that do not use low-level atomics. The remaining proof can be split into one part for race-free programs and another for racy ones. For race-free programs there are two cases. Given a consistent execution in the C/C++11 model, we must construct a consistent execution in the total model with the same pre-execution and reads-from relation. The union of happens-before and SC order is acyclic, so we extend this to a total order and show that that is consistent according to the total model. In the other direction, given a consistent execution in the total model, we project partial relations from the total relation that serve as modification order, SC order and lock order in a C/C++11 candidate execution, and then show that it is consistent.

Given a racy execution in one model, we construct a (potentially different) racy execution in the other. In this part of the proof, we rely on several definitions and an assumption about the thread-local semantics. We define a *prefix* as a part of an execution where every sequenced-before or thread-synchronisation predecessor of any action within the part is also in the part. A *fringe action* of a prefix is an action that is not in the prefix, but is an immediate sequenced-before or thread-synchronisation successor of an action in the prefix. We must assume that the thread-local semantics is *receptive*: for any read or lock in the fringe of a prefix of a pre-execution, allowed by the thread-local semantics, and for every other value or lock outcome, there exists a pre-execution with the same prefix, but where the fringe action is changed accordingly.

Given a racy execution in the total model, we find the first race according to the total relation, and take the prefix made up of all predecessors of the later action in this race. The prefix is

consistent and race free, so we can translate it to a consistent prefix in the C/C++11 memory model with the same set of fringe actions. We extend this to a consistent prefix containing the racy action in the total model, appealing to receptiveness to change its value (if necessary for consistency), and we show that there is a race in the extended prefix. We then induct on the size of the prefix to show that for each larger finite prefix size, $n$, either there exists a racy consistent execution, or a racy consistent prefix with at least $n$ actions. Finally, we appeal to the boundedness of executions to establish that there is a racy consistent execution of the program in the C/C++11 memory model.

Given a racy execution in the C/C++11 model, the steps involved in the proof are similar, but finding the first race differs. For each race in the execution, we identify the set containing the racy actions and all of their happens-before predecessors. The execution is finite, so the set of all such sets is finite, and the subset relation is acyclic over them, so we can find a subset-minimal set made up of a pair of racing actions and their happens-before predecessors. We identify one of the racy actions and the happens-before predecessors of both as a race-free prefix. This prefix is consistent, so we can translate it to a consistent prefix in the total model. We then add the previously-racy fringe action to the prefix, and establish that it is consistent and racy, appealing to receptiveness, if necessary for consistency. In a similar fashion to the previous case, we complete the consistent racy prefix to get a consistent racy execution in the total model.

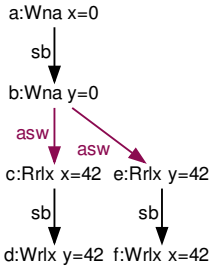
## 3. An operational C/C++11 concurrency model

C/C++11's axiomatic concurrency model [5] does not support incremental execution, making it difficult to explore execution paths of nontrivial programs. This section introduces an equivalent executable operational semantics, covering all features of the axiomatic model except the consume memory order. This is mechanised in Isabelle/HOL; the definitions and proof script (approx. 5k lines) is included in the supplementary material). To see why the axiomatic concurrency model makes it difficult to explore execution paths, we first recall its structure.

### 3.1 Summary of the axiomatic concurrency model

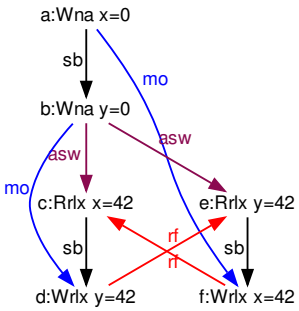
To compute the behaviour of a progam using the axiomatic model, one first calculates the set of all *pre-executions* using a *threadwise semantics* (this is a parameter of the concurrency model, not a part of it). Each pre-execution corresponds to a particular complete control-flow unfolding of the program and an arbitrary choice of the values read from memory, with the values written to memory as determined by the threadwise semantics. The threadwise semantics might calculate the set of all pre-executions inductively on program syntax (in that sense, this part of the semantics is denotational, though it involves no limit construction). Below we show an example program and one of its many pre-executions.

```
atomic_int x=0
atomic_int y=0
r1 = load_rlx(x)  | r2 = load_rlx(y)
store_rlx(y,42)   | store_rlx(x,42)
```




A pre-execution consists of a set of *memory actions*, which in the example are all writes (W) or reads (R), and some relations over them: program order ($sb$, "sequenced-before") and thread creation ($asw$, "additional synchronises-with"). Reads and writes with subscript NA are non-atomic reads and writes; others are atomic and the subscript specifies the *memory order* (the synchronisation strength of the action, not an order relation): SC, release, acquire, consume, or relaxed. Write and read actions both include an address and value. To keep the diagrams simple we suppress the memory actions of thread-local variables `ri`. Threads are separated by lines.

Then for each pre-execution, one enumerates all possible *execution witnesses*, each of which consists of a *reads-from* relation ($rf$) over the actions that relates a read to the write that it read from, a *modification order* ($mo$) which is a total order over atomic writes to the same location, an SC order, and a lock order. A *candidate execution* is a pair of a pre-execution and one of its execution witnesses. For example, here is one of the candidate executions for the above program.




The axiomatic model defines predicates $is\_consistent$ and $is\_undefined$ over candidate executions. If one of the consistent candidate executions is undefined (e.g. if it contains a race), then the program as a whole is undefined. Otherwise, the behaviour is the set of all consistent executions. The example on the left is consistent, and in fact no consistent execution of this program is undefined, so the program has well-defined behaviour including this execution (this execution is allowed on Power and ARM, so it must be allowed in the language).

This structure makes the algorithmic difficulty of exploring single executions apparent: the calculation of pre-executions and enumeration of execution witnesses takes place before one has any constraint on the values read from memory, so there is no knowledge about which paths are feasible. Moreover, for programs with any loops or recursion, the set of pre-executions will be infinite, and the action sets of some of them will also be infinite.

### 3.2 Challenges

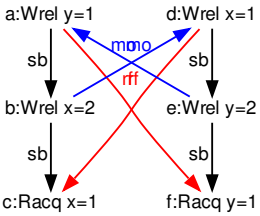
An operational semantics should build executions incrementally. The immediate challenge is that consistency is defined only for complete candidate executions and is not closed under $sb$-prefixes: the execution above is consistent, but if action `d` or `f` is removed, then there is no execution witness that makes the pre-execution consistent. To construct the execution while being consistent at every step, the semantics would have to execute the actions `c`–`f` in one step, but shapes like `c`–`f` can be arbitrarily large, both in the number of threads and the intervening actions on each thread, so this would defeat the operational goal. Instead, we execute one action per step and we allow executions to be temporarily inconsistent, proving that consistency is regained at termination (see §3.4).

To be fully operational, we need both an operational threadwise semantics (in contrast to that assumed by [5]) and an operational concurrency model, linked together. A standard operational threadwise semantics generates pre-executions incrementally, in program order, and a simple concurrency model can follow suit, building corresponding (consistent) execution witnesses incrementally. For example, a sequentially consistent concurrency model can provide the value for a read request (and commit the read) based only on the actions it has seen before in program order. But for C/C++11 this is impossible, for two reasons.

