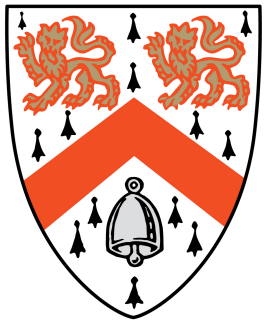
The C11 and C++11 Concurrency Model

Mark John Batty
Wolfson College
University of Cambridge



Saturday 29th November, 2014

This dissertation is submitted for the degree of Doctor of Philosophy

Declaration

This thesis does not exceed 61,400 words, and prior permission was granted for an extension of 1,400 words.

Mark John Batty

The C11 and C++11 Concurrency Model

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD, UK

Phone: 07962 440124
Email: mbatty@cantab.net
Homepage: www.cl.cam.ac.uk/~mjb220

Abstract

Relaxed-memory concurrency is now mainstream in both hardware and programming languages, but there is little support for the programmer of such systems. In this highly non-deterministic setting, ingrained assumptions like causality or the global view of memory do not hold. It is dangerous to use intuition, specifications are universally unreliable, and testing outcomes are dependent on hardware that is getting more permissive of odd behaviour with each generation. Relaxed-memory concurrency introduces complications that pervade the whole system, from processors, to compilers, programming languages and software.

There has been an effort to tame some of the mystery of relaxed-memory systems by applying a range of techniques, from exhaustive testing to mechanised formal specification. These techniques have established mathematical models of hardware architectures like x86, Power and ARM, and programming languages like Java. Formal models of these systems are superior to prose specifications: they are unambiguous, one can prove properties about them, and they can be executed, permitting one to test the model directly. The clarity of these formal models enables precise critical discussion, and has led to the discovery of bugs in processors and, in the case of Java, x86 and Power, in the specifications themselves.

In 2011, the C and C++ languages introduced relaxed-memory concurrency to the language specification. This was the culmination of a six-year process on which I had a significant impact. This thesis details my work in mathematically formalising, refining and validating the 2011 C and C++ concurrency design. It provides a mechanised formal model of C and C++ concurrency, refinements to the design that removed major errors from the specification before ratification, a proof in HOL4 (for a restricted set of programs) that the model supports a simpler interface for regular programmers, and, in collaboration with others, an online tool for testing intuitions about the model, proofs that the language is efficiently implementable above the relaxed x86 and Power architectures, a concurrent reasoning principle for proving specifications of libraries correct, and an in-depth analysis of problems that remain in the design.

Contents

Chapter 1

Introduction

The advent of pervasive concurrency has caused fundamental design changes throughout computer systems. In a bid to offer faster and faster machines, designers had been producing hardware with ever higher clock frequencies, leading to extreme levels of power dissipation. This approach began to give diminishing returns, and in order to avoid physical limitations while maintaining the rate of increase in performance, processor vendors embraced multi-core designs. Multi-core machines contain several distinct processors that work in concert to complete a task. The individual processors can operate at lower frequencies, while collectively possessing computing power that matches or exceeds their single core counterparts. Unfortunately, multi-core processor performance is sensitive to the sort of work they are given: large numbers of wholly independent tasks are ideal, whereas monolithic tasks that cannot be split up are pathological. Most tasks require some communication between cores, and the cost of this communication limits performance on multi-core systems.

Communication between cores in mainstream multi-core machines is enabled by a shared memory. To send information from one core to another, one core writes to memory and the other reads from memory. Unfortunately, memory is extremely slow when compared with computation. Processor designers go to great lengths to reduce the latency of memory by introducing caches and buffers in the memory system. In the design of such a memory, there is a fundamental choice: one can design intricate protocols that hide the details, preserving the illusion of a simple memory interface while introducing communication delay, or one can allow memory accesses to appear to happen out of order, betraying some of the internal workings of the machine. Mainstream processor vendors all opt for the latter: ARM, IBM's Power, SPARC-TSO, and Intel's x86 and Itanium architectures allow the programmer to see strange behaviour at the interface to memory in order to allow agressive optimisation in the memory subsystem.

The behaviour of memory is often abstracted from the rest of the computer system, and is defined by a *memory model* — a description of the allowable behaviours of the memory system. A simple memory model might guarantee that all threads' loads and

stores to memory are interleaved, to form a sequence in which they take effect. A memory model that exposes more behaviours than this simple model is said to be *relaxed*.

Programming languages for concurrent systems must also define an interface to memory, and face the same choice of whether to permit relaxed behaviour or not. Relaxed behaviour can be introduced at the language level either by the underlying processor, or by optimisations in the compiler. If the language decides to forbid relaxed behaviour, then the compiler must disable any optimisations that observably reorder memory accesses, and insert instructions to the processor to disable relaxed behaviour on the hardware. There is an important tradeoff here: disabling these optimisations comes at a cost to performance, but allowing relaxed behaviour makes the semantics of programs much more difficult to understand.

This thesis describes my work on the relaxed memory model that was introduced in the 2011 revision of the C++ programming language (C++11), and that was adopted by the 2011 revision of C (C11). I chose to work on these languages for several reasons. C is the de facto systems programming language, and intricate shared-memory algorithms have an important role in systems programming. The specification of the language was written in a style that was amenable to a formal treatment, and there were stated design goals that could be established with proof. The language specifications had not yet been ratified when I started analysing them, so there was the potential for wide impact.

To get a flavour of both the behaviour that relaxed memory models allow, and how one uses the new features of C/C++11, consider the following *message-passing* example (written in C-like pseudocode) where a flag variable is used to signal to another thread that a piece of data is ready to be read:

```
          int data = 0;
          int flag = 0;
data = 1; || while (flag <> 1){}
flag = 1; || r = data;
```

Here, a parent thread initialises two integer locations in memory, `data` and `flag`, each to the value 0, and then creates two threads, separated by a double bar indicating parallel composition: one that writes 1 to `data` and then to `flag`, and another that repeatedly reads from `flag`, waiting for the value 1 before reading `data`, then storing the result in thread-local variable `r`, whose initialisation is elided.

One might expect this program to only ever terminate with the final value 1 of variable `r`: that is the only outcome allowed by a naive interleaving of the accesses on each thread. But an analogous program executed on a Power or ARM processor could, because of optimisations in the memory system, terminate with value 0 stored to `r`, and to similar effect, a compiler might reorder the writes to `data` and `flag` when optimising the left-hand thread. This behaviour breaks our informal specification of this code: the data was not ready to be read when we saw the write of the flag variable. In general, as in this

case, relaxed behaviour can be undesirable: the correctness of the program might depend on its absence.

Hardware architectures like Power and ARM give the programmer barrier instructions that can be inserted to forbid this sort of behaviour, but inserting them degrades performance. In a similar fashion, C/C++11 provides the programmer with the *atomics* library that can be used to forbid the outcome where `r` reads 0, as in the following example:

```
                    int data = 0;
                atomic_int flag = 0;
data = 1;           ||  while (flag.load(acquire) <> 1){}
flag.store(1,release); ||  r = data;
```

Here the `flag` variable is declared as an *atomic*, and loads and stores of `flag` use a different syntax that includes a *memory-order* parameter (release and acquire, above). These new load and store functions perform two tasks: depending on the chosen memory order, they forbid some compiler optimisations, and they force the insertion of barrier instructions on some target architectures. Together, the choice of atomic accesses and memory-order parameters above forbids the outcome where `r` reads 0.

The 2011 C and C++ standard documents describe a memory model that defines the behaviour of these atomic accesses. The 2014 revision of C++ leaves this model unchanged, with only minor updates. Prior to 2011, the semantics of concurrent memory accesses in both languages had been specified by the Posix thread library [53], but in 2005 Hans Boehm noted [36] that when concurrency is described by a library, separate from the language specification, then there is a circularity between the two in the definition of the semantics. Following this observation, there was an effort to define the concurrency behaviour of C++ within the language specification. This thesis focuses on the relaxed memory model that was developed over the subsequent 6 years.

Programming language memory models Language designers have great freedom in their choice of memory model, and here we explore that design space. First we motivate the need for memory models, and then discuss minimal features that are necessary for concurrent programming. We go on to discuss the merits of models that provide strong ordering guarantees, and contrast them with the advantages of more relaxed models. Finally we discuss models that impose requirements on the programmer that are intended to provide some advantages of both strong and relaxed models.

A programming language might abstain from defining a memory model, and leave the language subject to the relaxed behaviour introduced by the target hardware and the optimisations of a particular compiler, but this damages program portability: different choices of compiler optimisations or a different series of processor within a common architecture may admit different behaviours. To be assured of the correctness of their code, a developer would have to test each supported configuration individually. Worse

still, relaxed-concurrency bugs can be sensitive to the execution environment and manifest with low probability, so this scheme mandates enormous testing resources. A well-defined memory model provides an abstraction of all of the various platforms that a program might execute above, and constrains their behaviour. It is then the responsibility of compiler writers and processor vendors to ensure that each platform meets the guarantees provided by the memory model. Defining a memory model enables the creation of tools that support portable concurrent programming, and avoids the need for interminable testing resources.

Attiya et al. showed that any concurrent programming language must include a minimal set of features in order to enable the programmer to construct consensus during the execution of their programs [19]; multiple threads must be able to agree on the state of a given piece of data. In a relaxed memory model, this can be an expensive operation: it requires some level of global synchronisation. On the hardware, the operations that provide this feature may be tightly linked to the hardware optimisations that they restrict, but in programming languages, the analogous constructs can be more intuitive. The specification of such features represents one axis in the design space.

The language memory model can provide strong ordering of memory accesses, or it can allow reordering. Strongly ordered memory models like *sequential consistency* (SC, see §2 for more details), where memory accesses are simply interleaved, have the advantage of usability: programmers need not consider intricate interactions of relaxed concurrent code — the most complex behaviour is simply forbidden. On the other hand, strong models force the compiler to emit code that implements the strong ordering guaranteed by the language. That may restrict the optimisations and force the introduction of explicit synchronisation in the emitted binaries, with a substantial overhead on modern multi-core processors.

At the other end of the spectrum, languages can provide a very relaxed memory model with the possibility of efficient implementation above relaxed processors, but this exposes the programmer to additional complexity. If the guarantees about the ordering of memory are too weak, then it can be impossible to build programs that implement reasonable specifications. Relaxed models can include explicit synchronisation features that allow the programmer to specify stronger ordering in parts of the program. This might take the form of mutexes, fences, barriers, or the memory-order annotations present in the example above. Given a relaxed model with these features, the programmer is burdened with the delicate task of inserting enough explicit synchronisation to ensure correctness, without introducing too much and spoiling performance.

Languages can provide a stronger model while maintaining efficient implementability by requiring a particular programming discipline. If programmers are required to avoid certain patterns, then their absence becomes an invariant for optimisation within the compiler. If a program fails to obey the discipline, then the language provides weaker guarantees about its behaviour.

All of these design decisions represent tradeoffs, and there is no universally superior approach; memory models should be designed in sympathy with the expected use of the programming language.

The C/C++ memory model This thesis focuses on the memory model shared by C and C++. Not only are C and C++ extremely well-used languages, but they represent the state of the art in memory-model design for mainstream programming languages.

The C and C++ languages aspire to be portable, usable by regular programmers who require an intuitive setting, and suitable for expert programmers writing high-performance code. For portability, the language defines a memory model, and for performance that model is relaxed.

The model is stratified by the complexity of its features. In its simplest guise, the memory model provides an intuitive setting for those who write single-threaded programs: the order of memory accesses is similar to that provided by previous sequential versions of the language. For programmers who want to write concurrent programs, there is extensive support provided in the concurrency libraries. This ranges from locks and unlocks, to the atomics library, that provides a low-level high-performance interface to memory.

The C/C++11 memory model design was strongly influenced by the work of Adve, Gharachorloo and Hill [12, 10, 11], who proposed a memory model whose programming discipline dictates that programs must annotate memory accesses that might take part in *data races*: two accesses on different threads that concurrently contend on the same piece of data. Following this work, in 2008 [37], Boehm and Adve described a simplified precursor of the C/C++11 memory-model design, imposing a similar programming discipline: programmers must declare objects that might be accessed in a racy way, these objects must be accessed only through the atomics library, and data races on all other objects must be avoided. If this discipline is violated in any execution of the program, then every execution has *undefined behaviour*. This is called a "catch-fire semantics" because programs with undefined behaviour are free to do anything — catch fire, order a thousand pizzas, email your resignation, and so on. This design choice carries a heavy cost to the usability of the language. Suppose a programmer identifies buggy behaviour in part of their program, and would like to debug their code. The program may be behaving strangely because of a race in a completely different part of the program, and this race may not even have been executed in the buggy instance. Debugging such a problem could be very difficult indeed. Note that this model of system programming does not match practice, where programmers try to understand racy programs in terms of an assumed model of the system comprising the compiler and the details of the underlying hardware. In this (unsanctioned) model of the system it is possible to debug racy programs by observing their behaviour, unlike in C/C++11.

Following earlier C++ design discussions [38, 35], Boehm and Adve provided a criteria under which programs executed in their relaxed memory model behave according

to sequential consistency [37], and this became a design goal of the C/C++11 memory model: programs that do not have any un-annotated data races, and that avoid using the lowest-level interface to memory, should execute in a sequentially consistent manner. This provides programmers who do not need to use the highest-performance features with an intuitive memory model (for race-free programs). The guarantee went further, stating that races can be calculated in the context of the sequentially-consistent memory model, rather than in the far more complex setting of the relaxed memory model. This is a powerful simplification that allows some programmers to be shielded from the full complexity of the memory model, while experts have access to high-performance features. Although, in early drafts of the C/C++11 standards, this laudable design goal was compromised (details in Chapter 5), the ratified language does provide this guarantee, as we show in Chapter 6.

The atomics library The atomics library provides versions of commonly used primitive data structures, like fixed-width integers, that can be used to write well-defined racy code. Accessor functions are used to read and write atomic variables. The C11 syntax for some of these is given below:

```
atomic_load_explicit(&x,memory_order)
atomic_store_explicit(&x, v, memory_order)
atomic_compare_exchange_weak_explicit(&x, &d, v, memory_order, memory_order)
```

The memory order argument decides how much ordering the access will create in an execution. There are six choices of memory order:

MEMORY_ORDER_SEQ_CST,
MEMORY_ORDER_ACQ_REL,
MEMORY_ORDER_ACQUIRE,
MEMORY_ORDER_RELEASE,
MEMORY_ORDER_CONSUME, and
MEMORY_ORDER_RELAXED.

This list is roughly in order, from strong to weak and expensive to cheap: MEMORY_ORDER_SEQ_CST can, under certain circumstances, provide sequentially-consistent behaviour with a substantial cost to performance, whereas accesses given MEMORY_ORDER_RELAXED exhibit many relaxed behaviours, but enable one to write very high-performance code. Typical concurrent programmers should use the former, whose behaviour is relatively straightforward, and expert programmers can use the whole gamut

of memory orders for fine-grained control over the ordering of memory accesses. The C/C++11 memory model allows a superset of the relaxed behaviour allowed by its target architectures. By choosing stronger memory orders, one can forbid this relaxed behaviour.

1.1 Focus of this thesis

The C and C++ memory models are defined by the International Standards Organisation (ISO) in two lengthy standard documents [30, 8]. Prior to my work, there were drafts describing the C/C++11 memory model, but those drafts, despite careful crafting by experts, were not known to describe a usable language memory model. The prose specifications were untestable, and the model was not well understood. It was not formally established whether the design was implementable, programmable, concise, or even internally consistent, nor had the central design tenets, laid out early in the design process [38, 35] and reiterated by Boehm and Adve [37], been established.

In my work, I have sought to understand the C/C++11 memory model in formal terms, to fix parts that were broken, to prove that the design is usable, and, where fixing problems was not yet possible, to highlight outstanding issues. In this thesis I assess the C/C++11 memory model design, presenting a clear and complete picture of a mainstream programming-language relaxed memory model. This effort both improved the C/C++11 definition and can inform the design of future programming-language memory models.

1.2 Contributions

Chapter 3 describes a formal version of the C/C++11 memory model that was developed in close contact with the standardisation committee. Work on this model fed corrections back to the language specification, and as a consequence, it is very closely in tune with the intention of the committee, and the ratified prose specification. The formal model is written in the specification language Lem [85, 90], and is readable, precise and executable (the full definitions are provided in Appendix C). The features of the model are introduced in stages through a series of cut-down models that apply to programs that do not use all of the language features. This chapter also presents a simplified model omits a redundant part of the specification. This work was developed in discussion with Scott Owens, Susmit Sarkar, and Peter Sewell, but I played the leading role. It was published in POPL in 2011 [28].

Chapter 4 describes Cppmem, a tool that takes very small programs and calculates all of the behaviours allowed by the memory model. Cppmem is joint work with Scott Owens, Jean Pichon, Susmit Sarkar, and Peter Sewell. I contributed to the initial design of the tool, and the tool uses an automatic OCaml translation of my formal memory model produced by Lem. Cppmem is invaluable for exploring the behaviour of the mem-

ory model. It has been used for communication with the ISO standardisation committee, for teaching the memory model to students, and by ARM, Linux and GCC engineers who wish to understand C/C++11. CPPMEM was described in POPL in 2011 [28], and an alternative implementation of the backend that used the Nitpick counterexample generator [31] was published in PPDP in 2011 [32], in work by Weber and some of the other authors.