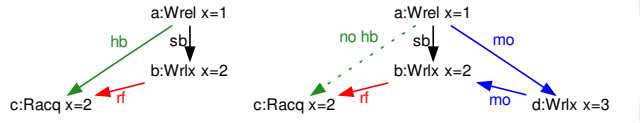
First, if the threadwise semantics and the concurrency model were to execute together in progam order, it would have the undesirable consequence that established synchronisation could disappear when executing a new action (happens-before would not grow monotonically as the operational model takes transitions). To see this, first consider the program below and one of its consistent executions. There is a cycle in $mo \cup sb$, so if the concurrency model executed actions in program order, it would have to insert new actions in modification order before actions that are already executed. While this is counterintuitive, it is not a problem in itself.

```
atomic_int x=0;
atomic_int y=0;
```

| | |
|---|---|
| $\text{store}_{\text{rlx}}$(x,1) | $\text{store}_{\text{rlx}}$(y,1) |
| $\text{store}_{\text{rlx}}$(y,2) | $\text{store}_{\text{rlx}}$(x,2) |
| r1 = $\text{load}_{\text{rlx}}$(y) | r2 = $\text{load}_{\text{rlx}}$(x) |




However, in the following example, inserting actions in the middle of $mo$ causes synchronisation to disappear. Synchronisation is expressed in the axiomatic model with the $hb$ (*happens-before*) relation, defined using the data of a candidate execution. In the first execution, the writes $a$ and $b$ are part of a release sequence [5, §2.6], 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.




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 from which writes 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 $hb$ 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.

We resolve this by committing writes in modification order, as we describe in §3.4. The example with the cycle in $mo \cup sb$ then shows that we cannot always commit writes in program order.

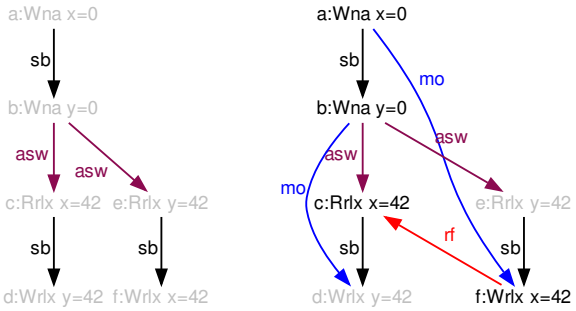
For the second reason that the concurrency model cannot always follow program order, when committing a read the concurrency model must determine which write the read reads from (recall that the reads-from ($rf$) relation is part of the execution witness), but in C/C++11 there can be cycles in $sb \cup rf$, as we saw in §3.1. So if the concurrency model were to follow program order, it would have to let the read read from a write that has not been reached yet in order to be complete.

We resolve this by making the threadwise semantics symbolic, so that it can continue after such a read (see §3.6) and by letting the concurrency model defer commitment of a read (and the choice of which write it reads from) until later.

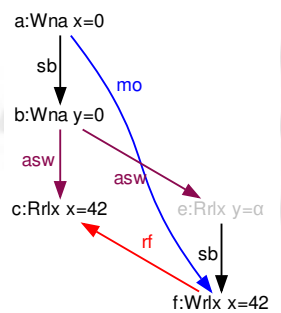
### 3.3 Overview

We deal with these challenges in three stages. In the first stage (§3.4) we construct an *incremental concurrency model*.

For this stage we assume (as in the axiomatic concurrency model) an axiomatic threadwise semantics that provides it with a complete pre-execution; we define a transition relation that incrementally commits the actions of that pre-execution and constructs (partial) execution witness data for them, in an order consistent with the constraints we saw above. A state consists of the subset of actions of the pre-execution that the incremental concurrency model has *committed*, and the (partial) execution witness that the incremental concurrency model has constructed so far. In the figure below on the left, we see the initial state: it consists of the complete pre-execution generated by the axiomatic threadwise semantics, where none of the actions have been committed (denoted by their grey colour) and without any execution witness relations. In the figure below on the right, we see a state where actions $a$, $b$, $c$ and $f$ have been committed but $d$ and $e$ have not.




The transition relation of this model is still defined in an axiomatic style: to compute the set of possible transitions, one has to generate all states and filter them with the transition predicate, which is infeasible for anything but tiny pre-executions. In the second stage (§3.5) we solve this by constructing a more operational model that explicitly defines how each relation in the partially generated execution witness can change for each type of action (in this stage we continue to assume an axiomatic threadwise semantics that generates complete pre-executions). The state type is the same as in the incremental concurrency model, and we prove that the former can make a transition if and only if the latter can do that.




In the third and last stage (§3.6), we integrate the operational concurrency model with an operational threadwise semantics that generates pre-executions action by action. Because the operational concurrency model might not immediately commit actions that the operational threadwise semantics executed, we made the operational threadwise semantics symbolic: in the case of uncommitted reads we use a fresh symbol as its return value. This is the case in the partial execution above, for example, where the threadwise semantics has had to continue past an as-yet-uncommitted read $e$. Whenever the concurrency semantics commits a read, the symbol gets resolved.

We flesh these out in the remainder of this section (again referring to the supplementary material for the full definitions). This necessarily involves more technical details of the C/C++11 ax-

iomatic model, but these subsections are not necessary for the later §4–7.

### 3.4 The incremental concurrency model

The incremental concurrency model commits the actions of a complete pre-execution one by one, constructing execution witness data (nondeterministically) along the way.

To show that it can generate all finite consistent executions, we have to show that for each finite consistent execution the model can take a path from the its initial state (consisting of the pre-execution where none of the actions have been committed), to a state containing the entire execution. Since actions are committed one by one, this path amounts to the order in which the actions of the execution are committed, with, for each step, the generated (partial) execution witness.

In §3.2 we saw the need to commit actions in $rf$ and in $mo$ order. To reduce unnecessary non-determinism, we want to follow $hb$ as much as possible (recall that happens-before ($hb$) is calculated from a candidate execution, and includes program order and synchronisation). For a candidate execution $ex$, we define the *commitment order* of $ex$ to be

$$com\_order(ex) = \left(ex.rf \cup ex.mo \cup ex.hb^{-}\right)^{+}$$

where $\cdot^{+}$ is transitive closure and $ex.hb^{-}$ is defined as

$$\{(a,b) \in ex.hb \mid b \text{ not an atomic write}\}.$$

**Lemma 2.** *Let $ex$ be a consistent execution, then the commitment order of $ex$ is indeed an order: it is transitive and irreflexive.*

If we add either $sc$ or $lo$ to the commitment order, the resulting relation is in general not an order.

Now that we have established the order, we look at what the partially generated execution witnesses should look like. For a candidate execution $ex$, let $C$ be the set of actions that have been committed in a certain state. We define $restrict(ex, C)$ to be the execution witness $wit$ with

$$\begin{aligned} wit.rf &= ex.rf \cap C \times C \\ wit.mo &= ex.mo \cap C \times ex.actions \\ wit.sc &= ex.sc \cap C \times C \\ wit.lo &= ex.lo \cap C \times C \end{aligned}$$

To explain why we restrict $mo$ to $C \times ex.actions$ instead of $C \times C$, recall that the incremental concurrency model commits actions in $mo$ order, so even before we commit an action $b$, we know that $b$ will be modification-order after any action $a$ to the same location that has been committed. We want to include this information as early as possible in the generated (partial) execution witness, because it constrains some choices.

The orders $sc$ and $lo$ are not included in the commitment order, so the incremental concurrency model needs to be able to insert new actions in those orders before actions it already committed. Hence, we only know how actions $a$ and $b$ are related to each other in those orders when both of them have been committed, and therefore we restrict the relations to $C \times C$.

We restrict $rf$ to $C \times C$ because we only want to create an $rf$ edge when we commit a read, and reads can only read from committed writes.

***The transition relation*** The incremental concurrency model can transition from state $s_1$ to $s_2$ if:

1. $s_2.committed = s_1.committed \cup \{a\}$, where $a$ is an action with $a \notin s_1.committed$ (to ensure that a single, new action is committed during the transition);
2. $restrict(s_2.wit, s_1.committed) = s_1.wit$ (in other words, $s_2$ extends $s_1$: all execution witness data present in $s_1$ is present in $s_2$);
3. for every action $b$ we require $b \in s_1.committed \rightarrow (a,b) \notin com\_order(s_2)$ and $(b,a) \in com\_order(s_2) \rightarrow b \in s_1.committed$ (requiring that the transition respects the commitment order we defined earlier in this section); and
4. $s_2$ is consistent according to the incremental consistency predicate $is\_consistent\_inc$ that we define below.

***The incremental consistency predicate*** The incremental consistency predicate $is\_consistent\_inc$ is the same as the predicate $is\_consistent$ of the axiomatic model except for the following seven conjuncts.

The predicate $det\_read$ requires that for every load $r$ there exists a visible side effect [5] if and only if $r$ reads from somewhere. The incremental predicate $det\_read\_inc$ only requires that for committed loads.

The predicate $rmw\_atomicity$ requires that every read-modify-write reads from its immediate predecessor in modification order if it exists, and does not read from anywhere otherwise. The incremental predicate $rmw\_atomicity\_inc$ only requires that for committed read-modify-writes.

The predicate $consistent\_locks$ requires that for all successful locks $a$ and $c$ with $(a,c) \in lo$ there exists an unlock $b$ with $(a,b)$ and $(b,c) \in lo$. The incremental predicate $consistent\_locks\_inc$ only requires that in the case that $c$ has been committed.

The predicate $well\_formed\_rf$ requires some properties of $a$ and $b$ for every pair $(a,b) \in rf$, for example that $a$ is a write, $b$ a read and their values correspond. In addition to all the properties of $a$ and $b$ that $well\_formed\_rf$ requires, $well\_formed\_rf\_inc$ requires that $a$ and $b$ are committed.

The predicates $consistent\_mo$, $consistent\_sc$ and $consistent\_lo$ have similar forms and need to be adapted in similar ways. They define three conditions, say $\varphi_{mo}$, $\varphi_{sc}$ and $\varphi_{lo}$, over pairs of actions, and require that $(a,b)$ or $(b,a) \in mo$ if and only if $\varphi_{mo}(a,b)$ holds (resp. $sc$ and $lo$). The incremental variants of those predicates change the conditions $\varphi$. In the case of $mo$ the condition becomes "$\varphi_{mo}(a,b)$ and either $a$ or $b$ is committed". In the case of $sc$ the condition becomes "$\varphi_{sc}(a,b)$ and both $a$ and $b$ are committed", and similarly for $\varphi_{lo}$. In addition to the change to $\varphi_{mo}$, $consistent\_mo\_inc$ also requires for every $(c,d) \in mo$ that $c$ is committed.

**Theorem 3.** *Let $ex$ be a finite candidate execution. Then $ex$ is consistent according to the axiomatic model if and only if it is a reachable state of the incremental concurrency model in which all the actions have been committed.*

Recall from §3.2 that we have to allow partial executions to be temporarily inconsistent. It is possible that a partial execution has no (outgoing) transitions and can therefore not be extended to a consistent execution. However, one direction of the theorem (soundness) states that if there is a transition to a state where all actions have been committed, then consistency is always regained. The proof is straightforward: if all the actions have been committed, then $is\_consistent\_inc$ implies $is\_consistent$. In the remainder of this subsection we sketch the (mechanised) proof of the other direction (completeness).

We identify the properties of $hb$ that the completeness proof depends on. There are three trivial properties: $hb$ is a relation over the actions of the pre-execution, we have $sb \subseteq hb$, and pairs $(a,b) \in lo$ with $a$ an unlock and $b$ a lock are in $hb$. Then there are two nontrivial properties.

The first is that $hb$ grows monotonically during transitions of the incremental concurrency model. We prove something

stronger, namely for every set $C$ of actions that is $mo$-downclosed (if $(a, b) \in mo$ and $b \in C$ then $a \in C$), we have $restrict(ex, C).hb \subseteq ex.hb$. This is stronger because $mo$ is included in the commitment order.

The second is that $hb$ is stable for certain prefixes: let $c \in C$ with $c$ not an atomic write, then no action $a$ can retroactively happen before $c$. Because $hb$ does not need to be transitive, this is expressed as follows: $(a, b) \in ex.hb$ with $b = c \vee (b, c) \in ex.hb$ implies $(a, b) \in restrict(ex, C).hb$. We proved that this holds for all sets $C$ that are downclosed in the commitment order.

(We conjecture that the above properties of $hb$ remain true in the presence of memory order consume. Because adding consume only changes $hb$, this would mean that our incremental concurrency model also covers the memory order consume correctly.)

To show that the incremental concurrency model can transition along the commitment order, we prove the following.

**Lemma 4.** *Let $ex$ be a consistent execution and $C$ a set of committed actions that is downclosed in the commitment order. Then $is\_consistent\_inc(restrict(ex, C))$.*

**Lemma 5.** *Let $ex$ be a finite consistent execution and $C$ a prefix of the commitment order. There is a finite number of transitions that the incremental concurrency model can take and which result in a state $s$ with $s.committed = C$ and $s.wit = restrict(ex, C)$.*

We prove this by induction on the size of $C$. In the induction step we have to prove that the incremental concurrency model can transition from the state given by the induction hypothesis to the state with $s.committed = C$ and $s.wit = restrict(ex, C)$, so we have to check all the conditions of the transition predicate as defined earlier in this section. The most interesting condition is that $is\_consistent\_inc$ needs to hold for $s.wit$, which is what Lemma 4 gives us.

### 3.5 The operational concurrency model

Rather than (conceptually) enumerating all of a large set of potential transitions and then filtering them, as above, we now define a more precise transition relation that characterises how the generated (partial) execution witness can change during a transition, which greatly reduces the number of possible transitions one has to filter and therefore makes it feasible to compute the set of possible transitions. There are opportunities for further optimisation which we discuss at the end of this subsection.

To compute the set of possible transitions, we generate a set of candidate execution witnesses and then filter them with the $is\_consistent\_inc$ predicate of the previous subsection. If the action that is committed has sequential consistent memory order, we generate a candidate execution witness for every insertion point in the $sc$ order. Otherwise, the $sc$ order does not change. We describe how the other relations $rf$, $mo$ and $lo$ change depending on the type of the committed action.