Chapter 5 describes problems found with the standard during the process of formalisation, together with solutions that I took to the C and C++ standardisation committees. Many amendments were adopted by both standards in some form. This achievement involved discussing problems and drafting text for amendments with both my academic collaborators and many on the standardisation committee, including: Hans Boehm, Lawrence Crowl, Peter Dimov, Benjamin Kosnik, Nick Maclaren, Paul McKenney, Clark Nelson, Scott Owens, Susmit Sarkar, Peter Sewell, Tjark Weber, Anthony Williams, and Michael Wong. Some of these problems broke the central precepts of the language design. My changes fix these problems and are now part of the ratified standards for C11 and C++11 [30, 8], as well as the specification of the GPU framework, OpenCL 2.0 [86]. This chapter ends by identifying an open problem in the design of relaxed-memory programming languages, called the "thin-air" problem, that limits the compositionality of specifications, and leaves some undesirable executions allowed that will not appear in practice. This leaves the memory model sound, but not as precise as we would like. Many of the comments and criticisms were submitted as working papers and defect reports [29, 20, 75, 73, 111, 27, 76, 77, 74].

Chapter 6 describes a mechanised HOL4 proof that shows the equivalence of the progressively simpler versions of the C/C++11 memory model, including those presented in Chapter 3, under successively tighter requirements on programs. These results establish that a complicated part of the specification is redundant and can simply be removed, and they culminate in the proof that the specification meets one of its key design goals (albeit for programs without loops or recursion): despite the model's complexity, if a race-free program uses only regular memory accesses, locks and SEQ_CST-annotated atomic accesses, then it will behave in a sequentially consistent manner. This proof validates that the model is usable by programmers who understand sequential consistency.

Chapter 7 describes work done in collaboration with Jade Alglave, Luc Maranget, Kayvan Memarian, Scott Owens, Susmit Sarkar, Peter Sewell, Tjark Weber and Derek Williams. We took the compilation mappings from C/C++11 to the x86, Power and ARM architectures that had been proposed by the C++11 design group and proved that they do indeed preserve the semantics of the programming-language memory model in execution above those processors. This led to the discovery and resolution of a flaw in one of the mappings. This chapter represents a second form of validation of the formal model: it is implementable above common target architectures. My contribution, which was smaller in this work, involved proving equivalent variants of the C/C++11 memory

model that more closely matched the processors, and discussion of cases in the proof of soundness of the mappings. This work was published in POPL 2011, POPL 2012 and PLDI 2012 [26, 96, 28].

Chapter 8 describes work with Mike Dodds and Alexey Gotsman, developing a compositional reasoning principle for C/C++11. This takes the form of an abstraction relation between a library specification and its implementation. If the specification abstracts the implementation, then we show that the behaviour of a program consisting of arbitrary client code calling the library implementation is a subset of the behaviour of the same client calling the specification. We use this abstraction theorem to prove that an implementation of a concurrent data structure, the Treiber stack, meets its specification. This is another form of validation of the formal model: one can reason about programs with it. My contribution involved writing the implementation and specification of the Treiber Stack and producer-consumer examples, proving that the Treiber Stack specification abstracts its implementation, and contributions to the proof of soundness of the abstraction relation. This work was published in POPL 2013 [25].

Appendix A presents a key piece of evidence that validates the formal model: a side-by-side comparison of the C++11 standard text [30] and the formal memory model presented in Chapter 3, establishing a tight correspondence between the two. Appendix A follows the text of the standard, and is suited to those more familiar with the text. Appendix B presents the same link, following the structure of the model.

Together, this work represents the refinement and validation of a mainstream programming-language memory model. My work establishes that one can write high-performance programs with sound specifications in C/C++11, and that those programs can be correctly compiled to common processor architectures. In the broader context, this work provides a critical analysis of C/C++11 as an example of a relaxed-memory programming language, and identifies design goals for future memory models. The work serves as an example of the benefit of using rigorous semantics in language design and specification.

1.3 Related work

The beginning of this chapter explained that the effort to define a memory model within the C/C++11 specification was motivated by the work of Boehm, who observed that the compiler can interfere with the semantics of relaxed concurrency primitives if they are specified separately from the language [36].

The definition of the C/C++11 memory-model design borrows many concepts from earlier work. The sequential consistency that C/C++11 provides to some programs was first described by Lamport [60]. The combination of relaxed behaviour and explicit programmer-declared synchronisation was a feature of the weak ordering described

by Dubois et al. [48]. C/C++11 coherence matches the coherence property of Censier and Feautrier [42] The compare and swap feature of C/C++11 follows the IBM 370 instruction [44]. C/C++11 provides racy programs with undefined behaviour, a concept borrowed from Adve, Gharachorloo and Hill [12, 10, 11] who defined memory models with stronger guarantees for race-free programs. The C/C++11 memory model builds upon a simple precursor model, defined by Boehm and Adve [37], who expressed the high-level design intent of the memory model (that race-free programs using only the SC atomics should behave in an SC manner), and proved this property of their memory model.

The C/C++11 memory model is expressed in an *axiomatic* style: the model is made up of a predicate that decides whether a particular whole execution is allowed for a given program, or not. There are several examples of early axiomatic memory models, by Collier [43], by Kohli et al. [13], and by Adve et al. [12, 10]. Contrast this with *operational* memory models, where the model is described as an abstract machine, with a state made up of buffers and queues. Many formal hardware memory models adopt the operational style [97, 71, 99, 91, 104], because hardware architecture specifications are often described in terms of an abstract machine.

There are many formal memory models of hardware architectures. Sarkar et al. created an operational formalisation of the x86 architecture's memory model [99], following the incomplete and ambiguous published specification documents of the time. This model was superseded by the operational x86-TSO model of Owens et al. [91, 104], which is easy to understand and is validated both by discussion with Intel and by hardware testing. We describe this model in Chapter 2, and refer to it throughout this thesis. In a suggested extension to the architecture, Rajaram et al. propose a hardware-optimised version of C11 read-modify-write operations on x86, including an alternative compilation scheme that preserves the semantics of the language over systems using the optimised variants [93]. For the x86-TSO memory model, Owens provides a stronger alternative to the typical SC-if-data-race-free guarantee, introducing triangular-race-freedom [89].

There are several formal Power memory models to note. Chapter 2 outlines the operational model of Sarkar et al. [97, 71] that was developed together with Williams, a leading processor designer at IBM, and was systematically tested against hardware. This model can claim to match the architectural intent of the vendor. The axiomatic models of Alglave et al. [14] are informed by systematic testing of hardware, with tests generated by the Diy tool [16], executed on current hardware with the Litmus tool [17], and executed according to the model with the Herd tool [18]. This systematic testing led to the discovery of a bug in the Power 5 architecture [16]. Mador-Haim et al. present an axiomatic model [68] that is intended to be abstract and concise, while matching the relatively intricate model of Sarkar et al.

There is an earlier body of work on the Java memory model (JMM), another relaxed-memory language, with a rather different design (discussed in Section 5.10.3). Manson et al. provided a formal description of the official JMM [70]. Cenciarelli et al. provided a

structured operational semantics for the JMM and proved that it is correct with respect to the language specification [41]. Huisman and Petri explore the JMM design and provide a formalisation in Coq [52]. Lochbihler provides a mechanised formal specification of the JMM in Isabelle/HOL, extended to cover a more complete set of features [66, 67]. The Java memory model is intended to admit compiler optimisations, and forbid thin-air behaviour. Ševčík and Aspinall analysed this model and showed that it fails to admit compiler optimisations that are performed by the Hotspot compiler, leaving the model unsound over one of the key compilers for the language [101]. Demange et al. describe an effort to retrofit Java with a buffered memory model similar to that of x86-TSO [46].

There has been some work on compilation of relaxed-memory languages. Ševčík et al. extended Leroy's verified compiler, Compcert [63], to a relaxed-concurrent variant of C with a TSO memory model, in CompcertTSO [102, 103]. Morisset et al. made a theory of sound optimisations over the C/C++11 memory model (as formalised in this thesis), tested compilers and found several bugs in GCC [84].

There have been several formalisations of other parts of C and C++. Norrish provided a mechanised formalisation of C expressions, establishing that a large class of them are deterministic [88]. Ramananandro et al. formalised object layout in C++ and proved several object-layout optimisations correct [94, 95]. Ellison presented a formal thread-local semantics for C [49]. Krebbers formalised C11 dynamic typing restrictions in Coq [59]. Klein et al. provided a formal machine-checked verification of the single-threaded seL4 microkernel [58, 57]. A series of papers by Palmer et al. [92] and Li et al. [64, 65] present a formal specification for a subset of MPI, the high-performance message-passing-concurrency API. None of these addresses shared-memory concurrency.

One line of work has attempted to provide the programmer with a strongly-ordered concurrency model while maintaining the performance of a relaxed model. Sasha and Snir [105] propose recognising dependency cycles in a graph of program segments, and using this analysis to add delay instructions that provide sequential consistency. Gotsman et al. enable a relaxed implementation to hide behind a strongly-ordered interface: they present a variant of linearisability that can be used to show that a library written above the x86-TSO memory model matches a specification in an SC model [50]. To a similar end, Jagadeesan et al. establish an abstraction theorem that allows one to provide sequential specifications to code written above the SC, TSO, PSO and Java memory models [56]. Marino et al. quantify the cost of preserving an SC programming model in the compiler by altering internal LLVM passes to preserve SC, and measuring the slowdown [72]. Alglave et al. provide the Musketeer tool, that performs a scalable static analysis, and automatically inserts fences in order to regain a strong memory model [15].

There are several approaches to the specification and verification of concurrent code in the literature. Schwartz-Narbonne et al. provide a concurrent assertion language for an SC memory model [100]. Burckhardt et al. define linearisability over the TSO memory model [40].

There has been some work on verification for C/C++11. In each piece of work, the presence of thin-air values (discussed in Chapter 5) forces a compromise. Vafeiadis and Narayan presented Relaxed Separation Logic (RSL) for reasoning about C11 programs, and proved it sound in Coq over an augmented C++11 memory model that includes a very strong restriction on thin-air values [110]. Turon et al. present Ghosts, Protocols, and Separation (GPS), a program logic, that can also be used to reason about C11 programs that use release and acquire atomics [109]. Norris and Demsky present CDSchecker, a tool for exhaustive checking of executions of C11 programs [87]. Their tool incrementally executes programs, and will miss executions that feature thin-air behaviour.

There have been several C11 implementations of concurrent data structures. Lê et al. implement an efficient concurrent FIFO queue in C11, test its performance over several hardware architectures, and prove that the code executes as a FIFO queue [61]. With different authors, Lê et al. provide an optimised C11 implementation of Chase and Lev's deque [62].

Chapter 2

Background

This chapter includes a brief introduction to three memory models: sequential consistency, the x86 memory model, and the Power memory model (ARM is similar to Power, SPARC-TSO is similar to x86). Sequential consistency is a strong model that is considered to be usable by regular programmers. The x86, Power and ARM architectures (together with Itanium, that we do not consider in detail) represent the most important targets of the C/C++11 language, so the relaxed behaviour that they allow is key to the C/C++11 memory model design. The chapter goes on to introduce the interaction of compiler optimisations with the relaxed memory model, finishing with an overview of the C/C++11 memory model design and the process of its definition.

2.1 Sequential consistency

The simplest design choice one might make for a multi-core system would be to imagine that all memory accesses across all cores are interleaved, and belong to a total order. Each memory read would then get the value of the immediately preceding write to the same location in the total order. This memory model is called *sequential consistency* (SC) and was first articulated as such by Lamport [60].

To understand the flavour of sequential consistency, consider the following example (written in C-like pseudocode). A parent thread initialises two integer locations in memory, `x` and `y`, each to the value 0 and then creates two threads: one that writes 1 to `x` and then reads from `y`, and another that writes 1 to `y` and then reads from `x`. Variables `r1` and `r2` are used only to identify the outcome of the loads, and it is convenient to ignore the memory effects of the writes to each. The example below presents the program using a double bar to indicate the parallel composition of the two child threads, and uses layout to indicate the parent thread above them.

```
   int x = 0;
   int y = 0;
x = 1;  ||  y = 1;
r1 = y; ||  r2 = x;
```

The SC memory model permits any interleaving of the accesses to memory that agrees with the order apparent in the source text of the program, that is: thread-local program order and parent-to-child thread order. The following table lists all possible interleavings of the accesses to memory alongside the resulting outcomes for the values of `r1` and `r2`:

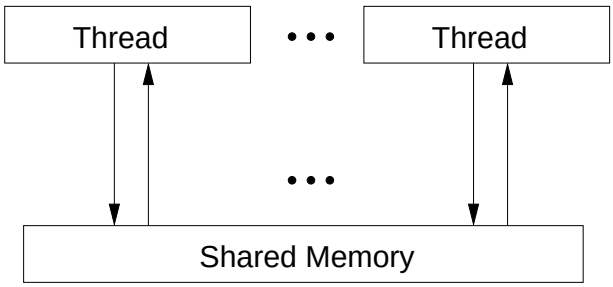
Interleaving	Outcome

`x = 0; y = 0; x = 1; y = 1; r1 = y; r2 = x;`
`x = 0; y = 0; y = 1; x = 1; r1 = y; r2 = x;`
`x = 0; y = 0; x = 1; y = 1; r2 = x; r1 = y;`
`x = 0; y = 0; y = 1; x = 1; r2 = x; r1 = y;`
	`r1 = 1, r2 = 1`

`x = 0; y = 0; x = 1; r1 = y; y = 1; r2 = x;`	`r1 = 0, r2 = 1`
`x = 0; y = 0; y = 1; r2 = x; x = 1; r1 = y;`	`r1 = 1, r2 = 0`

Three outcomes are possible for `r1` and `r2`: 1/1, 0/1, and 1/0. The program above is a *litmus test* — a small program, used to test whether a memory model exhibits a particular non-SC memory behaviour.

The SC memory model can be thought of as an abstract machine consisting of a single shared memory serving a set of threads, each of which can take steps writing or reading the memory. The following diagram represents this abstract machine:

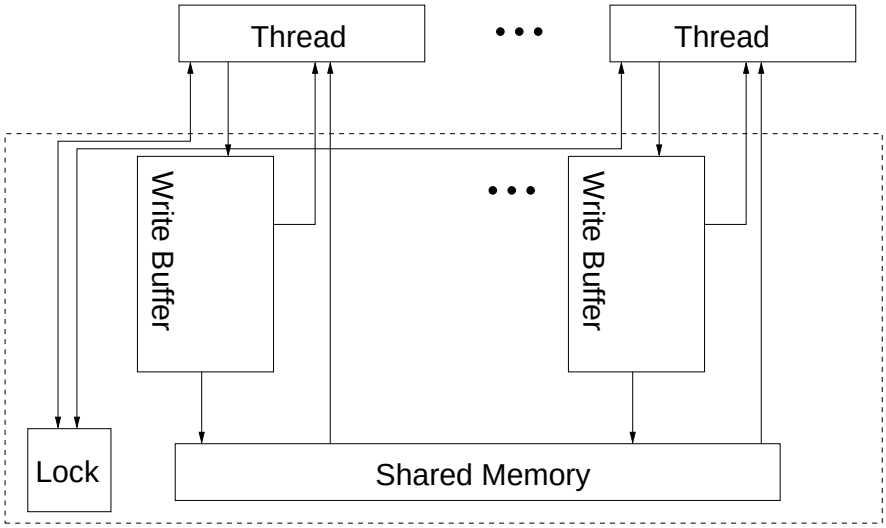


Any memory model that admits behaviour that is not allowed by the SC model, the outcome 0/0 in the test above for instance, is said to be *relaxed*. Real processors include intricate optimisations of memory that involve caching and buffering. Maintaining a fully sequentially consistent interface for the programmer would require hiding such optimisations, ultimately reducing performance, so processor designers allow some non-sequentially consistent behaviour to be seen. The outcome 0/0 is allowed for an analogous program on x86, Power and ARM processors, where optimisations in the memory subsystem can delay the write on each thread from reaching the other until the reads have read from the initialisation write. The following sections introduce two common processor architectures: x86 and Power, sketching the details of the memory systems that are made visible to the programmer.

2.2 x86 and Dekker's algorithm

The x86 model described here is that of Scott Owens, Susmit Sarkar, and Peter Sewell [91, 104]. The memory model has been validated both by extensive discussion with experts, and by testing on hardware. It covers a useful subset of x86 features, but it ignores non-temporal accesses. This work highlighted deficiencies in the processor architecture documentation, and Intel released further documents that fixed the problems.

For the sake of performance, the x86 memory model makes some of the details of internal optimisations visible in the form of out-of-order memory accesses. The memory model is best understood as a small change to the SC-abstract machine that was presented in the previous section. Each thread gains a first-in-first-out buffer that temporarily holds writes from that thread before they are flushed to (and become visible in) shared memory. There is also a global lock that can be used to coordinate operations that atomically read and write memory. The abstract machine is as follows:



Execution is still step based: a thread can read or write, as in the SC abstract machine, but now the memory subsystem can take a step too — it can flush a write from the end of one of the threads' write buffers to memory. When a thread writes, the write is added to the thread-local write buffer and leaves the memory unaffected. When a thread reads, it must read the most recent value for the variable present in the thread-local write buffer, if there is one. Only if there is no such write in the buffer does it read from main memory.