***Loads*** The consistency predicate requires that loads read from a write if and only if there is a side effect visible to the load. Because reading from a write can cause the write to become a visible side effect, we cannot decide a priori whether the load $r$ that is being committed has to read from somewhere or not, so we consider both possibilities: we generate a candidate witness where $rf$ stays the same, and for each write $w$ that has been committed that has the same location and value as $r$ we generate a candidate witness where $(w, r)$ is added to $rf$. The relations $mo$ and $lo$ do not change.

***Stores*** With $w$ the store being committed, we generate only one candidate witness: we add the $mo$-edges $(w, w')$ for all writes $w'$ to the same location that have not yet been committed (we explained the reason for this when we defined $restrict$ earlier in this section). The relations $rf$ and $lo$ do not change.

***Read-modify-writes*** We generate one candidate witness. The relation $mo$ changes in the same way as in the case of a store. If there exists a write $w$ that is immediately before (in $mo$ order) the read-modify-write $rmw$ that is being committed, then we add $(w, rmw)$ to $rf$. Otherwise $rf$ stays the same. The relation $lo$ does not change.

***Blocked read-modify-writes*** Blocked read-modify-writes model a read-modify-write that is blocked indefinitely. All relations stay the same.

***Locks and unlocks*** We generate a candidate witness for all insertions points in $lo$. The other relations stay the same.

***Fences*** If the fence has sequential consistent memory order, $sc$ changes as described before. All other relations stay the same. Note that a fence that is not sequential consistent does not change the generated execution witness, but it does restrict future choices.

**Theorem 6.** *The transition relations of the operational concurrency model and the incremental concurrency model of §3.4 are identical.*

Soundness is straightforward to show, because we filter the candidate executions with the same predicate as in the incremental concurrency model. The completeness proof is more interesting and depends on some subtleties of the $is\_consistent\_inc$ predicate.

The semantics can be optimised further. Candidate execution witnesses are filtered with the $is\_consistent\_inc$ predicate, but we conjecture that most conjuncts hold by construction. Furthermore, the state of the operational concurrency model contains the execution up to that point so it grows at each step, but some actions are irrelevant for future transitions (for example writes from which new reads cannot read) so they could be pruned from the state.

We can operationally detect races in a single path, though of course to find all races (and hence to determine whether a program is well defined) we need to explore all paths; that is unfortunate but intrinsic to the C/C++11 semantics.

### 3.6 Integration with an operational threadwise semantics

In §3.5 we described the use of our operational concurrency model with an axiomatic threadwise semantics, which provides complete pre-executions. We now integrate the operational concurrency model with an operational threadwise semantics, to build 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 expressed as a Core program.

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 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 value in Core programs must be symbolic. This means in particular that the execution of control operator (Core has a single if-then-else construct) is done symbolically. When a control point is reach, 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.

While the interleaving execution of the threadwise semantics and the concurrency model has the advantage of allowing the exploration of larger programs, for exhaustively finding all possible executions of a program the explosion of nondeterminism it introduces quickly becomes intractable. For validation purposes, we therefore first run the threadwise semantics on the whole Core program, to produce a complete pre-execution, and then invoke the concurrency model on that.

We have exercised the semantics, integrated into an executable as above, on the examples of §4 (translated into Core programs), for which it gives the correct C/C++11 results. Time has not yet permitted more extensive testing; we have tried a few other litmus tests taken from `cppmem` [5], most of which give the correct results but one reveals what appears to be a bug in the driver harness (the operational concurrency model of §3.5 gives the correct results for that test when integrated with `cppmem`).

To the best of our knowledge this is the first operational model for C/C++11 concurrency. The most closely related work we are aware of is the model-checker of Norris and Demsky [24]. This is focussed on efficiency, but its relationship to the C/C++11 model is not completely clear (they note that their CDSCHECKER "may not explore all behaviors involving satisfaction cycles", but it is unclear what other executions are also excluded).

## 4. The thin-air problem has no per-candidate-execution solution

The question of "thin-air" reads is a longstanding issue in the design of memory models for C and C++, specifically for C/C++11 relaxed atomics: accesses for which races are permitted but which should be implemented with normal load and store instructions, without the cost of additional barriers or synchronisation instructions. Related questions arise in the semantics of C as used in the Linux Kernel (for `ACCESS_ONCE` accesses), and in Java [20].

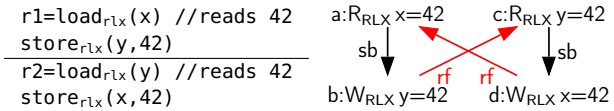
The C++11 standard [6] included text intended to forbid thin-air executions (29.3p9), and it says explicitly (29.3p10) that that text forbids the LB+data example below, but the text was already recognised as flawed: a non-normative note in the standard (29.3p11) observed that *"The requirements do allow [the LB+ctrldata+ctrl-single example below]. However, implementations should not allow such behavior."*. Batty et al. identified further problems [5, §4], and their formal model does not attempt to capture that text or to exclude thin-air executions in any other way. The current proposal [8] for C++14 acknowledges difficulties with the C++11 version and proposes a deliberately vague placeholder as an interim replacement: *"Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation."*.

There is not a precise definition of what it means for a read to be "out of thin air" (if there were, the problem would be solved, as the semantics could simply exclude those). Rather, there are some example executions for which there is a consensus that the language should forbid them, and that current hardware and compiler optimisations do not exhibit. This is a high-level-language specification problem: there is no suggestion that thin-air executions occur in practice with current compilers and hardware; the problem is rather how to exclude them without preventing desired compiler optimisations.

In this section, we describe the thin-air problem via a series of examples, and we show that thin-air executions cannot be forbidden without restricting current compiler optimisations by any per-candidate-execution condition using the C/C++11 notion of candidate executions.

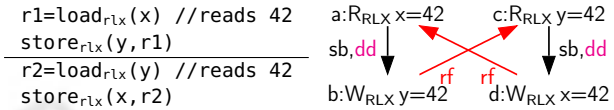
For each example we identify a particular execution by specifying the values read, and discuss whether it should be allowed by the semantics or not. Here all locations are initially `0`.

***Example LB (language must allow)***




Returning to the first example of the previous section, this execution is permitted by the ARM and IBM POWER architectures (presuming the code is compiled in the obvious way into machine load and store instructions): the actions of the each thread are to manifestly different addresses and so can be done out of order; it is moreover experimentally observable on current ARM multiprocessors [28]. Hence, the language semantics must allow it for relaxed atomics.

***Example LB+datas (language can and should forbid)***

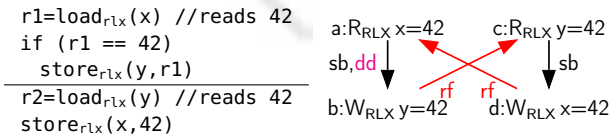



There are two paradigmatic kinds of thin-air execution, the *thin-air read value* executions like this one, in which a value (here `42`) "appears out of thin air", and the *self-satisfying conditional* example we discuss below. This example is architecturally forbidden on current hardware (x86, ARM, and IBM POWER), we do not expect future hardware to adopt the load-value prediction that would be required to make it observable, and to the best of our knowledge it cannot be exhibited by any reasonable current compiler optimisation combined with current hardware. Hence, the language semantics could forbid it.