Store buffering on x86 makes behaviours observable that would not have been observable on the SC memory model. Recall the example program from the previous section:

```
   int x = 0;
   int y = 0;
x = 1; || y = 1;
r1 = y; || r2 = x;
```

On the SC memory model this program had three possible outcomes for the values of `r1` and `r2`: 1/1, 0/1, and 1/0. In x86 a new outcome is possible: 0/0. To understand how, first label the Threads 0, 1 and 2, for the parent, left hand and right hand threads respectively. The label, `t:flush;`, will be used to describe when the memory system flushes a write from the end of a buffer on thread `t`. All other accesses to memory will be similarly labeled with the thread that performs them. Now imagine that the x86 abstract machine executes according to the sequence given in the left hand column of the table below. There are columns representing the contents of the write buffers on each thread, and a column representing the contents of memory. Recall that `r1` and `r2` are just used to identify the values of reads, so their memory behaviour is elided.

Step	Write buffer			Memory
	Thread 0	Thread 1	Thread 2	
`0:x = 0;`	`x = 0`			
`0:flush;`				`x=0`
`0:y = 0;`	`y = 0`			`x=0`
`0:flush;`				`x=0, y=0`
`1:x = 1;`		`x = 1`		`x=0, y=0`
`2:y = 1;`		`x = 1`	`y = 1`	`x=0, y=0`
`1:r1 = y;`		`x = 1`	`y = 1`	`x=0, y=0`
`2:r2 = x;`		`x = 1`	`y = 1`	`x=0, y=0`
`1:flush;`			`y = 1`	`x=1, y=0`
`2:flush;`				`x=1, y=1`

When Thread 1 reads from `y` into `r1`, there is no write to `y` in the write buffer of Thread 1, so Thread 1 reads from memory. The write of `y` on Thread 2 has reached its thread-local write buffer, but has not yet been flushed to memory. Consequently, Thread 1 reads the value 0 from memory. The read of `x` into `r2` takes value 0 symmetrically. This sequence gives rise to the values `r1 = 0` and `r2 = 0`, an outcome that was impossible in the sequentially-consistent memory model. This behaviour, called *store-buffering*, is produced by keeping the write on each thread in its respective write buffer until the reads have completed, and only then flushing the writes to memory. This non-SC behaviour means that x86 is a relaxed memory model.

Note that a read must read its value from the most recent write to the same location in its thread-local write buffer if one exists, so a thread can read its own writes before they become visible to other threads.

The consequences of store-buffering The relaxed behaviour allowed by the x86 architecture can make programs that are correct in the SC memory model incorrect. Take as an example Dekker's algorithm [47], that provides mutual exclusion between threads

over a critical section. Although it is not a well-used mutual-exclusion mechanism in practice, it neatly illustrates the impact of allowing store-buffering relaxed behaviour. Pseudocode for a version of this algorithm is given below:

```
1                    int flag0 = 0;
2                    int flag1 = 0;
3                     int turn = 0;
4   flag0 = 1;                 ||  flag1 = 1;
5   while(flag1 == 1) {        ||  while(flag0 == 1) {
6     if (turn == 1) {         ||    if (turn == 0) {
7       flag0 = 0;             ||      flag1 = 0;
8       while (turn == 1);     ||      while (turn == 0);
9       flag0 = 1;             ||      flag1 = 1;
      }                        ||    }
    }                          ||  }
                               ||
    ... critical section ...   ||  ... critical section ...
                               ||
10  turn = 1;                  ||  turn = 0;
11  flag0 = 0;                 ||  flag1 = 0;
```

On an SC memory model, this algorithm provides mutual exclusion. To enter the critical section, one thread declares that it will try to enter by writing to its flag variable. It then checks for contention by reading the other thread's flag variable. If there is no contention, then it enters the critical section. Otherwise, the thread engages in a turn-taking protocol, where its flag is written to zero, it waits for its turn, it writes 1 to its flag and then checks the flag on the other thread. If the other thread is contending, it will either pass into the critical section and, on exit, relinquish the turn to the other thread, or be blocked with its flag set to 0, because it is already the other thread's turn.

On the x86 memory model, relaxed behaviour can cause both threads to enter the critical section, breaking mutual exclusion. It is the store-buffering relaxed behaviour that gives rise to this outcome. Recall the shape of the store-buffering litmus test, and compare it to lines 1, 2, 4 and 5 projected out from the Dekker's algorithm example above:

```
   int x = 0;                          int flag0 = 0;
   int y = 0;                          int flag1 = 0;
x = 1;  ||  y = 1;          flag0 = 1;          ||  flag1 = 1;
r1 = y; ||  r2 = x;         while(flag1 == 1)... ||  while(flag0 == 1)...
```

Store-buffering allows both threads to write 1 to their own flag, read the other thread's flag as 0, and then enter the critical section, breaking mutual exclusion.

The x86 architecture provides the facility to flush the thread-local store buffer by adding explicit barriers. In the abstract machine, when a thread encounters an `MFENCE` barrier, it must wait until its thread-local write buffer has been emptied before continuing with the instructions following the `MFENCE`. If we augment the store-buffering litmus test with `MFENCE`s, it becomes:

```
   int x = 0;
   int y = 0;
x = 1;   || y = 1;
MFENCE;  || MFENCE;
r1 = y;  || r2 = x;
```

Now the sequence of steps in the abstract machine that led to the relaxed behaviour is no longer allowed. The writes to `x` and `y` cannot remain in their respective write buffers: before we perform the read on each thread, we will encounter an `MFENCE`, and must flush the write buffer to memory first, making the relaxed behaviour impossible. Inserting fences into Dekker's algorithm does indeed reestablish mutual exclusion.

Global lock The x86 semantics has a global lock that can be used to atomically read and then write a location in memory. This ability is essential for establishing consensus across multiple threads, and enables us to implement a compare-and-swap primitive in C/C++11. If any processor has acquired the global lock, then only that processor may read from memory until it is released, and on release, that processor's store buffer is flushed.

2.3 Even more relaxed: Power and ARM

This section describes the Power memory model of Sarkar et al. [98, 96]. It was developed in close communication with IBM processor architect Williams. The memory model has been validated by extensive discussion with experts, and by testing on hardware. The Power and ARM architectures have similar memory models to one another, and each is more relaxed than that of x86. This section will provide an informal introduction to the Power model, and some of the relaxed behaviour that can be exhibited by it.

We return to the message-passing litmus test that was introduced in Chapter 1, using variables `x` and `y` for the data and flag, respectively. This test models a programming idiom where one thread writes some data, here `x`, then writes to a flag variable, here `y`. The other thread checks the flag variable, and if the flag has been written, it expects to read the data written by the writing thread.

```
      int x = 0;
      int y = 0;
x = 1;  || while (y <> 1){}
y = 1;  || r = x;
```

Executions of this program might read from y any number of times before seeing the value 1, or they may never see the value 1. We focus on executions where the first read of y reads the value 1, so we simplify the test to remove the loop:

```
   int x = 0;
   int y = 0;
x = 1; || r1 = y;
y = 1; || r2 = x;
```

In the sequentially consistent memory model, the behaviour of the program is given by the set of all interleavings of the memory accesses, as presented in the table below:

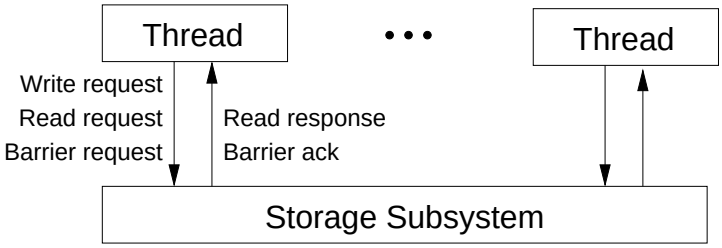
Interleaving	Outcome
x = 0; y = 0; x = 1; y = 1; r1 = y; r2 = x	r1 = 1, r2 = 1
x = 0; y = 0; x = 1; r1 = y; y = 1; r2 = x x = 0; y = 0; r1 = y; x = 1; y = 1; r2 = x x = 0; y = 0; x = 1; r1 = y; r2 = x; y = 1 x = 0; y = 0; r1 = y; x = 1; r2 = x; y = 1	r1 = 0, r2 = 1
x = 0; y = 0; r1 = y; r2 = x; x = 1; y = 1	r1 = 0, r2 = 0

Note that the outcome 1/0 is not allowed under sequential consistency or x86 — the first-in-first-out nature of the write buffers makes it impossible to see the second write without the first already having flushed to memory. On Power, the relaxed behaviour could be introduced by any of the following three architectural optimisations:

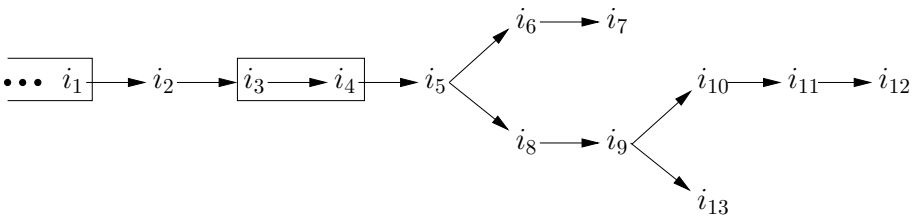
- The left hand thread's writes might be committed to memory out of order, by reordering the stores.
- The right hand thread's reads might be performed out of order by some speculation mechanism.
- The memory subsystem might propagate the writes to the other thread out-of-order.

The Power and ARM architectures expose all three sorts of reordering mentioned above, and produce the result 1/0 on the message-passing example.

The Power and ARM abstract machine Sarkar et al. define the Power and ARM architecture memory model as an abstract machine. This machine is split between thread-local details such as speculation, and memory-subsystem details such as write propagation. Threads can make write, read and barrier requests, and the memory subsystem can respond with barrier acknowledgements and read responses. Read requests, rather than containing a simple value, are associated with a particular write that is identified by a read-response event. Each read-request from a thread results in an immediate read-response from the storage subsystem.

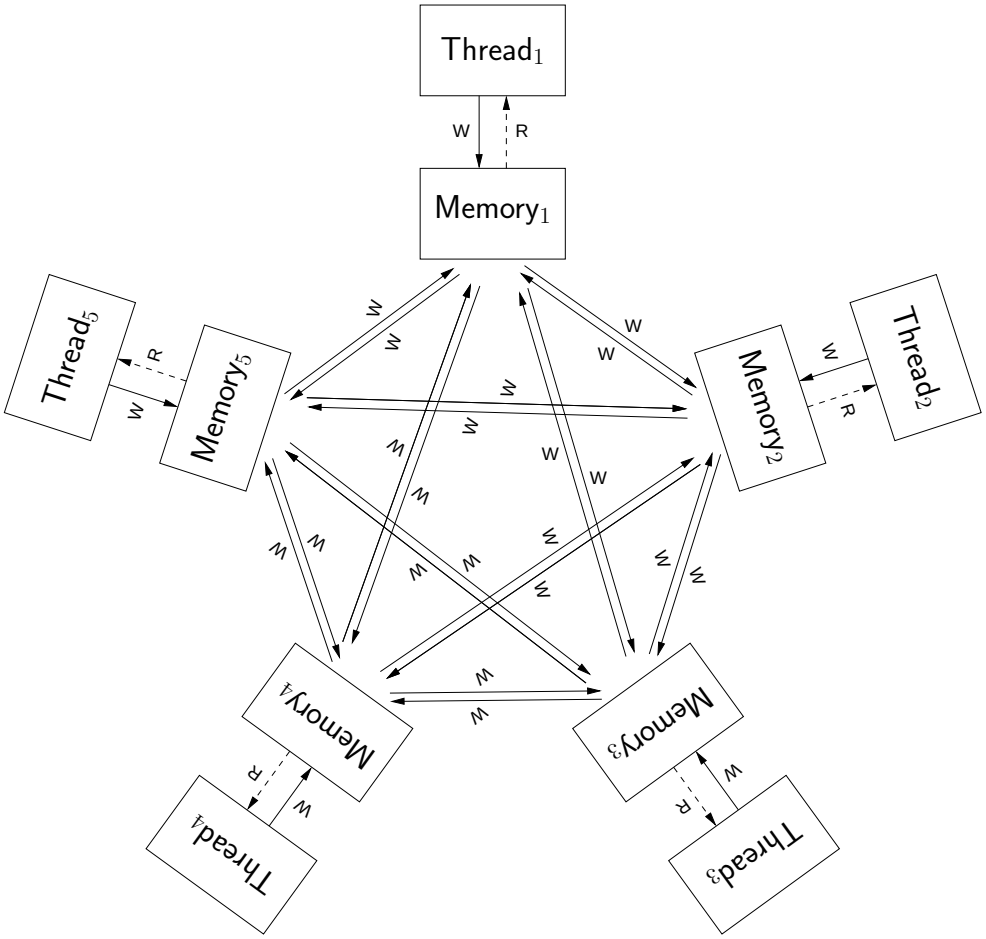


Each thread is represented by an abstract machine that maintains a tree of all possible instructions that the thread might perform, with branches in the tree corresponding to branches in the control flow of the program. Initially, all instructions are marked as *in flight*. Instructions are then *committed* one by one, not necessarily in program order. Instructions cannot be committed beyond a branch — the branch itself must be committed first, and the untaken subtree is discarded. Some instructions can be processed before being committed; this allows the speculation of reads (but not writes) beyond uncommitted control flow branches. The following diagram (taken from the tutorial of Maranget et al. [71]) shows an instruction tree, with committed instructions boxed.



In the message-passing example above (without the while loop), there is no branch between the read of `y` and the read of `x`, so this mechanism can commit the write of `x` before the write of `y`, or the read of `x` before the read of `y`, producing the relaxed behaviour in either case.

Propagation lists For each thread, the abstract machine's storage subsystem maintains a list of the writes and barriers that have been propagated to the thread. Intuitively, the propagation list keeps the set of global events that a thread has observed so far. The storage subsystem can propagate an event from one thread's propagation list to another thread's list at any time, with some caveats. A thread can also commit a write to its own propagation list at any time, again with some caveats. Read requests are satisfied with a write from the thread's propagation list. The diagram below (taken from the tutorial of Maranget et al. [71]) depicts these paths of communication between threads and their propagation lists $Memory_1$ through $Memory_5$:



Propagation order is constrained by barriers and coherence (discussed below), but in the message-passing example there are neither, so the write of `x` can be propagated to the other thread before the write of `y`, permitting the relaxed behaviour.

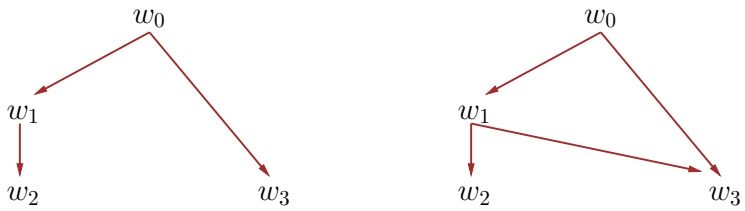
Coherence The X86, Power and ARM processors are all *coherent*: the writes to a single location are globally ordered and all reads of the location must see values that are consistent with the order. On x86 writes become visible to all threads when they reach main memory, and the order that writes reach memory orders the writes at each location. So far, there is no analogous order in the Power model.

To ensure coherence, the storage subsystem of the Power abstract machine keeps a *coherence-commitment order* for each location. This is a partial relation that records the global order in which writes are committed, so that at the end of the execution it will be total. The order is constructed thread-wise, built up from the order in which writes are propagated to each thread.

Read requests can be issued by any thread at any time. Reads are requested and committed separately, and read requests can be invalidated. The propagation list of the issuing thread, and indirectly, the coherence-commitment order, restrict the write that the storage subsystem can return for a given read request. The storage subsystem must return the most recent write of the location in the thread's propagation order, and the

coherence-commitment order restricts which writes can be committed to the propagation list.

Take for example (from Sarkar et al. in PLDI 2011 [98]) an abstract-machine storage subsystem with the coherence-commitment order as depicted on the left below. Suppose we are interested in one particular thread that is about to perform a read request, and w_1 is the most recent write to the location of the read in the thread's propagation list. The storage subsystem must return the most recent write to the location in the thread's propagation list, so the write returned is decided by which writes may be propagated to the thread before the read request. Consider each of the cases in turn:



1. The coherence-earlier write, w_0, cannot be propagated to the read's thread, so the read cannot see w_0.

2. With no further propagation, the read could see w_1.

3. The reading thread could be propagated w_2, which would then be seen by the write, leaving coherence order unchanged.

4. The reading thread could be propagated w_3, adding w_3 to the propagation list, and causing it to be seen by the read. This has the effect of adding an edge to the coherence order from w_1 to w_3, but leaves w_2 and w_3 unordered. The updated coherence graph is on the right above.

Message passing: avoiding relaxed behaviour Now that the Power abstract machine has been introduced, we return to the message-passing example, and the details of the machine that can give rise to the relaxed behaviour:

```
  int x = 0;
  int y = 0;
x = 1; || r1 = y;
y = 1; || r2 = x;
```

The three causes of observing the outcome 1/0 in this program were as follows:

- Writes can be committed out of order.
- Writes can be propagated out of order.
- Reads can be requested out of order.

It is easy to imagine programs where the programmer intends to use the message-passing idiom and the relaxed behaviour would be undesirable, so, much in the same way that x86 provided the `MFENCE`, Power and ARM provide additional ordering through barriers and *dependencies*. There are three sorts of dependency from a read, r, to another accesses of memory, a: if the value of r is used to compute the address of the access a then there is an *address* dependency, if the value of r is written to memory by a there is a *data* dependency, and if the control flow path that leads to a is selected by the value returned by r then there is a *control* dependency.