Moreover, it is clearly desirable to forbid it, to make the language semantics as intuitive as possible. Boehm and Demsky [10] give examples where programming with relaxed atomics that permit thin-air values would be problematic, and in languages that aim to preserve implementation invariants at some types (such as that all pointer values point to allocated memory) it would be essential.

As for *how* it might be forbidden, the example suggests that one might simply forbid candidate executions with cycles in the union of the reads-from and dependency relations (the model has a data dependency relation shown as *dd* above). But the next two examples show that a combination of hardware behaviour and compiler optimisations make that infeasible.

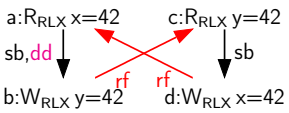
***Example LB+ctrldata+po (language must allow)***



This is architecturally allowed on ARM and Power (for the same reason as LB), and likewise observable on ARM, hence the language must allow it.

***Example LB+ctrldata+ctrl-double (language must allow)***

```
r1=load_rlx(x) //reads 42
if (r1 == 42)
  store_rlx(y,r1)
-------------------------
r2=load_rlx(y) //reads 42
if (r2 == 42)
  store_rlx(x,42)
else
  store_rlx(x,42)
```

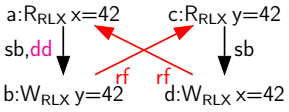



This is forbidden on hardware if compiled naively, as the architectures respect read-to-write control dependencies, but in practice compilers will collapse conditionals like that of the second thread, removing the control dependencies from the read of y to the writes of x and making the code identical to the previous example. As that example is allowed and observable on hardware (and we presume that it would be impractical to outlaw such optimisation for C or C++), the language must also allow this execution. But this execution has a cycle in the union of reads-from and dependency, so we cannot simply exclude all those.

Then one might hope for some other adaptation of the C/C++11 model, but the following example shows at least that there is no per-candidate-execution solution.

***Example LB+ctrldata+ctrl-single (language can and should forbid)***

```
r1=load_rlx(x) //reads 42
if (r1 == 42)
  store_rlx(y,r1)
-------------------------
r2=load_rlx(y) //reads 42
if (r2 == 42)
  store_rlx(x,42)
```




This is the paradigmatic "self-satisfying conditional" example. It is forbidden on hardware if compiled naively (both ARM and POWER architectures prevent speculative writes becoming visible to other threads), and applying reasonable thread-local compiler optimisation does not change that. Hence, the language could forbid it. Moreover, it is problematic for informal and formal compositional reasoning [3, 10, 31], so the language should forbid it.

But the candidate execution that we want to forbid here is identical to the execution of the previous example that we have to allow. Hence, we cannot do both simultaneously with any adaptation of the C/C++11 per-candidate-execution definitions that uses the same notion of candidate execution.

The basic point here is that compiler optimisations (such as the collapse of the LB+ctrldata+ctrl-double conditional) are operating over a representation of the *program*, covering all its executions, while the C/C++11 definition of candidate execution and consistency for those considers each candidate *execution* independently (it ignores the set of all executions); it is not able to capture the fact that the conditional is unnecessary because the two candidate executions corresponding to taking the two branches are equivalent. We develop this observation in §6.

***Restricting optimisation involving relaxed atomics?*** One might think that it would be feasible to restrict just compiler optimisations involving relaxed atomics, e.g. requiring that the compiler should respect all dependencies between relaxed atomic operations, while permitting more optimisation elsewhere. But (as observed by Boehm [7]) dependencies can be via functions in other compilation units that only involve non-atomic accesses, e.g. as in the version of LB+ctrldata+ctrl-double below, where the second thread's conditional is factored out into a function `f()` that does not involve atomics and that is in a different compilation unit. When compiling `f()` the compiler cannot tell whether it might be used in a dependency chain between atomic accesses, and so it would have to preserve all such dependencies. The cost of that is unknown, and worth investigating experimentally, but we suspect it to be unacceptable.

```
// in one compilation unit
void f(int ra, int*rb) {
  if (ra==42)
    *rb = 42;
  else
    *rb = 42; }

// in another compilation unit
  r1=load_rlx(x) //reads 42  |  r2=load_rlx(y) //reads 42
  if (r1 == 42)              |  f(r2,&r3)
    store_rlx(y,r2)          |    store_rlx(x,r3)
```

In practice, GCC (checked with 4.6.3 on x86) does optimise away the control dependency in `f()`, at `O1`, `O2`, or `O3`.

## 5. Integrating non-atomics and atomics leads back to thin air

We now show that the thin-air problem is not confined to relaxed atomics.

The C++11 standard prose refers to *"atomic objects"* as if they are quite different from non-atomic objects, and the mathematical model of Batty et al. [5] for the C++11 and C11 concurrency primitives followed suit by imposing a simple type discipline: a *location kind* map in each candidate execution partitioned locations into atomic, nonatomic, and mutex locations. The definition of consistent execution permitted atomic accesses only at atomic locations, and the only nonatomic accesses allowed at atomic locations were atomic initialisations[1].

However, when one considers generalising that semantics for the concurrency primitives to cover more of C, it becomes clear that an up-front location-kind distinction is unrealistic, for several reasons:

1. In C it is permitted to reuse a region of allocated storage (e.g. from `malloc`) at a new type, simply by overwriting the bytes of memory with a new value. Restricting that to prevent strong updates from atomic to nonatomic (or v.v.) would not give a usable language.
2. In C one can inspect the representation bytes of a value by casting a pointer to `(char *)`, or by type-punning via a union.
3. In C one can copy a value by copying its representation bytes, e.g. using `memcpy`. This could perhaps be deemed illegal for structures containing atomic values (indeed, it would have to be if atomic values had to be registered somewhere in the implementation), but it would be preferable, and in keeping with the rest of the language, to permit it.
4. In C11 one can construct atomic versions of structure and union types (with `_Atomic(`*`type-name`*`)` or the `_Atomic` qualifier), but their members can be accessed only via a non-atomic object which is assigned to or from the atomic object, not directly [1, 6.5.2.3p5].

[1] It is desirable to have nonatomic initialisations so that they do not require fences, but then to obtain a DRF-SC result initialisation had to be limited to be happens-before all other accesses, and without reinitialisation.

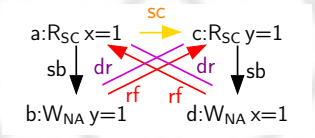
Hence, contrary to [5], we have to allow mixtures of atomic and nonatomic accesses at the same location, at least where the nonatomic accesses do not race with each other or with any atomic accesses.

But what should the semantics be for these? The standard text does not address these mixtures, but for the entirely nonatomic and entirely atomic cases it and the formal model [5] are clear:

- for a non-atomic read, the definition of consistent execution requires, in *consistent_non_atomic_rf*, the read to read from the most recent happens-before-visible write to the same location; while
- for an atomic read, the analogous *consistent_atomic_rf* lets the read read from any write that is not after it in happens-before (subject to the other predicates of the model).