The following program augments the message-passing example, adding an `lwsync` on the writing thread, and wrapping the second read within an if-statement with an `isync` barrier, that enforces order of the read requests, so that it is dependent on the value read by the first:

```
        int x = 0;
        int y = 0;
x = 1;  || r1 = y;
lwsync; || if (r1 == 1) {
y = 1;  ||  isync;
        ||  r2 = x; }
```

Inserting an `lwsync` barrier between two writes in the Power architecture (or a `dmb` for ARM) prevents the writes from being committed or propagated out of order. Similarly, the control dependency added to the reading thread, combined with the `isync` barrier, prevents the reads from being requested out of order. So with these new additions, the relaxed behaviour is no longer allowed. The Power architecture also provides the `sync` barrier that is even stronger than the `lwsync` or a dependency with `isync`. With liberal use of `sync`, one can regain sequential consistency.

It is interesting to note that when programming such processors, one can rely on dependencies to provide ordering. It is sufficient to create an artificial dependency that has no real bearing on the execution of the program — the sort of dependency that a compiler would routinely remove. This turns out to be a useful programming idiom, and lies in stark contrast to the treatment of dependencies in programming languages. In the message-passing example above, we can create an artificial address dependency as follows:

```
        int x = 0;
        int y = 0;
x = 1;  || r1 = y;
lwsync; || r2 = *(&x+r1-r1);
y = 1;  ||
```

This dependency is sufficient to disable the thread-local speculation mechanism, and ensure the relaxed outcome 1/0 is not visible.

Multi-copy atomicity Some memory models, like x86 and SC, order all of the writes in the system by, for example, maintaining a shared memory as part of the state of the model. Other memory models are weaker and allow different threads to see writes across the system in different orders. Consider the following example, called IRIW+addrs for independent reads of independent writes with address dependencies:

```
                int x = 0;
                int y = 0;
x = 1; || y = 1; || r1 = y;            || r3 = x;
                    r2 = *(&x+r1-r1); || r4 = *(&y+r3-r3);
```

In this test, there are two writing threads and two reading threads, and the question is whether the two reading threads can see the writes in different orders; can the values of `r1`, `r2`, `r3` and `r4` end up being 1/0/1/0? On the Power and ARM architectures, this outcome would be allowed by the thread-local speculation mechanism if the dependencies were removed. With the dependencies in place, an unoptimised machine-instruction version of the test probes whether the Power and ARM storage subsystem requires writes across the system to be observed in an order that is consistent across all threads, a property called *multi-copy atomicity* [3]. Neither Power nor ARM are multi-copy atomic, so the outcome 1/0/1/0 is allowed.

Cumulativity Power and ARM provide additional guarantees that constrain multi-copy atomicity when dependencies are chained together across multiple threads. Consider the following example, called ISA2 [71], where the dependency to the read of `x` is extended by inserting a write and read to a new location, `z`, before the read of `x`. Note that there is still a chain of dependencies from the read of `y` to the write of `x`:

```
                    int x = 0;
                    int y = 0;
x = 1;   || r1 = y;       || r2 = z;
lwsync;  || z = 1+r1-r1;  || r3 = *(&x+r2-r2);
y = 1;   ||               ||
```

Here there are loads of `y`, `z` and `x`. The outcome 1/1/0 is not visible on Power and ARM because the ordering guaranteed by the `lwsync` is extended through the following dependency chain. Without the `lwsync` or the dependencies the relaxed behaviour 1/1/0 would be allowed: the writes on Thread 1 could be committed or propagated out of order, the instructions of Thread 2 or 3 could be committed out of order, or the writes of Threads 1 and 2 could be propagated out of order to Thread 3. With the addition of the barriers and dependencies, it is clear that Thread 1 and 2 must commit in order, and that Thread 1 must propagate its writes in order.

It is not yet obvious that the writes of Thread 1 must be propagated to Thread 3 before the write of Thread 2, however. A new guarantee called *B-cumulativity*, provides ordering through executions with chained dependencies following an `lwsync`. In this example, B-cumulativity ensures that the store of `x` is propagated to Thread 3 before the store of `z`.

The example above shows that B-cumulativity extends ordering to the right of an `lwsync`, *A-cumulativity* extends ordering to the left: consider the following example, called WRC+lwsync+addr for write-to-read causality [37] with an `lwsync` and an address dependency. Thread 1 writes `x`, and Thread 2 writes `y`:

```
                    int x = 0;
                    int y = 0;
x = 1; || r1 = x;  || r2 = y;
       || lwsync   || r3 = *(&x + r2 - r2);
       || y = 1;   ||
```

The `lwsync` and the address dependency prevent thread-local speculation from occurring on Threads 2 and 3, but there is so far nothing to force the write on Thread 1 to propagate to Thread 3 before the write of Thread 2, allowing the outcome 1/1/0 for the reads of `x` on Thread 2, `y` on Thread 3 and `x` on Thread 3. We define the *group A* writes to be those that have propagated to the thread of an `lwsync` at the point that it is executed. The write of `x` is in the group A of the `lwsync` in the execution of the program above. A-cumulativity requires group A writes to propagate to all threads before writes that follow the barrier in program order, guaranteeing that the outcome 1/1/0 is forbidden on Power and ARM architectures; it provides ordering to dependency chains to the left of an `lwsync`.

Note that in either case of cumulativity, if the dependencies were replaced by `lwsync`s or `sync`s, then the ordering would still be guaranteed.

Load-linked store-conditional The Power and ARM architectures provide *load-linked* (LL) and *store-conditional* (SC) instructions that allow the programmer to load from a location, and then store only if no other thread accessed the location in the interval between the two. These instructions allow the programmer to establish consensus as the global lock did in x86. The load-linked instruction is a load from memory that works in conjunction with a program-order-later store-conditional. The store-conditional has two possible outcomes; it can store to memory, or it may fail if the coherence commitment order is sufficiently unconstrained, allowing future steps of the abstract machine to place writes before it. On success, load-linked and store-conditional instructions atomically read and then write a location, and can be used to implement language features like compare-and-swap.

Further details of the Power and ARM architectures This brief introduction to the Power architecture is sufficient for our purposes, but it is far from complete. There are further details that have not been explained: speculated reads can be invalidated, for instance. A more complete exploration of the Power memory model and many more litmus tests can be found in Maranget, Sarkar, and Sewell's tutorial on the subject [71].

ARM processors allow very similar behaviour to the Power memory model, but ARM implementations differ in micro-architectural details so that the Power model does not closely resemble them.

2.4 Compiler optimisations

In addition to the relaxed behaviour allowed by the underlying processors, language memory models must accomodate relaxed behaviour introduced by optimisations in the compiler, and optimisations must be sensitive to the memory model. In this section, we discuss three interactions of compiler optimisations with the memory model: introducing reordering by applying a local optimisation, introducing reordering by applying a global optimisation, and an optimisation that is only allowed in a relaxed memory model.

A local optimisation Consider the following example of a thread-local optimisation. A function `f`, on the left below, writes to a memory location pointed to by `b` regardless of the value of `a`. The compiler should be free to optimise this to the code on the right that simply writes, without the conditional.

```
void f(int a, int* b) {
  if (a == 1)
    *b = 1;
  else
    *b = 1;
}
```

```
void f(int a, int* b) {
  *b = 1;
}
```

It is interesting to note that in the unoptimised code, the store to the location of `b` appeared, at least syntactically, to depend on the value `a`. In the analogous program on the Power or ARM architecture, this would have constituted a control dependency that would have created ordering between memory accesses. Compiler optimisations may remove these dependencies, so the language cannot provide the same guarantees. This optimisation has been observed when compiling with Clang and GCC.

A global optimisation The following is an example of value-range analysis: an optimisation that uses global information across threads. The original program on the left

only writes values 0 and 1 to `x`, so the compiler might reason that the write to `y` will always execute, transforming the code to that on the right, and again breaking a syntactic dependency:

```
      int x = 0;
      int y = 0;
x = 1; || if (x < 2);
       ||  y = 1;
```

```
   int x = 0;
   int y = 0;
x = 1; || y = 1;
```

A memory-model-aware optimisation In the relaxed setting, we can optimise in new ways because more executions are allowed. Recall the message-passing example from the previous section:

```
                int data = 0;
                int flag = 0;
data = 1; || while (flag <> 1);
flag = 1; || r = data;
```

The Power memory model allows this program to terminate with the final value 0 for `data`. This is because the processor might commit or propagate the writes on the left hand thread out of order. An optimising compiler targeting the Power memory model could optimise more aggressively than under SC: it could reorder the writes on the left hand thread while preserving the semantics of the program. A compiler for such a relaxed system need only check whether there is any thread-local reason that ordering should be preserved, and if not, it can reorder. This example demonstrates that under a relaxed memory model, more agressive optimisation is possible than under SC.

2.5 The C++11 memory model design

In this chapter we reviewed the memory models of the x86 and Power/ARM architectures. These, together with Itanium and MIPS, are the most common target architectures for the C/C++11 language, and the language is tuned to match x86, Power and Itanium. The processor architectures are relaxed: x86 allows store buffering and Power and ARM allow a range of behaviours. The C/C++11 memory model is intended to provide a low level interface to relaxed hardware, so for the highest performance, it must be at least as relaxed as the plain memory accesses of each supported architecture. The relaxed behaviours allowed by these systems include buffering of stores, speculation of reads and violations of multi-copy atomicity.

At the same time, there are properties that hold in each target architecture that the language can provide without a need for the compiler to insert explicit synchronisation;

they can be guaranteed with no degradation to performance. These properties include atomicity of accesses, coherence of the writes at a single location, atomicity of a compare-and-swap through either locked instructions (x86) or load-linked store-conditionals (Power/ARM), and ordering provided by dependencies in the program (note that dependencies are not always preserved in an optimising compiler). Chapter 5 describes outstanding problems with the C/C++11 treatment of dependencies.

When barriers are inserted, there are further properties that the language can take advantage of: in particular, on Power/ARM the barriers are cumulative. The design of the language features should map well to the sorts of barriers that the targets provide: if the specification of a language feature does not provide much ordering, but its underlying implementation is expensive and provides stronger guarantees, then the feature is poorly designed.

2.5.1 Compiler mappings

The concurrent C/C++11 features are syntactically provided as a library: the programmer is given atomic types and atomic accessor functions that read and write to memory. The previous chapter listed the load, store and atomic-exchange functions over atomic types, and enumerated the choices for the memory-order parameter of each. Each accessor function, when combined with a particular memory order, introduces a well-specified amount of ordering in the execution of the program, as described by the memory model. This ordering is ensured both by restricting optimisations on the compiler, and by inserting explicit synchronisation during code generation for a given architecture.

Throughout the design phase of the memory model, there were tables mapping the atomic accessor functions to their expected machine-instruction implementations on various target architectures: x86 [107], Power [78], ARM [106]. These tables relate to the relative cost of the primitives and help to understand the least-common ordering provided by each. The table below shows an implementation of the C++11 primitives with various choices of memory order over the x86, Power, ARM and Itanium architectures. The table below includes refinements and extensions to the early design-phase mappings.

C/C++11	X86	Power	ARM	Itanium
`load` RELAXED	`MOV` (from memory)	`ld`	`ldr`	`ld.acq`
`load` CONSUME	`MOV` (from memory)	`ld` + keep dependencies	`ldr` + keep dependencies	`ld.acq`
`load` ACQUIRE	`MOV` (from memory)	`ld; cmp; bc; isync`	`ldr; teq; beq; isb`	`ld.acq`
`load` SEQ_CST	`MOV` (from memory)	`hwsync; ld; cmp; bc; isync`	`ldr; dmb`	`ld.acq`
`store` RELAXED	`MOV` (into memory)	`st`	`str`	`st.rel`
`store` RELEASE	`MOV` (into memory)	`lwsync; st`	`dmb; str`	`st.rel`
`store` SEQ_CST	`MOV` (into memory), `MFENCE`	`hwsync; st`	`dmb; str; dmb`	`st.rel; mf`
`fence` ACQUIRE	⟨ignore⟩	`lwsync`	`dmb`	⟨ignore⟩
`fence` RELEASE	⟨ignore⟩	`lwsync`	`dmb`	⟨ignore⟩
`fence` ACQ_REL	⟨ignore⟩	`lwsync`	`dmb`	⟨ignore⟩
`fence` SEQ_CST	`MFENCE`	`hwsync`	`dmb`	`mf`

Note that the table provides the same implementation for atomic loads on x86 regardless of the memory order they are given. If C++11 targeted only x86, then it would be

poorly designed: programmers would be given access to many complicated features that provide few guarantees, while their programs would execute according to a much stronger implicit memory model. The Power and ARM architectures justify the more relaxed features of the model: the relaxed load maps to a plain load on each, for instance, while the others are more complex. Chapter 7 describes a proof that the x86 and Power mappings correctly implement C/C++11.

2.5.2 Top-level structure of the memory model

This section introduces the top-level structure of the C/C++11 memory model as defined by the standard. We presented the memory models of key processor architectures. These models are expressed as abstract machines: one should imagine them ticking through the execution of the program step-by-step, filling buffers, propagating writes, and so on. The C/C++11 memory model is rather different: it is an *axiomatic* memory model. Axiomatic models do not execute stepwise, instead they judge whether particular whole executions are allowed or not. The model forms its judgement over execution graphs, whose vertices are memory accesses, and whose labeled edges represent relationships such as program order. The most important sort of edge is the happens-before relation, but there are other relations that describe intuitive concepts: for example, the relation lo is a union of relations, where each totally orders all of the accesses to a particular mutex in memory.

At the top level, the memory model is a function from a program to either a set of executions or undefined behaviour. The behaviour of a program is defined by both a thread-local semantics and the memory model. Each execution is made up of two parts: there is a component that represents the syntactic structure of a particular path of control flow through the program, generated by the thread-local semantics, and another that represents the execution's dynamic interaction with memory, as allowed by the memory model. At the top level, the definition of the behaviour of a program according to the memory model is defined by the following steps:

1. The thread-local semantics generates the set of all executions whose memory accesses match those of a particular path of control flow through the program. The values read from memory are constrained only by the satisfaction of conditions in control-flow statements, so this set is large.

2. The set of executions is then pared down to the consistent executions: those whose read values correspond to memory behaviour allowed by the memory model.

3. Finally, if the set of filtered executions contains an execution with a cause of undefined behaviour (such as a data race), then the behaviour of the whole program is undefined. Otherwise, the filtered set of executions from 2 are the behaviours of the program.

The memory model's executions are an abstraction of the program's interaction with memory. Memory reads, writes, locks, unlocks and fences, are modeled by *actions* —

indivisible events that affect the memory. Executions impose no total order over memory actions. In particular there is no time ordering and no total sequentially-consistent ordering. Instead there are multiple partial relations that describe constraints present in the program, the sources of observed values, and the order of actions in memory. The behaviour of a program is largely decided by the partial *happens-before* relation, that can be thought of as a proxy for a total temporal order. Happens-before collects together syntactic constraints and dynamically induced inter-thread synchronisation. It is acyclic, but as we shall see, in its most general form it is partial and not transitive.

Reads and writes to memory can be either *atomic* or *non-atomic*. Non-atomic reads and writes are regular accesses of memory. They can give rise to *data races*, when happens-before does not sufficiently order actions on different threads. Data races are one of several causes of undefined behaviour: a program with even a single execution that contains a data race is allowed to do anything. Undefined behaviour is to be avoided, and in C/C++11 preventing the causes of undefined behaviour is left the responsibility of the programmer.

Races between atomic reads and writes do not produce undefined behaviour (races between non-atomics and atomics do), so atomics can be used to build racy data structures and algorithms that have well-defined behaviour. Together with locks and fences, atomic actions produce inter-thread synchronisation that contributes to happens-before. The programmer should use the atomics to create enough happens-before ordering in their program to avoid data races with non-atomic accesses.

In the most general case, happens-before is a complicated relation over actions, built from various constituent relations that are carefully combined to provide just the right amount of ordering through dependencies. Among the constituents are *sequenced-before* (sb) and *synchronises-with* (sw). Sequenced-before is a relation over actions, identified by the thread-local semantics, that corresponds to thread-local program order. The synchronises-with relation captures dynamically-created inter-thread synchronisation. Thread creation, locks and atomic accesses can create synchronises-with edges. For the purpose of explaining the following programming idioms, it is sufficient to think of happens-before as the transitive closure of the union of sequenced-before and synchronises-with.

2.5.3 Supported programming idioms

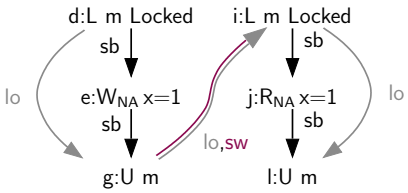
The various memory orders provided by C/C++11 are intended to support a variety of programming idioms. In this section we discuss the key ones, explain the intuition behind their treatment in the C/C++11 memory model, and the architectural justification for their correctness.

Concurrent programming with locks The language is intended to support concurrent programming with locks. In the following example, the program on the left is written

in C++11-like pseudocode. For clarity, layout and parallel composition are used instead of the more verbose thread-creation syntax of the language. Otherwise, the keywords are faithful to C++11.