It is this second clause that permits LB and the related thin-air executions for relaxed atomics that we saw earlier. If we allow the mixing of non-atomics and atomic accesses, we can appeal to *consistent_atomic_rf* to write programs that violate the essential DRF-SC property, described in §1 and §2. Our first example program uses `memcpy` to mix atomic and non-atomic accesses at the same location. It is race-free in every SC execution, but it has racy executions in the C/C++11 memory model as it stands:

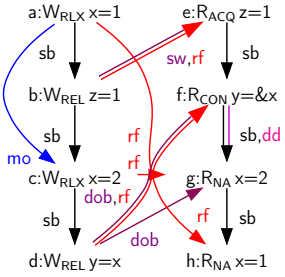
```
// parent thread
size_t s = sizeof(atomic_int)
atomic_int x = 0
atomic_int y = 0
atomic_int a = 1
```
---
```
int r1 = load_sc(x)
if(r1 != 0)
  memcpy(&y,&a,s)
```
---
```
int r2 = load_sc(y)
if (r2 != 0)
  memcpy(&x,&a,s)
```




In the execution above, each atomic load reads from the non-atomic write implicit in the `memcpy` of the other thread. The execution is consistent, and has data races, making it a counterexample to DRF-SC.

In C/C++11 the *consistent_non_atomic_rf* predicate governs the behaviour of reads from non-atomic objects. The example below establishes that this predicate is not suitable for non-atomic reads at locations that mix atomic and non-atomic accesses. In the program below, there is a reading thread that spins until it sees the other thread's writes of `z` and `y`, and then reads from `x` twice: once with acquire memory order and once with consume. After the loop, there are two `memcpy`s of location `x`:

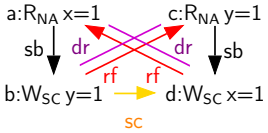
```
// parent thread
size_t s = sizeof(atomic_int)
atomic_int n=0, x=0, y=0, z=0
```
---
```
store_rlx(x,1)
store_rel(z,1)
store_rlx(x,2)
store_rel(y,&x)
```
---
```
do { r1 = load_acq(z)
     r2 = load_con(y)}
while (r1!=0 && r2!=0)
memcpy(&n,r2,s)
memcpy(&n,&x,s)
```




In the candidate execution on the right above, the loop exits (we elide the implicit write of the memcpys, and the initialisation writes). The first `memcpy` happens after all atomic writes of `x`, but before the write implicit in the second `memcpy`, so according to *consistent_non_atomic_rf*, it must read write `c`. The second `memcpy` reads a pointer provided by the consume read, creating a dependency and forcing it to read `a`, but this execution, shown above, contains a CoRR coherence violation between accesses `a`, `c`, `g` and `h`, making the execution inconsistent, so the only behaviour that the model allows of this program is spinning on the conditional of the loop (similar executions arise if we swap atomics with non-atomics and vice versa).

The consistency condition of that semantics cuts out executions we need to allow: this can make reasonable executions of race-free programs inconsistent and remove racy executions from racy programs, making them race-free and well-defined. We explore the possibility of using *consistent_atomic_rf* for governing the behaviour of non-atomic reads of locations that are accessed with a mixture of atomics and non-atomics. The following example shows that this adjustment to the model also breaks the DRF-SC property:

```
// parent thread
size_t s = max(sizeof(atomic_int),sizeof(int))
atomic_int* x = calloc(1,s);
atomic_int* y = calloc(1,s);
int* r1 = calloc(1,s);
int* r2 = calloc(1,s);
```
---
```
memcpy(r1,x,s);
if (*r1 != 0)
  atomic_init(*y,1)
  store_sc(*y,2)
```
---
```
memcpy(r2,y,s);
if (*r2 != 0)
  atomic_init(*x,1)
  store_sc(*x,2)
```




In the candidate execution above, where each `memcpy` reads from the atomic store (a racy execution, with the initialisation writes and implicit writes of the memcpys are elided), we use *consistent_atomic_rf* to decide that the execution is consistent, and the program has undefined behaviour. Contrast this with the SC executions of the program: the program executes no atomic accesses, both conditionals fail in every execution, and the program is race free. This discrepancy violates DRF-SC, even for programs that do not use any atomic accesses in their SC executions.

We have seen that using *consistent_nonatomic_rf* to govern the behaviour of non-atomic reads at locations accessed atomically removes too many behaviours. We cannot use *consistent_atomic_rf* to govern such reads either: that would break DRF-SC. It is not clear what the semantics of non-atomic reads should be in C11.

## 6. An out-of-order operational construction

The examples of §4 showed that for relaxed atomics the language semantics has to admit reorderings that are enabled by removals of syntactic control dependencies that can be justified only by examination of multiple control-flow paths (not just inspection of a single candidate execution). For example, consider again the second thread of LB+ctrldata+ctrl-double:

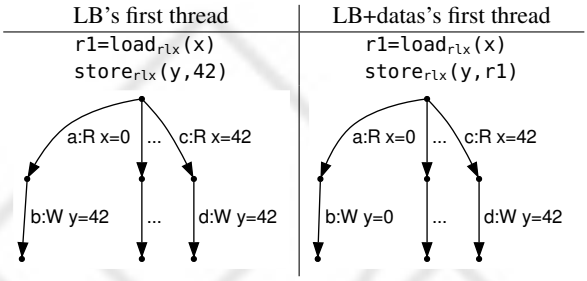
```
r2=load_rlx(x)                r2=load_rlx(y)          store_rlx(x,42)
if (r2 == 42)      compiler   store_rlx(x,42)  h/w    r2=load_rlx(y)
  store_rlx(x,42)   ----->                     ----->
else
  store_rlx(x,42)
```

The key fact here is that the `store_rlx(x,42)` is possible on all control-flow paths of this thread, and a sufficiently "smart" compiler can detect that and then remove the control dependency from the read of `y`. In this section we generalise this observation: we

give a semantics for relaxed and nonatomic accesses (and locks and fences) that correctly accounts for all the thin-air examples of §4 in an interesting and reasonably clean way, though (as we explain in §6.2) this is not a magic bullet: extending it to cover more optimisations reveals other difficulties.

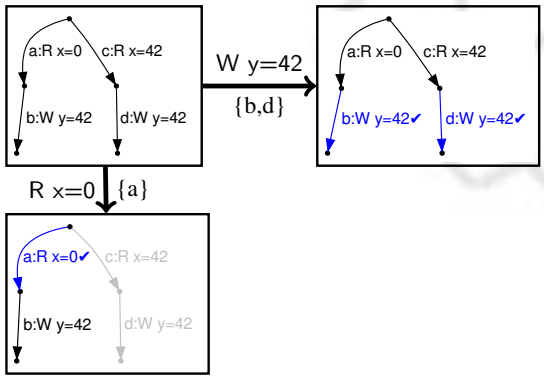
### 6.1 The semantics for reorderings

We start from a standard labelled transition system (LTS) semantics for each thread in isolation, describing its interactions with memory by transitions labelled a:R x=v and b:W x=v for a read or write of value v at location x. This thread-local base semantics does not constrain the values read from memory in any way; it simply has a transition for each possible read value. For example[2]:



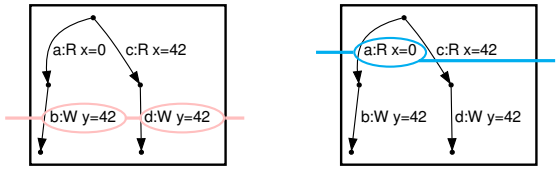

In LB's first thread, in all branches of the LTS, there is a write of 42 to y; and we will allow the thread to write 42 before reading, letting both threads read 42. On the other hand, in LB+datas's first thread, it is not the case that a write of 42 is available in all branches, so it will have to do the read first, preventing LB+datas from exhibiting out-of-thin-air behaviour.