In our first example, a parent thread creates an integer `x` and a mutex `m` and then spawns two threads, each of which lock the mutex, access `x` and then unlock the mutex. On the right, there is an execution of this program. The execution is a graph over memory accesses (we elide the accesses from the parent thread, and accesses to thread-local `r1`). Each access is written in a concise form: first there is a unique action identifier, then a colon, a letter indicating whether the access is a read (R), a write (W), an unlock (U) or a lock (L), then, reads and writes are followed by a memory order (NA indicating non-atomic below), a location and a value, unlocks are followed by a location and locks are followed by a location and an outcome (locked or blocked). Accesses from the same thread are printed vertically, and program order is captured by the *sequenced-before* relation, labeled sb below, between them. The lo and sw relations will be explained below.

```
          int x = 0;
          mutex m;
m.lock();     ‖ m.lock();
x = 1;        ‖ r1 = x;
m.unlock();   ‖ m.unlock();
```



This program uses mutexes to protect the non-atomic accesses and avoid data races. The C/C++11 memory model places all of the accesses to a particular mutex in a total order called *lock order*, labeled lo above. The lock and unlock actions that bound locked regions of code must agree with this order, and there is a locking discipline required of programmers: they must not lock a mutex twice without unlocking it, or unlock twice without locking it. (see Chapter 3 for details). Unlocks and locks to the same location create synchronisation, in particular lock-order-earlier unlocks *synchronise with* lock-order later locks.

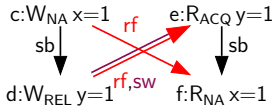
The execution shows a particular lock order, marked lo, and the synchronises-with edge, marked sw, that it generates. Together with sequenced-before, this edge means that the store and load to `x` are ordered by happens-before, and this execution does not have a data race.

Mutexes can be used to write concurrent programs without data races. The implementation of locks and unlocks on the target architectures are expensive, so programs that require higher performance should use the atomic accesses.

Message passing C/C++11 efficiently supports the message-passing programming idiom on all target architectures. The programmer can use atomic release writes and atomic acquire loads to perform racy accesses of the flag variable (`y` below) without leading to

undefined behaviour. In the program on the left below, the parent thread initialises some non-atomic data `x` and an atomic `y`. The child threads then attempt to use `y` as a flag in the message-passing idiom. The execution on the right represents the outcome where the read of `y` read from the write in the left-hand thread on the first iteration of the while loop. The edge labeled `rf` is the *reads-from* relation, that relates writes to the reads that take their values.

```
                    int x = 0;
                atomic_int y = 0;
x = 1;          ||  while (y.load(acquire) <> 1);
y.store(1,release); r1 = x;
```



This program involves non-atomic accesses to `x` from both threads, but the release and acquire atomics create synchronisation where the acquire reads from the release, so the while loop ensures that there will always be a happens-before edge between the two non-atomic memory accesses, and the program is not racy. Moreover, the memory model states that non-atomic loads from memory must read from the most recent write in happens-before, so we cannot read the initialisation of `x` on the right-hand thread.

Note that the compilation mappings above preserve this behaviour on the hardware when the program is translated. On x86, the stores and loads in the program are translated to `MOV` instructions. The first-in-first-out nature of the architecture's write buffers ensures that if we see the write of `y` on the left hand thread, then we will see the write of `x`. In general, on Power, this is not the case, but the mapping inserts barrier instructions that ensure the correct behaviour. The following pseudocode represents the Power translation with inserted barriers, removing the while loop for clarity:

```
          int x = 0;
          int y = 0;
x = 1;  || r = y;
lwsync; || cmp;bc;isync;
y = 1;  || r1 = 1;
```

The barriers on the right-hand thread act as a control dependency followed by an `isync` barrier. The previous section explained that this is sufficient to disable the Power speculation mechanism, and combined with the `lwsync`, that prevents out-of-order commitment or propagation of the writes, these barriers ensure correct behaviour.

The language supports several variants of the message passing programming idiom. One can use separate fences with relaxed atomics instead of the release and acquire annotated atomics — using fences, the cost of synchronisation need not be incurred on every iteration of the loop in the example above, but instead only once, after the loop.

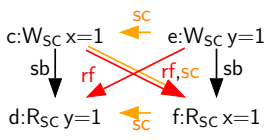
Variants of the message passing test above with an address dependency between the loads on the right hand thread can use the consume memory order rather than acquire, resulting in a data-race-free program with no costly barrier insertion on the read thread on Power and ARM hardware.

Racy sequential consistency The third programming idiom that C/C++11 is intended to support is sequential consistency. Programmers who require SC can use atomic accesses with the SEQ_CST memory order. The memory model places all SEQ_CST atomics in a total order called *SC order*. Reads in SC order must read the immediately preceding write at the same location. Race-free programs that use only the SEQ_CST memory order for shared accesses forbid all relaxed behaviour and have SC semantics, and we can use this fragment of the language to write code that relies on SC behaviour.

In Section 2.2, we showed that it is store-buffering relaxed behaviour that causes a naive fence-free implementation of Dekker's to violate mutual exclusion on x86 hardware. We can use the SEQ_CST atomics to write programs like Dekker's algorithm.

To understand the memory model's treatment of SEQ_CST atomics, we return to the store-buffering example, with all accesses made SEQ_CST atomic.

```
                atomic_int x = 0;
                atomic_int y = 0;
x.store(1,seq_cst);   || y.store(1,seq_cst);
r1 = y.load(seq_cst); || r2 = x.load(seq_cst);
```



If the outcome 0/0 can be observed, then this test admits the store-buffering relaxed behaviour, and the behaviour of the program is not SC. The execution on the right shows the SC order for the two child threads. Because of the thread-local order of the accesses, the final action in SC order must be one of the loads on the child thread, and the write on the other thread must precede it in SC order. According to the model, the load may not read the initialisation write that happens before this SC write, so it is not possible for this program to exhibit the 0/0 outcome.

The compilation mappings place an `MFENCE` after the SEQ_CST stores on x86, an `hwsync` before loads on Power, and a `dmb` after stores on ARM. In each case there is a strong barrier between the accesses on each thread in the store-buffering test:

```
                int x = 0;
                int y = 0;
x = 1;                || y = 1;
MFENCE/hwsync/dmb;    || MFENCE/hwsync/dmb;
r1 = y;               || r2 = x;
```

On each architecture, this is sufficient to disallow the relaxed behaviour. Section 2.2 discusses the architectural intuition behind this for x86. On Power and ARM, the relaxed outcome is allowed without the barriers because the stores and loads can be committed out of order, or the loads can be committed before the stores are propagated. The barriers disable these optimisations on each target.

2.5.4 Standard development process

The standards of the C and C++ languages are developed by ISO SC22 working groups 21 and 14 respectively. The membership of each workgroup comprises a mix of industrial members from various companies and countries.

The language is mutated over time through an evolutionary process that involves collecting proposed amendments to a working draft of the standard, and periodically voting to apply them to the draft on a case-by-case basis. The drafts are prepared with a deadline for ratification in mind, although in the case of C++11, called C++0x in development, this turned out to be flexible. Prior to ratification, the committee stabilises the working draft and declares it to be a "Final Committee Draft", which can then be ratified by the working group, subject to the outcome of a vote. After ratification the working group can publish corrigenda that become part of the specification. These corrigenda are used to make clarifications and sometimes to fix errors.

The working groups record reports containing comments, potential amendments, hypothetical ideas and working drafts in a stream of enumerated documents, called colloquially *N-papers*. The process is admirably open: there is a public archive of N-papers that one can browse by date [55]. The ability to present a publicly visible document that is registered as officially submitted to the language designers is invaluable for academic collaboration.

In my case, interaction with the standardisation committee took place in two modes. The first was informal email exchange. The committee are open to contact from external experts, and have mailing lists on which language issues are discussed. This informal contact was invaluable for the early development of the formal model. The other mode of contact was through a process of official report and face-to-face advocacy at WG21 meetings in the lead up to the release of the standard. In earlier stages, issues can be raised for discussion by submitting the issue by email to one of the subgroup chairs, or by submitting an N-paper. In the later stages of the standardisation process, a member of a national delegation must submit a formal comment to allow discussion of an issue. Such comments often refer to an N-paper that describes the issue in detail.

The main working group is large, and the meetings I went to had a high enough attendance that it would have been difficult to discuss contentious technical issues among all attendees. Usefully, WG21 had a concurrency subgroup, attended by a small number of well-informed experts.

C++11 was to be the first major revision of the language since its initial standardisation in 1998 [4] (C++03 [5] was a minor update, and the 2007 revision was merely a technical report [6], rather than a full international standard). Development of the atomics library seems to have started in 2007 [34], and my involvement began in 2009 [20]. I attended meetings of working group 21 in Rapperswil, Switzerland in August 2010, Batavia, Illinois (USA) in November 2010, and Madrid, Spain in March 2011. My comments and suggestions were incorporated into the August 2010 national body comments [51] on the final draft of C++11. The finalised standard was ratified in September 2011 [30].

Chapter 5 describes concrete issues with the language specification, some of which were fixed in amendments to drafts before ratification, some of which became part of a corrigendum, and some of which have not been addressed. Each issue references the N-papers that described the issue to the standardisation committee and those that suggested amendments to drafts.

Chapter 3

The formal C/C++ memory model

This chapter describes a mechanised formal definition of the C/C++11 memory model, following the C++11 standard (Appendix A establishes a close link to the text). The formal model was developed in discussion with the C++11 standardisation committee during the drafting phases of the C++11 standard. This process brought to light several major errors in drafts of the standard and led to solutions that were incorporated in the ratified standards (Chapter 5 discusses these changes, together with remaining issues). The close contact with the standardisation committee, the link to the standard text and the fact that this model fed changes to the standard establish it as an authoritative representation of the C++11 memory model. C11 adopts the same memory model as C++11 for compatibility, so the model presented here applies to C as well.

The formal model relies on several simplifications. Details like alignment, bit representation, and trap values are ignored, we assume variables are aligned and disjoint, signal handlers are not modeled and neither is undefined behaviour introduced thread-locally: e.g. division by zero, out-of-bound array accesses. We do not consider mixed size accesses or allocation and deallocation of memory: both would require a memory-layout model that is omitted for simplicity.

The memory model is introduced in stages, as a sequence of derivative models that apply to successively more complete sublanguages of C/C++11. The mathematics that describe the models is automatically typeset from the source, written in the Lem specification language [90] (the full set of definitions are reproduced in Appendix C). The first section introduces a cut-down version of the C/C++11 memory model that describes the behaviour of straightforward single-threaded programs, and in doing so, introduces the underlying types and top-level structure of the memory model that will apply to the rest of the models in the chapter. This introductory section is followed by a series of formal memory models that incrementally introduce concurrency features, together with the mathematics that describe how they behave, and the underlying architectural intuitions related to them. The chapter culminates in the presentation of the full formal model of C/C++11 concurrency as defined by the ISO standard. The following table displays the

sublanguages and the related parts of the model that each section introduces:

Section	sublanguage	elements of model introduced
§3.1	single threaded (no locks, no atomics, no fences)	top-level model structure, sequenced-before
§3.2	concurrent with locks	lock-order, synchronises-with, data races
§3.3	concurrent with relaxed atomics and locks	atomics, modification order, coherence, CAS
§3.4	concurrent with release and acquire atomics and locks (no relaxed atomics)	release-acquire synchronisation
§3.5	release, acquire and relaxed atomics and locks	release sequences
§3.6	all of the above plus release and acquire fences	hypothetical release sequences
§3.7	all of the above plus SC atomics	SC-order
§3.8	all of the above plus SC fences	SC fence restrictions
§3.9	all of the above plus consume atomics (locks, all atomics, all fences)	data-dependence, carries-a-dependency-to, dependency-ordered-before
§3.10	locks, all atomics, all fences	visible-sequences-of-side-effects

3.1 Top-level structure by example: single-threaded programs

Before introducing the memory model that governs simple single-threaded programs, it is necessary to formally define the top-level structure and underlying types of the memory model. We start with the type of actions.

Memory actions Reads and writes to memory, locks, unlocks and fences are modelled by memory events called actions. The action type is given below. Each action has a unique action-identifier, of type AID, and a thread identifier, of type TID, identifying its host thread.

```
type ACTION =
  | LOCK of AID * TID * LOCATION * LOCK_OUTCOME
  | UNLOCK of AID * TID * LOCATION
  | LOAD of AID * TID * MEMORY_ORDER * LOCATION * CVALUE
  | STORE of AID * TID * MEMORY_ORDER * LOCATION * CVALUE
  | RMW of AID * TID * MEMORY_ORDER * LOCATION * CVALUE * CVALUE
  | FENCE of AID * TID * MEMORY_ORDER
  | BLOCKED_RMW of AID * TID * LOCATION
```

Locks have an outcome and can either leave the mutex LOCKED or BLOCKED:

```
type LOCK_OUTCOME =
    LOCKED
  | BLOCKED
```

Loads and stores both have an associated VALUE, that is read or written respectively. A *read-modify-write* action is the result of a successful compare-and-swap, atomic increment, atomic add, or similar call. It has two VALUES: first is the value read, and the second the value written. A BLOCKED_RMW action is generated when one of these calls permanently blocks (e.g. from a non-terminating load-linked/store-conditional loop).

The *memory order* of an action, defined below, specifies the strength of ordering that an action generates in an execution. Actions annotated with NA are regular non-atomic C/C++ memory accesses, whereas the other memory orders are part of the new low-level atomic library. The precise details of the interaction of memory orders are discussed in the coming sections.

```
type MEMORY_ORDER =
  | NA
  | SEQ_CST
  | RELAXED
  | RELEASE
  | ACQUIRE
  | CONSUME
  | ACQ_REL
```

Actions that read from or write to memory specify a LOCATION: the abstract location in memory that the action accesses. Locations can be one of three kinds: non-atomic, atomic and mutex. Only non-atomic actions can act at non-atomic locations. Similarly, only locks and unlocks can act at mutex locations. Atomic locations are accessed by atomic reads and writes, but their initialisation is non-atomic. C/C++11 requires the programmer to use synchronisation to avoid data races on the initialisation of atomic locations. This design decision allows compilers to implement atomic initialisations without emitting additional synchronisation on the target processor. *Location-kind* is defined below.

```
type LOCATION_KIND =
    MUTEX
  | NON_ATOMIC
  | ATOMIC
```

The memory model described in the rest of this section applies to programs that have only a single thread and do not use any of the concurrency features of the language. The only memory accesses that such programs can produce are non-atomic loads and stores of memory. Take for example, the following program:

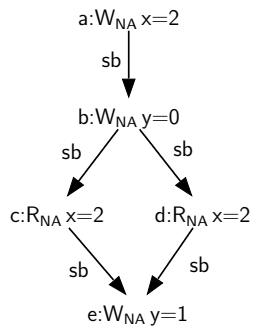
```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0; }
```

Executions of this program have two non-atomic store actions corresponding to the initialisation of `x` and `y`, two non-atomic load actions corresponding to the two reads of `x` in the comparison, and a write of the result to `y` in a final non-atomic store action.

The behaviour of the program is represented by a set of *executions*. Each execution is a graph whose vertices are memory actions, and whose edges represent syntactically imposed constraints or dynamic memory behaviour. For single-threaded programs, the only important syntactically imposed constraint is sequenced-before (sb). Sequenced-before captures the program order of the source program. In C and C++ this is a partial relation over the actions on a particular thread — the ordering is partial to provide compilers flexibility in their order of evaluation.

Returning to the previous example, we present its execution (this example has only one that satisfies the model) on the right below. The vertices of the graph are the memory actions; each has a label, an R or W, representing a read or write respectively, a subscript that identifies the memory order of the access, then a location, an equals sign and a value. The labeled directed edges correspond to the sequenced-before relation. Observe that it is not total: the two loads in the operands of the == operator are unordered. Sequenced before is transitive, but for clarity we draw only its transitive kernel.

```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0; }
```



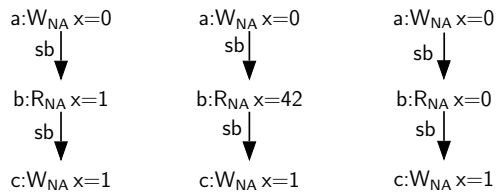
Pre-execution For a given path of control flow through a program, we define a *pre-execution* as a record containing a set of uniquely identified actions that represent the program's accesses of memory, a set of thread identifiers, a map from locations to location kinds, and relations representing the constraints imposed by the syntactic structure of the program. Three relations track syntactic constraints: sequenced-before (sb) is the thread-local program order, data-dependence (dd) represents data and address dependencies, and

additional-synchronises-with (asw) captures parent to child ordering on thread creation. The pre-execution type is then:

```
type PRE_EXECUTION =
  ⟪ actions : SET (ACTION);
    threads : SET (TID);
    lk : LOCATION → LOCATION_KIND;
    sb : SET (ACTION * ACTION) ;
    asw : SET (ACTION * ACTION) ;
    dd : SET (ACTION * ACTION) ;
  ⟫
```

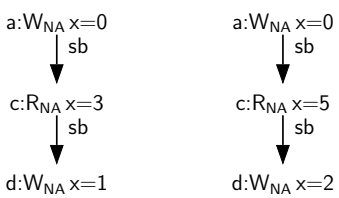
For each program, a set of pre-executions represents each path of control flow, with the values of accesses that read from memory constrained only by the values that are required to satisfy conditionals in the control-flow path of the pre-execution. This set is calculated from the program source code. For example, the following program gives rise to pre-executions including the three on the right below. Note the unexpected read values; they are unconstrained in a pre-execution.

```
int main() {
  int x = 0;
  r1 = x;
  x = 1;
  return 0; }
```

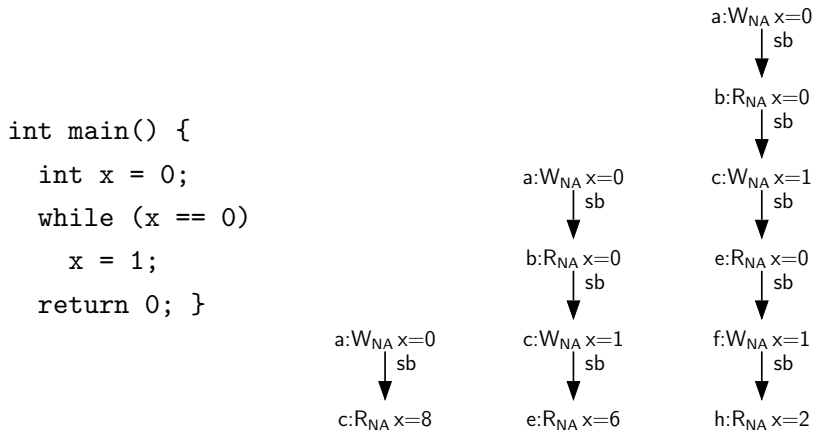


If-statements produce a set of pre-executions for each possible branch, so the set of pre-executions of the following program contains both of the pre-executions below.

```
int main() {
  int x = 0;
  if (x == 3)
    x = 1;
  else
    x = 2;
  return 0; }
```



While loops produce a set of pre-executions for each possible unrolling of the loop. Three pre-executions (printed side-by-side) of the program are shown below:



Thread-local semantics A *thread-local semantics* is a function from the source code of a program to a set of pre-executions that represent possible executions of each path of control flow. We leave the thread-local semantics as a parameter to the concurrency model in order to escape modelling the whole of the language. Chapter 4 introduces a tool that calculates the executions of small C/C++11 programs. This tool contains a thread-local semantics, expressed in OCaml, for a small subset of the C++11 language. Ideally one would have a formal interpretation of the thread-local semantics defined by the C and C++ standards. The type of the thread-local semantics is:

```
type OPSEM_T = PROGRAM → PRE_EXECUTION → BOOL
```

3.1.1 The execution witness, calculated relations and candidate executions

The memory model represents the behaviour of the program as a set of complete executions. The type of each of these is made up of three components: a pre-execution as described above, an execution witness, made up of more relations, that describes the dynamic behaviour of memory, and a set of calculated relations.