We capture this by constructing a derived *out-of-order* labelled transition system for each thread. Its states are copies of the entire base in-order LTS with some edges *ticked*. The initial state is the base LTS with no edge ticked. For example, part of the out-of-order LTS for LB's first thread is shown below.




Its transitions are labelled with the same memory actions as the base semantics; each transition of the derived LTS corresponds to ticking a set of base transitions. But the base transitions can be performed out-of-order, when they are not blocked (as we define below) in any branch by coherence or fences. Specifically: a set of edges can be ticked iff it forms a *frontier*, that is, it is non-empty, the edges are not ticked, the edges have the same memory action label, there is exactly one edge per non-discarded path that is not going to be discarded by this ticking (this proviso is for reads: only one sibling of each set of siblings induced by a read can be in a frontier), and no edge is blocked (see below). Here an edge is *discarded* if it is has a ticked sibling, and a path is discarded if it contains a discarded edge.

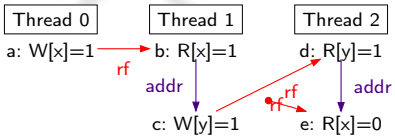
For example, the horizontal transition above is justified by the frontier consisting of all the W y=42 edges (b, d, and all the similar edges in elided paths), while the vertical transition is justified by the frontier consisting just of a (and there is a similar transition, not shown, for each base transition with a different read value).




To maintain coherence (the fact that execution respects a per-location total order over writes to each location, consistent with program order, as guaranteed by standard hardware and by C11 relaxed atomics), the definition of "blocked" ensures that an action cannot be ticked before all the previous actions to the same location have been ticked. Ticking also respects fences: all the actions before the fence have to be ticked before the fence can be ticked; moreover, all the actions before the fence and the fence itself have to be ticked before actions after the fence can be ticked. Lock and unlock actions block actions after and before them, respectively, but not the other way around, to allow for roach motel reordering.

***Handling nonatomics*** Non-atomic accesses can be executed out-of-order, like relaxed accesses, but in addition, they can also cause races. As non-atomic accesses can race with atomic accesses, all accesses are logged to detect races.

***Non-multi-copy-atomic memory*** For two-thread examples, one can combine the derived LTS of each thread with an underlying sequentially consistent shared memory (and that is what we have done for the testing described below). But in general the language semantics must also admit the lack of *multi-copy atomicity* permitted by the Power and ARM architectures: writes to different locations can be propagated to other threads in different orders. For example, at the hardware level, in the test below the fact that Thread 0's write of x propagates to Thread 1 before its write of y can be committed (due to the address dependency) does not guarantee that the write of x has propagated to Thread 2 before the write of y does.




Test WRC+addrs: Allowed

This could be handled by combining the derived LTS of the threads with a non-multi-copy-atomic storage subsystem semantics following that of Sarkar et al. [28].

We have a precise definition of the out-of-order semantics construction (available in the supplementary material), and built a tool that lets one explore the semantics of small examples, based on OCaml code generated by Lem from the semantics, and integrated with an underlying SC memory. That confirms that it does give the desired behaviour for each of the thin-air examples of §4: the semantics is liberal enough to allow the reordering (introduced by compiler or hardware) that gives rise to the "must be allowed" examples, and restrictive enough to prevent the "should be forbidden" examples, ruling out thin-air executions basically by executing along a totally ordered trace of the derived LTS, with reads

[2] We only show the branches for the values 0 and 42; in reality there is one branch per possible value, as we assume the base LTS is receptive.

reading from previous writes in that trace. At present it does not cover non-relaxed atomic accesses, so we cannot discuss the examples of §5.

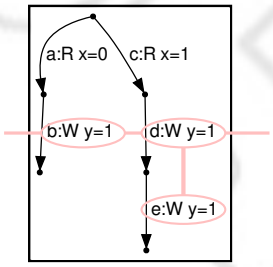
This out-of-order semantics has several good features:

- It is operational and relatively concrete, which makes it easier to understand than (say) the C11 axiomatic memory model.
- The construction is independent from the language syntax and thread-local operational semantics, which is highly desirable for tackling a complex language like C. This is in contrast to calculi with explicit speculation, e.g. Boudol and Petri [11] and Jagadeesan et al. [19].
- It does not involve syntactic notions of dependency, which are difficult for compilers to preserve.

However, looking at how to extend it to cover other optimisations highlights some subtle issues that any semantics for C will have to tackle. Rather than considering the syntactic optimisations that are performed by compilers (GCC and Clang each have of the order of 100 passes, and it is unclear exactly what each does) we focus on the abstract classes of optimisations used by Ševčík [32] and Morisset et al. [22].

### 6.2 Issues with other optimisations

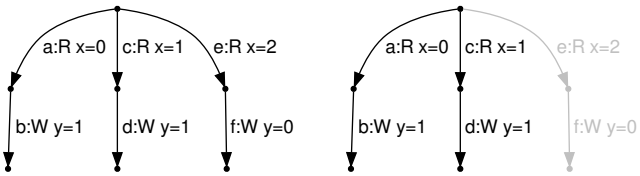
***X after Y elimination*** Read after read, read after write, write after read, and overwritten write elimination consist in conflating actions when the effect of one subsumes that of the others. We conjecture that the notion of frontier can be relaxed to deal with these, e.g. with extended frontiers as below.




These optimisations need information about multiple paths, but only in a limited way: they only need the existence of particular actions (in a non-blocked path context) in each path, and read and write introductions appear to have a similar character. However, this is unfortunately not the case for all optimisations:

***Irrelevant read elimination*** Intuitively, irrelevant read elimination consists in removing a read action when its result does not affect the thread's behaviour: for example, if the branches of a read have identical subtrees, it is certainly irrelevant. But in general a read is irrelevant if its subtrees are in some sense semantically equivalent, and it is not clear what notion of equivalence is appropriate here — in principle it should also allow reorderings, introductions, and eliminations, suggesting a recursive construction, but that would be difficult to use and reason about.

***Inter-thread optimisations*** By requiring that the thread-local LTS has a branch for every value of each read, the out-of-order construction essentially builds in the assumption that there are no inter-thread optimisations. This is to some extent reasonable for C, and perhaps also Java, as compilers typically work per compilation unit (though link-time optimisation is becoming more feasible). It is in contrast with some of the examples considered in the JMM design. But if any inter-thread analysis (such as alias analysis) determines that a variable can only contain certain values, then that might enable optimisations that the out-of-order construction does not permit. Identifying a value restriction amounts to discarding some "impossible" branches of the LTS, but this creates more valid frontiers, and hence permits more out-of-order behaviour (which can even invalidate the analysis). For example, in the LTS below, if, by looking at all the writes to x by all the threads, the compiler determines that x can only contain values 0 and 1, then it can discard the branch where the value 2 is read, which makes $\{b, d\}$ into a frontier, which allows the write to y to be executed before the read from x:




Moreover, as the value restrictions can vary during program execution, these inter-thread optimisations cannot be separated to an initial phase, but have to be intertwined with the other optimisations. As a result, the memory model would have to be defined recursively in yet another way.