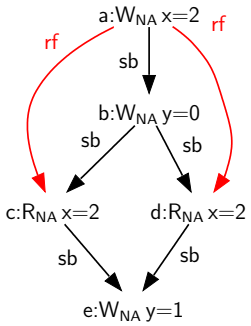
Execution witness The *execution witness* is defined as record containing the following relations over memory actions: reads-from (rf) relates writes to each read that reads from them, modification order (mo) is a per-location total order over all the writes at each atomic location, SC-order (sc) is a total order over all actions with the SEQ_CST memory order, lock order (lo) is a per-location total order over all the mutex actions, and total memory order (tot) is a total order over all memory actions that is used in Chapter 6. The memory models in the following sections will each use a selection of these relations. In the single-threaded model, we use only the reads-from relation. The type of the execution witness is:

```
type EXECUTION_WITNESS =
  ⦃ rf : SET (ACTION * ACTION);
```

mo : SET (ACTION $*$ ACTION);
sc : SET (ACTION $*$ ACTION);
lo : SET (ACTION $*$ ACTION);
tot : SET (ACTION $*$ ACTION);
⦈

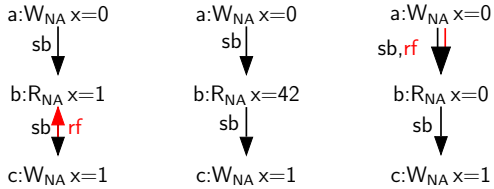
The reads-from (rf) relation relates stores to the load actions that take their values. Returning to the example from the previous section, we show the pre-execution together with a reads-from relation, with directed edges labeled rf:

```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0; }
```



The memory model will decide which execution witnesses represent behaviour that is allowed for a given pre-execution. The model enumerates all possible execution witnesses and then filters them. Some of the enumerated witnesses will not correspond to allowable behaviour. For instance, the following three candidate executions (printed side-by-side) are some of those enumerated for the program on the right. Not all candidate executions will be allowed by the memory model. We adopt the convention of eliding accesses to variables named "r" followed by a number, and use them only to capture values read from memory and to construct dependencies.

```
int main() {
  int x = 0;
  r1 = x;
  x = 1;
  return 0; }
```



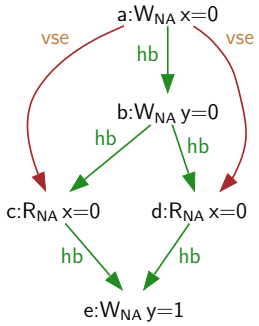
Calculated relations Given a pre-execution and an execution witness, the memory model defines a set of derived relations that collect together multiple sources of ordering, and represent higher-level concepts. The single-threaded model only has two calculated relations: happens-before (hb) and visible side effect (vse). As discussed in Chapter 2, happens-before is intended to serve as an intuitive ordering over the actions of the execution. In later models happens-before will include inter-thread synchronisation, but in the single-threaded model, happens-before is equal to sequenced-before. The *visible side*

effects of a particular read are all of the writes at the same location that happen before it, such that there is no happens-before-intervening write to the same location; these are the writes that the read may legitimately read from. In simple cases this set will be a singleton, but because sequenced-before is partial, it may not be. The model represents visible side effects as a relation from each visible side effect to its corresponding read. The relation is expressed formally as a set comprehension:

$$
\begin{aligned}
&\mathsf{let}\ \mathit{visible\_side\_effect\_set}\ \mathit{actions}\ \mathit{hb}\ = \\
&\quad \{\ (a,\ b)\ |\ \forall\ (a,\ b) \in \mathit{hb}\ | \\
&\qquad \text{is\_write}\ a \wedge \text{is\_read}\ b \wedge (\text{loc\_of}\ a = \text{loc\_of}\ b) \wedge \\
&\qquad \neg\ (\ \exists\ c \in \mathit{actions}.\ \neg\ (c \in \{a,\ b\}) \wedge \\
&\qquad\quad \text{is\_write}\ c \wedge (\text{loc\_of}\ c = \text{loc\_of}\ b) \wedge \\
&\qquad\quad (a,\ c) \in \mathit{hb} \wedge (c,\ b) \in \mathit{hb})\ \}
\end{aligned}
$$

Returning to the example program, the following shows the happens-before and visible-side-effect relations of a candidate execution, but elides all others.

```
int main() {
  int x = 0;
  int y = 0;
  y = (x==x);
  return 0; }
```



Each memory model collects these relations in a list of named calculated relations:

$$\mathsf{type}\ \text{RELATION\_LIST}\ =\ \text{LIST}\ (\text{STRING}\ *\ \text{SET}\ (\text{ACTION}\ *\ \text{ACTION}))$$

Candidate executions Together, a pre-execution, an execution witness and a list of calculated relations form a *candidate execution* of the program:

$$\mathsf{type}\ \text{CANDIDATE\_EXECUTION}\ =\ (\text{PRE\_EXECUTION}\ *\ \text{EXECUTION\_WITNESS}\ *\ \text{RELATION\_LIST})$$

In summary, the first step in working out the behaviour of a program is to find all pre-executions that agree with the source program, according to the thread-local semantics. For each pre-execution, we then find all possible execution witnesses, and for each pair of pre-execution and associated execution witness, we link this with a set of calculated relations. This gives us a large set of candidate executions that the rest of the memory model will constrain.

3.1.2 The consistency predicate

The behaviour of a program, whether defined or undefined, is decided by the set of all candidate executions that satisfy the *consistency predicate* of the memory model: a conjunction of predicates that restrict what sort of dynamic behaviour of memory is allowed in an execution. This section walks through the conjuncts of the consistency predicate that applies to single-threaded programs.

Well-formed threads One conjunct of the consistency predicate is present for every memory model: the *well_formed_threads* predicate encodes properties of the set of pre-executions that the thread-local semantics must satisfy. The predicate is itself a conjunction of predicates, each of which is presented here.

All load, store, read-modify-write and fence actions in the pre-execution must be annotated with an appropriate memory order: it would be a programmer error to annotate an atomic load with the release memory order, for instance:

```
let well_formed_action a =
  match a with
    | Load _ _ mo _ _ → mo ∈ {NA, Relaxed, Acquire, Seq_cst, Consume}
    | Store _ _ mo _ _ → mo ∈ {NA, Relaxed, Release, Seq_cst}
    | RMW _ _ mo _ _ _ → mo ∈ {Relaxed, Release, Acquire, Acq_rel, Seq_cst}
    | Fence _ _ mo → mo ∈ {Relaxed, Release, Acquire, Acq_rel, Consume, Seq_cst}
    | _ → true
    end
```

Each action must act on an appropriate location type: locks and unlocks act on mutex locations, non-atomic actions act on non-atomic locations or atomic locations, and atomic actions act on atomic locations. Note that in C++11, non-atomic reads cannot act on atomic locations, but in C11 they can. The formal model applies to C as well, but we follow the C++ restriction here.

```
let actions_respect_location_kinds actions lk =
 ∀ a ∈ actions. match a with
    | Lock _ _ l _ → lk l = Mutex
    | Unlock _ _ l → lk l = Mutex
    | Load _ _ mo l _ →
        (mo = NA ∧ lk l = Non_Atomic) ∨ (mo ≠ NA ∧ lk l = Atomic)
    | Store _ _ mo l _ →
        (mo = NA ∧ lk l = Non_Atomic) ∨ lk l = Atomic
    | RMW _ _ _ l _ _ → lk l = Atomic
    | Fence _ _ _ → true
```

```
  | Blocked_rmw _ _ l → lk l = Atomic
  end
```

Some actions block forever, and the thread-local semantics must generate pre-executions in which permanently blocked actions have no successors in sequenced-before on the same thread:

```
let blocking_observed actions sb =
  (∀ a ∈ actions.
    (is_blocked_rmw a ∨ is_blocked_lock a)
    ⟶
    ¬ (∃ b ∈ actions. (a, b) ∈ sb))
```

The standard says that functions are ordered with respect to all other actions on the same thread by sequenced-before, whether or not an order is imposed by the syntax of the program; their sequencing is described as indeterminate (see Appendix A for the rationale). All memory accesses on atomic and mutex locations are the result of calls to library functions, and as a consequence, they are ordered with respect to all other actions on the same thread by sequenced before. This property is captured by the following conjunct of the well-formed-threads predicate:

```
let indeterminate_sequencing Xo =
  ∀ a ∈ Xo.actions b ∈ Xo.actions.
    (tid_of a = tid_of b) ∧ (a ≠ b) ∧
    ¬ (is_at_non_atomic_location Xo.lk a ∧ is_at_non_atomic_location Xo.lk b) ⟶
    (a, b) ∈ Xo.sb ∨ (b, a) ∈ Xo.sb
```

We do not model the creation and joining of threads explicitly. Instead, the thread-local semantics provides us with the additional synchronises-with relation, that captures parent-to-child thread creation ordering. The calculated relation *sbasw* is the transitive closure of the union of sb and asw. This relation captures program order and the inter-thread ordering induced by creating and joining threads. We require this relation to be acyclic.

```
let sbasw Xo = transitiveClosure (Xo.sb ∪ Xo.asw)
```

In addition to the requirements above, *well_formed_threads* predicate provides the following guarantees: all actions have a unique identifier (we require the projection of the action identifier to be injective), the sb and asw relations only relate actions of the execution, the sb relation must only relate actions on the same thread, the asw relation must only relate actions on different threads, the sb and dd relations are both strict partial orders, and the dd relation is a subset of the sb relation. The *well_formed_threads* predicate:

```
let well_formed_threads ((Xo, _, _) : (PRE_EXECUTION * EXECUTION_WITNESS *
RELATION_LIST)) =
  (∀ a ∈ Xo.actions. well_formed_action a) ∧
  actions_respect_location_kinds Xo.actions Xo.lk ∧
  blocking_observed Xo.actions Xo.sb ∧
  inj_on aid_of Xo.actions ∧
  relation_over Xo.actions Xo.sb ∧
  relation_over Xo.actions Xo.asw ∧
  threadwise Xo.actions Xo.sb ∧
  interthread Xo.actions Xo.asw ∧
  isStrictPartialOrder Xo.sb ∧
  isStrictPartialOrder Xo.dd ∧
  Xo.dd ⊆ Xo.sb ∧
  indeterminate_sequencing Xo ∧
  isIrreflexive (sbasw Xo) ∧
  finite_prefixes (sbasw Xo) Xo.actions
```

Consistency of the execution witness The rest of the consistency predicate judges whether the relations of the execution witness are allowed by the memory model. These remaining conjuncts, presented below, are straightforward for the model that covers sequential programs.

The sc, mo and lo relations are required to be empty; they are not used in this model:

```
let sc_mo_lo_empty (_, Xw, _) = null Xw.sc ∧ null Xw.mo ∧ null Xw.lo
```

The reads-from relation must only relate actions of the execution at the same location, it must relate writes of some value to reads that get the same value, and a read can only read from a single write:

```
let well_formed_rf (Xo, Xw, _) =
  ∀ (a, b) ∈ Xw.rf.
    a ∈ Xo.actions ∧ b ∈ Xo.actions ∧
    loc_of a = loc_of b ∧
    is_write a ∧ is_read b ∧
    value_read_by b = value_written_by a ∧
    ∀ a′ ∈ Xo.actions. (a′, b) ∈ Xw.rf ⟶ a = a′
```

There is a reads-from edge to a read if and only if there is a write to the same location that happens before it. This leaves open the possibility that consistent executions might have reads with no rf edge, modelling a read from uninitialised memory that might take any value:

```
let det_read (Xo, Xw, _ :: ("vse", vse) :: _) =
  ∀ r ∈ Xo.actions.
    is_load r ⟶
    (∃ w ∈ Xo.actions. (w, r) ∈ vse) =
    (∃ w' ∈ Xo.actions. (w', r) ∈ Xw.rf)
```

Finally, all reads that do have a corresponding reads-from edge, must read from one of the read's visible side effects. In this model only non-atomic locations are allowed, so this applies to all rf edges:

```
let consistent_non_atomic_rf (Xo, Xw, _ :: ("vse", vse) :: _) =
  ∀ (w, r) ∈ Xw.rf. is_at_non_atomic_location Xo.lk r ⟶
    (w, r) ∈ vse
```

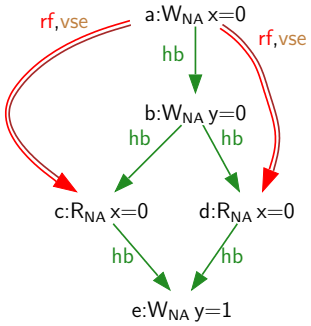
The consistency predicate, *single_thread_consistent_execution*, collects these conjuncts together. It is represented as a tree, whose branches and leaves are named in order to provide names and useful structure to tools that use the model.

```
let single_thread_consistent_execution =
  Node [ ("assumptions", Leaf assumptions);
         ("sc_mo_lo_empty", Leaf sc_mo_lo_empty);
         ("tot_empty", Leaf tot_empty);
         ("well_formed_threads", Leaf well_formed_threads);
         ("well_formed_rf", Leaf well_formed_rf);
         ("consistent_rf",
           Node [ ("det_read", Leaf det_read);
                  ("consistent_non_atomic_rf", Leaf consistent_non_atomic_rf) ]) ]
```

Consistency of an example execution The behaviour of C/C++ programs is largely decided by the happens-before relation. Note that the value of a read is only constrained by the memory model when there is a reads-from edge, and can take any value if there is none.

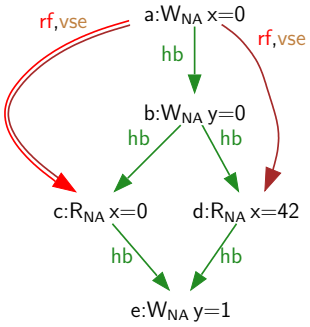
Consider the consistency predicate applied to the running example. The predicate is phrased in terms of the reads-from, happens-before and visible side effect relations, so in the candidate execution below, only those relations are shown:

```
int main() {
  int x = 0;
  int y = 0;
  y = (x==x);
  return 0; }
```



Recalling that sequenced-before is transitive, and hence so is happens-before, it is clear that all of the conjuncts of the predicate hold in this small example, so this candidate execution is indeed consistent.

Contrast this with the following inconsistent execution of the program:



Here, despite the fact that there is a visible side effect of the read labeled d, there is no reads-from edge to it, and the execution violates the *det_read* conjunct of the consistency predicate.

3.1.3 Undefined behaviour

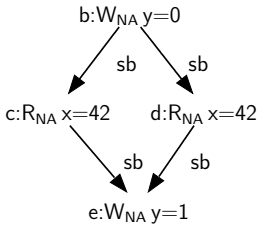
In C/C++11 there are several behaviours that programmers have a responsibility to avoid. The model provides a 'catch-fire' semantics: if even one consistent execution of a program exhibits one of these faulty behaviours, then the semantics of the entire program is undefined. In single-threaded programs, there are two possible sources of undefined behaviour: *indeterminate reads* and *unsequenced races*.

Indeterminate reads If a read is not related to any write by a reads-from edge, then the memory model does not restrict the value it reads. This can only be consistent if there are no writes of the same location that happen before the read, or, intuitively, if the program has read from uninitialised memory. This behaviour, called an *indeterminate read*, is considered a fault, and results in undefined behaviour:

let *indeterminate_reads* $(Xo,\ Xw,\ \_)\ =$
$\{b \mid \forall\ b \in Xo.actions \mid \text{is\_read}\ b \wedge \neg\ (\exists\ a \in Xo.actions.\ (a,\ b) \in Xw.rf)\}$

To illustrate a consistent execution with an indeterminate read, consider a small adjustment of the running example: removing the initialisation write of x leaves the reads of x in the consistent execution below without a preceding write in happens-before, and therefore indeterminate.

```
int main() {
  int x;
  int y = 0;
  y = (x==x);
  return 0; }
```



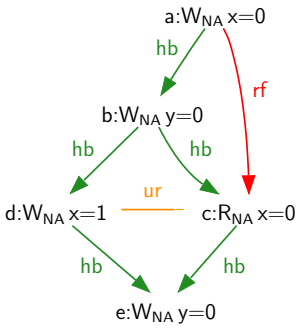
Unsequenced races As we have seen, sequenced-before is not total, and therefore it is possible to access a location twice from the same thread without ordering the accesses. The C/C++11 languages allow programmers to write code that leaves reads to the same location unordered, but a write that is unordered with respect to another read or write on the same location is considered a fault, called an *unsequenced race*, and this fault results in undefined behaviour for the whole program:

let *unsequenced_races* $(Xo,\ \_,\ \_)\ =$
$\{\ (a,\ b) \mid \forall\ a \in Xo.actions\ b \in Xo.actions \mid$
$\text{is\_at\_non\_atomic\_location}\ Xo.lk\ a \wedge$
$\neg\ (a = b) \wedge (\text{loc\_of}\ a = \text{loc\_of}\ b) \wedge (\text{is\_write}\ a \vee \text{is\_write}\ b) \wedge$
$(\text{tid\_of}\ a = \text{tid\_of}\ b) \wedge$
$\neg\ ((a,\ b) \in Xo.sb \vee (b,\ a) \in Xo.sb)\ \}$

A program with an unsequenced race has an execution that does not order a write and another access with respect to one another, so, intuitively, they might happen at the same time, and may interfere with one another.

A small modification of the running example introduces an unsequenced-race: change a read to a write in one operand of the equality operator. The altered program and a consistent execution are given below. Note the unsequenced race marked in orange.

```
int main() {
  int x = 0;
  int y = 0;
  y = (x==(x = 1));
  return 0; }
```



The list of undefined behaviours We collect the two sorts of undefined behaviour together, attaching names to them to differentiate faults made up of one action, and faults between two actions:

```
let single_thread_undefined_behaviour =
  [ Two ("unsequenced_races", unsequenced_races);
    One ("indeterminate_reads", indeterminate_reads) ]
```

3.1.4 Model condition

The models throughout this chapter range from simple ones for restricted subsets of the concurrency features to more complex ones for a more complete fragment of the language, culminating in the full C/C++11 memory model. Each model is associated with a restricted subset of the language to which the model applies — the *model condition* precisely identifies these programs. If the model is applied to a program that violates the model condition, then the model gives the program undefined behaviour, even though the program may have defined behaviour in a more complete model.

The model presented in this section applies to programs with a single thread that use only non-atomic memory locations.

```
let single_thread_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
  ∃ b ∈ Xo.actions. ∀ a ∈ Xo.actions.
    (tid_of a = tid_of b) ∧
    match (loc_of a) with
    | Nothing → false
    | Just l → (Xo.lk l = Non_Atomic)
    end
```

3.1.5 Memory model and top-level judgement

The constituent parts of the single-threaded memory model have all been introduced, so we can now present the *single_thread_memory_model* record:

```
let single_thread_memory_model =
  ⦃ consistent = single_thread_consistent_execution;
    relation_calculation = single_thread_relations;
    undefined = single_thread_undefined_behaviour;
    relation_flags =
      ⦃ rf_flag = true;
        mo_flag = false;
        sc_flag = false;
        lo_flag = false;
        tot_flag = false ⦄
  ⦄
```

The semantics of a program are decided by the combination of the thread-local semantics and the memory model. Programs can have undefined behaviour, or allow a set of candidate executions:

```
type PROGRAM_BEHAVIOURS =
    DEFINED of SET (OBSERVABLE_EXECUTION)
  | UNDEFINED
```

To generate the behaviours of a program, the semantics first generates a set of *consistent executions*. Each execution in this set must satisfy the thread-local semantics, satisfy the consistency predicate and have the correct set of calculated relations. If the program obeys the model condition, and there are no sources of undefined behaviour in any consistent execution, then the behaviour of the program is defined and is the set of consistent executions, otherwise the behaviour is undefined. The *behaviour* function performs this calculation, taking a thread-local semantics and a program, and returning the behaviours of that program.

```
let behaviour M condition opsem (p : PROGRAM) =
  let consistent_executions =
    { (Xo, Xw, rl) |
        opsem p Xo ∧
        apply_tree M.consistent (Xo, Xw, rl) ∧
        rl = M.relation_calculation Xo Xw } in
  if condition consistent_executions ∧
    ∀ X ∈ consistent_executions.
      each_empty M.undefined X
  then Defined (observable_filter consistent_executions)
  else Undefined
```

The behaviour of a program under the single-threaded memory model is calculated by applying this function to the program, a thread-local semantics, the single-threaded model

condition and the single-threaded memory model. The behaviour of a given program according to the models in the rest of this chapter can be calculated similarly.

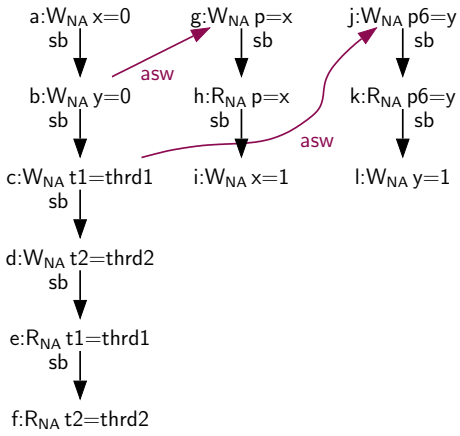
3.2 Multi-threaded programs with locks

This section introduces the *locks_only_memory_model* that allows the programmer to use multiple threads, making it possible to write concurrent programs. This introduces a new sort of fault: data races — unordered overlapping concurrent accesses to memory. Programmers are given mutexes, which allow locking of thread-local code, that can be used to create order between threads and avoid data races.

3.2.1 Thread creation syntax

The syntax for thread creation in C/C++11 is verbose, and its complexity would obscure relatively simple examples. The following example uses the standard thread creation syntax, defining a function, `foo`, that is run on two threads with different values. It is our convention in example executions to vertically align actions from a single thread.

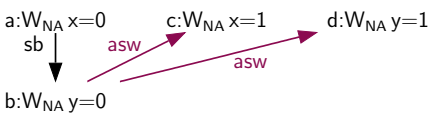
```
void foo(int* p) {*p=1;}

int main() {
  int x = 0;
  int y = 0;
  thread t1(foo, &x);
  thread t2(foo, &y);
  t1.join();
  t2.join();
  return 0; }
```



There are a great many accesses in the execution above whose sole purpose is bookkeeping: there are writes and dereferences of function pointers in thread creation, and loads and stores that pass references to variables on each thread. These accesses obscure the simplicity of the example. Instead of using this syntax, or C++11 lambda expression syntax, the examples that follow use an alternate more restrictive form that is not part of C/C++11. This supports structured parallelism only, which is sufficient for our purposes, clarifies example programs and executions, and elides uninteresting memory accesses. The new syntax consists of a fork of some number of threads that is opened with a triple left brace, {{{. Each thread is then expressed as an expression, delimited by a triple bar, |||, and the parallel composition is closed with triple right braces, }}}. The program above

becomes:

```
int main() {
  int x = 0;
  int y = 0;
  {{{ x = 1; ||| y = 1; }}};
  return 0; }
```



One consequence of this transformation is that executions of the transformed program lack non-atomic memory actions that would have been created in manipulating function pointers and accessing thread arguments. These actions are absent in the composition syntax, but because they are thread-local in the original program, and cannot be accessed from other threads, they cannot form races or synchronisation. Consequently, the transformation does not change the behaviour of the program.

There is a new constraint imposed by the structure of the program present in the executions above: *additional synchronises with*, asw. This new relation, identified by the thread-local semantics, records parent-to-child ordering introduced by thread creation and child-to-parent ordering induced by thread joining. The asw relation is a component of happens-before.

3.2.2 Data races and mutexes

As mentioned above, the sublanguage described in this section can be used to write programs with data races — one sort of fault that leaves a program with undefined behaviour. Two actions form a data race when they act on the same location from different threads, at least one of them is a write, and they are unordered by happens-before. The formal definition is given below:

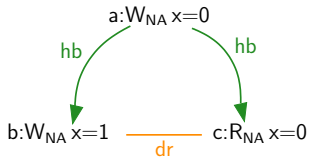
$$
\begin{aligned}
&\mathsf{let}\ \mathit{data\_races}\ (Xo,\ Xw,\ (\text{“}hb\text{”},\ hb) :: \_)\ = \\
&\quad \{\ (a,\ b)\ |\ \forall\ a \in Xo.\mathit{actions}\ b \in Xo.\mathit{actions}\ | \\
&\qquad \neg\ (a = b) \wedge (\mathrm{loc\_of}\ a = \mathrm{loc\_of}\ b) \wedge (\mathrm{is\_write}\ a \vee \mathrm{is\_write}\ b) \wedge \\
&\qquad (\mathrm{tid\_of}\ a \neq \mathrm{tid\_of}\ b) \wedge \\
&\qquad \neg\ (\mathrm{is\_atomic\_action}\ a \wedge \mathrm{is\_atomic\_action}\ b) \wedge \\
&\qquad \neg\ ((a,\ b) \in hb \vee (b,\ a) \in hb)\ \}
\end{aligned}
$$

There is a concrete architectural reason to have data races lead to undefined behaviour: each target architecture will implement non-atomic accesses differently, and for some types on some architecture it may be necessary to break up an access into several smaller accesses if, for instance, the object spans more bytes than the hardware can write or read in a single instruction. If two such accesses were made to a single object, and the constituent accesses were to interfere, even by simply interleaving, then that could result in corruption of the object. In a race-free program, these accesses can not interfere, avoiding the problem.

The compiler uses data-race freedom as an invariant in optimisation: it assumes that it can optimise non-atomic accesses as if no other thread is accessing them. This is very useful, because it allows the compiler to reuse sequential optimisations in the concurrent setting.

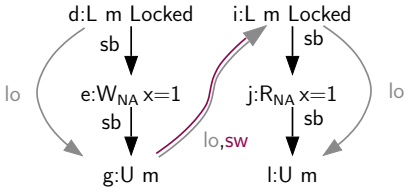
The following program gives rise to a consistent execution with a data race, shown on the right:

```
int main() {
  int x = 0;
  {{{ x = 1; ||| r1 = x; }}};
  return 0;
}
```



C/C++11 provide several mechanisms that allow multiple threads to communicate without introducing data races. The simplest of these is mutex locking and unlocking. Locks and unlocks act over mutex locations, and their behaviour is governed by a per-location total order over the locks and unlocks called *lock order* (lo). Rules in the consistency predicate ensure that regions of code that are locked and unlocked *synchronise-with* later regions in lock order. The following program differs from the previous example in that it has the racing actions inside locked regions. The execution shows one lock order over the actions at the mutex location. The locked regions synchronise, creating a new edge in the execution graph: synchronises-with, sw. The sw relation is part of happens-before, and happens-before is transitive in this model, so there is no race in this execution. We adopt the convention of eliding the actions of the parent thread when they are not relevant, as in this execution:

```
int main() {
  int x = 0;
  mutex m;
  {{{ { m.lock();
        x = 1;
        m.unlock(); }
  ||| { m.lock();
        r1 = x;
        m.unlock(); }
  }}};
  return 0;
}
```



Happens-before The new synchronises-with relation represents inter-thread synchronisation. Both thread creation and mutex actions can create sw edges. More precisely, all

additional-synchronises-with edges are in synchronises-with, and unlocks synchronise with all lock-order-later locks. The formal definition of synchronises-with is then a function:

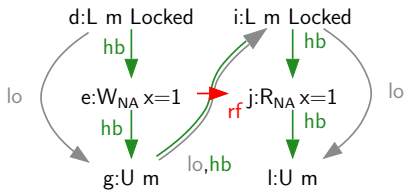
```
let locks_only_sw actions asw lo a b =
  (tid_of a ≠ tid_of b) ∧
  ( (* thread sync *)
    (a, b) ∈ asw ∨
    (* mutex sync *)
    (is_unlock a ∧ is_lock b ∧ (a, b) ∈ lo)
  )
```

Synchronisation creates happens-before edges: the calculation of the happens-before relation changes include the new synchronises-with edges, and the transitive closure:

```
let no_consume_hb sb sw =
    transitiveClosure (sb ∪ sw)
```

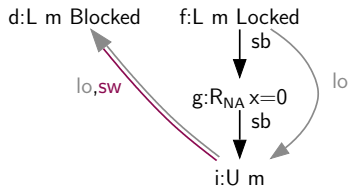
The execution in the following example shows the transitive reduction of the happens-before edges that exist because of thread-local sequenced-before and inter-thread synchronisation. Note that the write and read of x are now ordered, so this execution no longer has a data race.

```
int main() {
  int x = 0;
  mutex m;
  {{{ { m.lock();
        x = 1;
        m.unlock(); }
  ||| { m.lock();
        r1 = x;
        m.unlock(); }
  }}};
  return 0;
}
```



On IBM's Power architecture, mutexes are implemented with a load-linked/store-conditional loop, exiting when the store-conditional succeeds, acquiring the lock. The load-linked/store-conditional mechanism is implemented by monitoring the cache line containing the mutex. The cache line can be shared with other data, so seemingly-unrelated cache trafic can cause the store conditional to fail. This means there can be no formal guarantee of progress, and the lock can block arbitrarily, although in practice, processor designers try to avoid this. To make C/C++11 implementable above such architectures, each call to `lock()` may arbitrarily block for ever: the locks are implemented

with a loop involving load-linked and store-conditional accesses, and cache trafic can cause the loop to block indefinitely. To accomodate blocking, the thread-local semantics enumerates pre-executions with both the successful and blocked cases for each lock. A blocked lock causes the thread-local semantics to abbreviate the thread so that there are no sequenced-before successors. This is enforced by the *blocking_observed* predicate (§3.1). The following example represents an execution of the previous program where one of the locks blocks, and has no sb-successors.



3.2.3 Mutexes in the formal model

The consistency predicate restricts which lock orders can be observed in an execution with the *locks_only_consistent_lo* and *locks_only_consistent_locks* predicates, described below.

The *locks_only_consistent_lo* predicate restricts the lock order relation: it must be transitive and irreflexive, it must agree with happens-before, it must relate only locks and unlocks at the same mutex location, and it must be total over such actions:

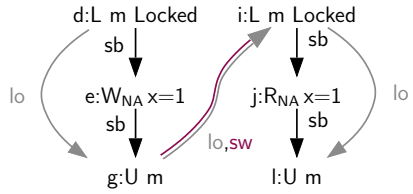
```
let locks_only_consistent_lo (Xo, Xw, ("hb", hb) :: _) =
  relation_over Xo.actions Xw.lo ∧
  isTransitive Xw.lo ∧
  isIrreflexive Xw.lo ∧
  ∀ a ∈ Xo.actions b ∈ Xo.actions.
   ((a, b) ∈ Xw.lo ⟶ ¬ ((b, a) ∈ hb)) ∧
   ( ((a, b) ∈ Xw.lo ∨ (b, a) ∈ Xw.lo)
     =
     ( (¬ (a = b)) ∧
       (is_lock a ∨ is_unlock a) ∧
       (is_lock b ∨ is_unlock b) ∧
       (loc_of a = loc_of b) ∧
       is_at_mutex_location Xo.lk a
     )
   )
```

The *locks_only_consistent_locks* predicate requires any pair of successful locks ordered by lock order to have an intervening unlock:

let *locks_only_consistent_locks* $(Xo,\ Xw,\ \_)\ =$
$(\forall\ (a,\ c) \in Xw.lo.$
is_successful_lock $a \wedge$ is_successful_lock c
$\longrightarrow$
$(\exists\ b \in Xo.actions.$ is_unlock $b \wedge (a,\ b) \in Xw.lo \wedge (b,\ c) \in Xw.lo))$

Returning to the previous example program and execution, we can see that it is consistent according to this criterion:

```
int main() {
  int x = 0;
  mutex m;
  {{{ { m.lock();
        x = 1;
        m.unlock(); }
  ||| { m.lock();
        r1 = x;
        m.unlock(); }
  }}};
  return 0;
}
```



Mutexes must be used correctly by programs, otherwise their behaviour will be undefined. C/C++11 impose a locking discipline that has two requirements. First, in every execution, all unlocks must be immediately preceded in lo by a lock that is sequenced before them. Second, no thread may lock a mutex and then perform another lock, of the same mutex, on the same thread without an intervening unlock of that mutex in lock order. These requirements are captured by the *locks_only_good_mutex_use* predicate:

let *locks_only_good_mutex_use actions lk sb lo a* $=$
`(* violated requirement: The calling thread shall own the mutex. *)`
(is_unlock a
$\longrightarrow$
($\exists\ al \in actions.$
is_successful_lock $al \wedge (al,\ a) \in sb \wedge (al,\ a) \in lo \wedge$
$\forall\ au \in actions.$
is_unlock $au \longrightarrow \neg\ ((al,\ au) \in lo \wedge (au,\ a) \in lo)$
)
) $\wedge$
`(* violated requirement: The calling thread does not own the mutex. *)`
(is_lock a

```
    ⟶
    ∀ al ∈ actions.
      is_successful_lock al ∧ (al, a) ∈ sb ∧ (al, a) ∈ lo
      ⟶
      ∃ au ∈ actions.
        is_unlock au ∧ (al, au) ∈ lo ∧ (au, a) ∈ lo
  )
```

A violation of this discipline by any execution of the program results in undefined behaviour — *locks_only_bad_mutexes* captures all of the actions in an execution that violate the locking discipline:

```
let locks_only_bad_mutexes (Xo, Xw, _) =
  { a | ∀ a ∈ Xo.actions |
    ¬ (locks_only_good_mutex_use Xo.actions Xo.lk Xo.sb Xw.lo a)}
```

The locks-only model condition This model applies to programs that use multiple threads with mutexes, but without atomic accesses. The model condition below precisely identifies the programs to which the model applies.

```
let locks_only_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    ∀ a ∈ Xo.actions.
      match (loc_of a) with
      | Nothing → false
      | Just l → (Xo.lk l ∈ {Mutex, Non_Atomic})
      end
```

3.3 Relaxed atomic accesses

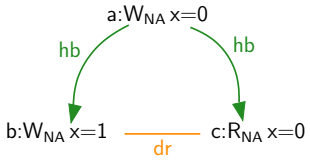
In the previous model, programs that contained data races had undefined behaviour. In multi-threaded programs, locks could be used to create synchronisation and avoid data races. C/C++11 cater to a variety of programmers, including systems programers who require high performance. For the highest performance, locking is too expensive and lock-free code is used, which may intentionally contain data races. C/C++11 provide atomic memory accesses with which to write racy programs.