***Thread-local and shared variables*** The out-of-order semantics is defined over a calculus that has a syntactic distinction between thread-local variables and potentially-shared variables. This distinction is important, as the semantics does not need to consider interference on thread-local variables, and thread-local optimisations on them are built into the base LTS construction. This might be reasonable in some languages but C does not have such a distinction, and whether a variable behaves thread-locally depends on the dynamic behaviour of the program.

***Relation to the operational memory model of §3*** Note that this model has very different goals and properties to the operational memory model of §3: that model is equivalent to the axiomatic memory model of C11, and therefore does exhibit out-of-thin-air behaviour; whereas the out-of-order operational semantics described in this section aims at providing a semantics for relaxed atomics *without* exhibiting out-of-thin-air behaviour, and at highlighting the problems faced by an operational memory model that avoids out-of-thin-air behaviour.

## 7. Conclusion

The C/C++11 concurrency model remains the state of the art for the semantics of a general-purpose shared-memory concurrent programming languages; it is, to the best of our knowledge, sound with respect to the compiler optimisation behaviour of implementations [22] (in contrast to the JMM [12, 29]), it is provably compilable to relaxed hardware models [4, 5, 27], and our work here establishes a machine-checked DRF-SC theorem and an equivalent operational model. But the thin-air problem shows that it allows too many behaviours, and we have shown here that that cannot be solved in a simple per-candidate-execution way, that the problem is not specific to relaxed atomics, and that, while an operational solution for those examples is possible, it brings other difficulties.

We conclude by summarising the possible approaches to the semantics of shared-memory concurrent languages that we are aware of. Most seem problematic:

- One could restrict to sequentially consistent concurrency, banning low-level atomics, but the basic driver for multicore machines is performance; it seems unlikely that the cost would be acceptable (e.g. for OS kernel code). Moreover, as we showed in §5, in a C-like language with integrated nonatomics and atomics, the difficulties would remain.

- The semantics could ban all dependency cycles, but (per Boehm, and as described in §4) that would rule out standard compiler optimisations.
- Banning just the dependency cycles involving relaxed has the difficulty that dependency cycles might go via other compilation units that do not use atomics (also per Boehm and in §4).
- One might imagine changing the definition of consistent execution, but, as we showed in §4, that cannot suffice while admitting the current compiler optimisations.
- One could declare programs with executions containing cycles (or perhaps un-annotated cycles) in the dependency and reads-from relations to have undefined behaviour, as proposed by Batty and Sewell, effectively making it the programmer's responsibility to avoid them, but that seems unlikely to be workable in practice.
- One could prevent all load-to-store reordering, as proposed by Boehm and Demsky [8, 10], but that comes with a potentially unpleasant performance cost that is difficult to justify for a general-purpose language that aims to support systems code (moreover, the §5 examples show that additional cost would be needed).
- One could try to develop an explicit out-of-order semantics along the lines of §6 or speculation calculi [11, 19], but irrelevant read elimination, inter-thread optimisation, and the apparent need to distinguish thread-local and shared variables are all challenging for a C-like language.
- One could lift a hardware memory model such as TSO to the language level, as in the CompCertTSO verified compiler of Ševčík et al. [33] and the BMM Java model of Demange et al. [14], but this both limits compiler optimisations and (for TSO) requires additional fencing on more relaxed architectures, so (while interesting in some contexts) this seems unlikely to be an acceptable general solution. It would be instructive to characterise more precisely the compiler optimisations that are not admissible in the ARM and Power architectural models.

The only remaining alternative that we see, short of regarding the implementations as defining the semantics, is to enumerate the abstract compiler optimisations, as used by Ševčík [32] and Morisset et al. [22], and base the semantics on the transitive closure of what they allow. This is broadly along the lines of the proposal by Saraswat et al. [26]. That transitive closure will be awkward to work with, but one might show that the current C/C++11 model is sound above that, generalising results of [22, 32].

Disturbingly, 40+ years after the first relaxed-memory hardware was introduced (the IBM 370/158MP), the field still does not have a credible proposal for the shared-memory concurrency semantics of any general-purpose high-level language that includes high-performance concurrency primitives.

**Acknowledgements** We would like to thank Hans Boehm, Jaroslav Ševčík, Ali Sezgin, Viktor Vafeiadis, and Francesco Zappa Nardelli for discussions about parts of this work. We acknowledge funding from EPSRC grants EP/H005633 (Leadership Fellowship, Sewell) and EP/K008528 (REMS Programme Grant), and a Gates Cambridge Scholarship (Nienhuis).

## References

[1] *Programming Languages — C*. 2011. ISO/IEC 9899:2011. `http://www.open-std.org/jtc1/sc22/wg14/docs/n1539.pdf`.

[2] S. V. Adve and M. D. Hill. Weak ordering — a new definition. In *Proc. ISCA*, 1990.

[3] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *Proc. POPL*, 2013.

[4] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Proc. POPL*, 2012.

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

[6] P. Becker, editor. *Programming Languages — C++*. 2011. ISO/IEC 14882:2011. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`.

[7] H.-J. Boehm. Memory model rationales. `http://open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2176.html`, March 2007.

[8] H.-J. Boehm. N3786: Prohibiting "out of thin air" results in C++14. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3786.htm`, September 2013.

[9] H.-J. Boehm and S.V. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, 2008.

[10] H.-J. Boehm and B. Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proc. MSPC*, 2014.

[11] G. Boudol and G. Petri. A theory of speculative computation. In *Proc. ESOP*, 2010.

[12] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *Proc. ESOP*, 2007.

[13] L. Crowl. Recent concurrency issue resolutions. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3278.htm`.

[14] D. Demange, V. Laporte, L. Zhao, S. Jagannathan, D. Pichardie, and J. Vitek. Plan B: A buffered memory model for Java. In *POPL*, 2013.

[15] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15:399–407, 1992.

[16] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[17] The HOL 4 system. `http://hol.sourceforge.net/`.

[18] Isabelle. `http://isabelle.in.tum.de/`.

[19] R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *Proc. ESOP*, 2010.

[20] J. Manson, W. Pugh, and S.V. Adve. The Java memory model. In *Proc. POPL*, 2005.

[21] A. Meredith. C++ standard library defect report list. `http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-defects.html`. n3822.

[22] 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.

[23] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. In *Proc. ICFP*, 2014.

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

[25] W. Pugh. Fixing the Java memory model. In *Proc. ACM 1999 Conference on Java Grande*, 1999.

[26] V. A. Saraswat, R. Jagadeesan, M. M. Michael, and C. von Praun. A theory of memory models. In *Proc. PPOPP*, 2007.

[27] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *Proc. PLDI*, 2012.

[28] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proc. PLDI*, 2011.

[29] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, 2008.

[30] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proc. OOPSLA*, 2014.

[31] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *Proc. OOPSLA*, 2013.

[32] J. Ševčík. Safe optimisations for shared-memory concurrent programs. In *Proc. PLDI*, 2011.

[33] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, June 2013.