This section describes the *relaxed_only_memory_model* that includes the least ordered atomic accesses, those with RELAXED memory order. Racing atomic accesses do not lead to undefined behaviour, and as a consequence, the definition of data races changes to exclude races between atomic accesses. Note that atomic initialisations are not themselves atomic accesses, so they can race with other accesses to the same location:

$$
\begin{aligned}
&\textsf{let } \mathit{data\_races}\ (Xo,\ Xw,\ (\text{“}hb\text{”},\ hb) :: \_)\ = \\
&\quad \{\ (a,\ b)\ |\ \forall\ a \in Xo.\mathit{actions}\ b \in Xo.\mathit{actions}\ | \\
&\qquad \neg\ (a = b) \wedge (\text{loc\_of}\ a = \text{loc\_of}\ b) \wedge (\text{is\_write}\ a \vee \text{is\_write}\ b) \wedge \\
&\qquad (\text{tid\_of}\ a \neq \text{tid\_of}\ b) \wedge \\
&\qquad \neg\ (\text{is\_atomic\_action}\ a \wedge \text{is\_atomic\_action}\ b) \wedge \\
&\qquad \neg\ ((a,\ b) \in hb \vee (b,\ a) \in hb)\ \}
\end{aligned}
$$

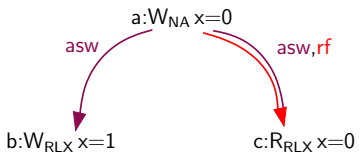
Returning to our previous example of a racy program with undefined behaviour:

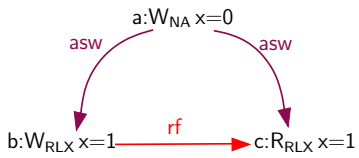
```
int main() {
  int x = 0;
  {{{ x = 1; ||| r1 = x; }}};
  return 0;
}
```



The program can be given defined behaviour by changing the shared location to an atomic one, and accessing it with atomic loads and stores:

```
int main() {
  atomic_int x = 0;
  {{{ x.store(1, relaxed);
  ||| r1 = x.load(relaxed);
  }}};
  return 0;
}
```





Atomic accesses allow the programmer to write racy code, so now the memory behaviour of racing relaxed atomic accesses must be defined. Now there are two possible executions of the program, both printed above: one where the load reads the value 0 and one where it reads 1. Note that in the execution on the right above, the read in the right-hand thread reads from the write in the left hand thread despite the absence of a happens-before edge from left to right. This would be forbidden for non-atomic accesses — they must read a visible side effect as in the execution on the left. Instead, reads of atomic locations are free to read any write that does not happen after them. This restriction forms a new conjunct of the consistency predicate:

let *consistent_atomic_rf* $(Xo,\ Xw,\ (\text{“}hb\text{”},\ hb) :: \_)\ =$
$\forall\ (w,\ r) \in Xw.rf.$ is_at_atomic_location $Xo.lk\ r \wedge$ is_load $r \longrightarrow$
$\neg\ ((r,\ w) \in hb)$

Modification order and coherence Most of the common target architectures (x86, Power, ARM, SPARC-TSO) provide a relatively strong guarantee about the ordering of writes to a single location in a program: they ensure that all of the writes at a single location appear to happen in a sequence, and that reads from any thread see the writes in an order consistent with that sequence. C/C++11 provides a similar guarantee over atomic locations: atomic actions are governed by a per-location total order over the writes, called *modification order*. Modification order does not contribute to happens-before, because that would rule out executions that are allowed on the Power architecture (see 2+2w in Section 3.3.1).

The following program has two possible executions, printed beneath. In each, there is a modification order edge between the writes on the two threads, but no happens-before edge. There is nothing to constrain the direction of modification order between the writes on the two threads, so there is an execution for either direction, and we could witness this direction with an observer thread that reads twice from `x`.

```
int main() {
  atomic_int x = 0;
  {{{ { x.store(1,relaxed); }
  ||| { x.store(2,relaxed); }
  }}}
  return 0;
}
```

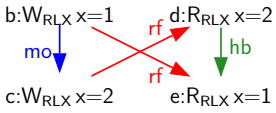
Modification order is a dynamic ordering of writes in memory, and is part of the execution witness. The *consistent_mo* predicate, given below, checks that modification order is a per-location strict total order over the writes:

let *consistent_mo* $(Xo,\ Xw,\ \_)\ =$
relation_over $Xo.actions\ Xw.mo \wedge$
isTransitive $Xw.mo \wedge$
isIrreflexive $Xw.mo \wedge$
$\forall\ a \in Xo.actions\ b \in Xo.actions.$
$((a,\ b) \in Xw.mo \vee (b,\ a) \in Xw.mo)$

$$= (\ (\neg\ (a = b)) \wedge$$
$$\text{is\_write}\ a \wedge \text{is\_write}\ b \wedge$$
$$(\text{loc\_of}\ a = \text{loc\_of}\ b) \wedge$$
$$\text{is\_at\_atomic\_location}\ Xo.lk\ a\ )$$

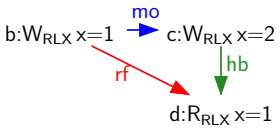
Although modification order does not directly contribute to happens-before, the two must be *coherent*. Intuitively, coherence is the part of the C/C++11 model that requires the reads of an atomic location to read from writes that are consistent with the modification order, as guaranteed by the processors. Coherence is defined as an absence of the following four subgraphs in an execution:

CoRR In the first subgraph, two writes are made, one before the other in modification order. Two reads, ordered by happens-before, are not allowed to read from the writes in the opposite order. The following execution fragment exhibits the forbidden behaviour:

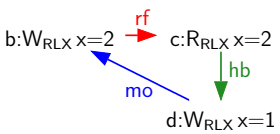


The other forbidden subgraphs can be thought of as derivatives of CoRR coherence where either one or both of the pairs of actions related by rf are replaced by a single write, and the subgraph is suitably contracted. First, we replace the reads-from edge that points to the first read to get CoWR coherence:

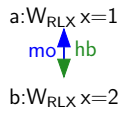
CoWR In this subgraph, there are two writes that are ordered by modification order. The later write happens before a read, and that read reads from the modification-order-earlier write. In this forbidden execution shape, the read is reading from a stale write when a more recent write in modification order happens before the read:



CoRW In this subgraph there is a cycle in the modification order, happens-before and reads-from relations. Modification order, happens-before and reads-from each have a temporal connotation, so it seems that they should not be cyclic, and indeed this is forbidden:



CoWW The final subgraph simply requires happens-before and modification order to agree. Execution fragments with opposing edges like the one below are forbidden:



These four execution fragments are forbidden in consistent executions by the *coherent_memory_use* predicate:

```
let coherent_memory_use (Xo, Xw, ("hb", hb) :: _) =
  (* CoRR *)
  ( ¬ ( ∃ (a, b) ∈ Xw.rf (c, d) ∈ Xw.rf.
        (b, d) ∈ hb ∧ (c, a) ∈ Xw.mo ) ) ∧
  (* CoWR *)
  ( ¬ ( ∃ (a, b) ∈ Xw.rf  c ∈ Xo.actions.
        (c, b) ∈ hb ∧ (a, c) ∈ Xw.mo ) ) ∧
  (* CoRW *)
  ( ¬ ( ∃ (a, b) ∈ Xw.rf  c ∈ Xo.actions.
        (b, c) ∈ hb ∧ (c, a) ∈ Xw.mo ) ) ∧
  (* CoWW *)
  ( ¬ (∃ (a, b) ∈ hb. (b, a) ∈ Xw.mo) )
```

Read-modify writes There are functions in the C/C++11 atomics library that allow the programmer to indivisibly read and write to a location in memory, providing the abilities like indivisibly testing and setting a flag, or incrementing an atomic location. we focus on the *compare-and-swap* (CAS) operation that allow one to indivisibly read and then write the memory. CAS takes four arguments:

atomic location the location of the atomic that the CAS acts on,

expected pointer a pointer to a memory location containing the value that must be read for the CAS to succeed,

desired value the value to write to the atomic location in the event of success,

failure memory order the memory order of the atomic read in the event of failure,

success memory order the memory ordering for the atomic access in the event of success.

A CAS reads an atomic location, checking for an expected value that is pointed to by the *expected* pointer. The CAS gives rise to two possible sequences of actions depending on whether it succeeds or fails. First the value at the expected location is read, and then

the atomic location is accessed. If the expected value is read at the atomic location, then the CAS writes the *desired* value to the atomic location. The read and write of the atomic location are made together in one atomic memory access called a *read-modify-write*. In the successful case, the CAS evaluates to *true*. If the value was not as expected, then the value read from the atomic location is written to the location pointed to in the *expected* argument, there is no write of the atomic location, and the CAS evaluates to *false*.

The third and fourth arguments of the CAS provide the memory orders of the accesses to the atomic location: in the case of a failure, the read of the atomic location is performed with the *failure* memory order, and on success, the read-modify-write-access is performed with the *success* memory order.

There are two sorts of CAS in C/C++11: weak CAS and strong CAS. A weak CAS may spuriously fail, even when the value read from the atomic location matches the value pointed to by *expected*. Strong CAS, on the other hand, fails only when the value read differs from the value at the *expected* location. Strong CAS must be implemented with a loop on the Power and ARM architectures. The body of the loop first does a load-linked, breaking out of the loop if the value does not match expected, and performing a store-conditional if it does. If that store-conditional fails, then the loop repeats, and if not then the CAS has succeeded. Unfortunately, accesses to memory cached in the same cache line as that of the CAS can cause the store-conditional to repeatedly fail, so there is no guarantee that it will eventually succeed, and the specification of the C/C++11 CAS must admit the possibility of blocking. Consequently, strong CAS calls in C/C++11 generate pre-executions where the CAS results in a blocked read-modify-write action in addition to pre-executions where it succeeds.

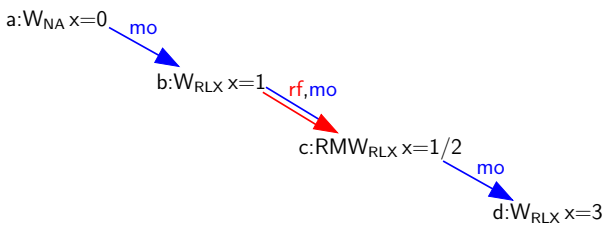
The read and write in a successful CAS operation are indivisible, and are represented by a read-modify-write action, that both reads from and writes to memory. In order to enforce atomicity, the memory model requires that the read-modify-write read from the immediately preceding write in modification order. A new conjunct to the consistency predicate, *rmw_atomicity*, given below, enforces this requirement:

$$\begin{array}{l}\mathsf{let}\ \mathit{adjacent\_less\_than}\ \mathit{ord}\ s\ x\ y\ = \\ \quad (x,\ y) \in \mathit{ord} \wedge \neg\ (\exists\ z \in s.\ (x,\ z) \in \mathit{ord} \wedge (z,\ y) \in \mathit{ord})\end{array}$$

$$\begin{array}{l}\mathsf{let}\ \mathit{rmw\_atomicity}\ (\mathit{Xo},\ \mathit{Xw},\ \_)\ = \\ \quad \forall\ b \in \mathit{Xo.actions}\ a \in \mathit{Xo.actions}. \\ \qquad \text{is\_RMW}\ b \longrightarrow (\text{adjacent\_less\_than}\ \mathit{Xw.mo}\ \mathit{Xo.actions}\ a\ b = ((a,\ b) \in \mathit{Xw.rf}))\end{array}$$

The following example illustrates an execution of the program where a successful CAS operation has given rise to a read-modify-write action in the execution. The RMW reads the immediately preceding write in modification order, as required by the predicate above. The accesses to e are elided:

```
int main() {
  atomic_int x = 0; int e = 1;
  {{{ { x.store(1,relaxed); }
  ||| { cas_weak_explicit(&x,&e,2,relaxed,relaxed); }
  ||| { x.store(3,relaxed); }
  }}}
  return 0;
}
```



Although adding relaxed atomic accesses allows many new complicated behaviours, the changes to the model are relatively modest. We have added modification order to the execution witness, but undefined behaviour, the calculated relations and the relations of the pre-execution remain the same. The consistency predicate changes to reflect the addition of the atomics and modification order.

3.3.1 Relaxed atomic behaviour

In this section we explore the C/C++11 memory model for relaxed atomics through a series of litmus tests, in the context of target processor architectures, compiler optimisations, and common programming idioms. We relate the tests to the hardware using the compiler mappings provided in Chapter 2. If these mappings are to be sound, any behaviour that the underlying processors allow for mapped analogous programs must be allowed by C/C++11 (see Chapter 7 for discussion of proofs that show the x86 and Power mappings are sound). The fragment of the mapping that applies to relaxed atomics is given below. Relaxed loads and stores map to plain loads and stores on x86, Power and ARM:

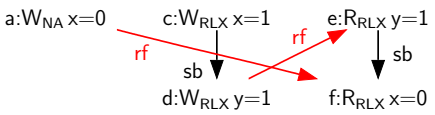
C/C++11	X86	Power	ARM	Itanium
`load` RELAXED	`MOV` (from memory)	`ld`	`ldr`	`ld.acq`
`store` RELAXED	`MOV` (into memory)	`st`	`str`	`st.rel`

The C/C++11 relaxed atomics are weaker than all three architectures, making the relaxed atomics implementable without adding any explicit synchronising. We will return to variants of these tests that use other memory orders as they are introduced.

Message passing, MP

The first example of relaxed behaviour is exhibited by the message-passing test, the first litmus test we considered in the introduction. In the version of the test below we omit the while loop on the read of the flag variable for simplicity, and we consider executions where the read of the flag variable `y` sees the write of 1. In the execution below, despite seeing the write of `y`, the second read fails to read from the write of `x` on the writer thread, and instead reads from the initialisation. The program demonstrates that a reads-from edge between relaxed atomics does not order a write with respect to a read on another thread.

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(1,relaxed);
        y.store(1,relaxed); }
  ||| { r1 = y.load(relaxed);
        r2 = x.load(relaxed); }
  }}}
  return 0;
}
```

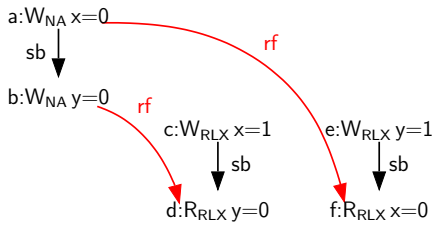


While the x86 architecture forbids this behaviour, Power and ARM allow it. On x86, the writes to `x` and `y` are placed in the thread local write buffer in order, and reach memory in order, so if the reading thread sees the write to `y`, then the write to `x` must have reached memory. On Power and ARM, the writes could be committed out of order, they could be propagated out of order, or the reads could be committed out of order, and each of these gives rise to the relaxed behaviour. A compiler could introduce this sort of behaviour by noticing that there are no thread-local dependencies and re-ordering the memory accesses on either the writing thread or the reading thread.

Store buffering, SB

In Chapter 2, the x86 memory model was discussed in terms of a concrete hypothetical micro-architecture: each processor has a write buffer between it and a global main memory. Writes in the buffer can be read directly by the associated processor, but cannot be seen by others until they are flushed to main memory. store-buffering is the only relaxed behaviour that x86 allows. The following example shows store buffering behaviour in C/C++11:

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(1,relaxed);
        r1 = y.load(relaxed); }
  ||| { y.store(1,relaxed);
        r2 = x.load(relaxed); }
  }}}
  return 0;
}
```

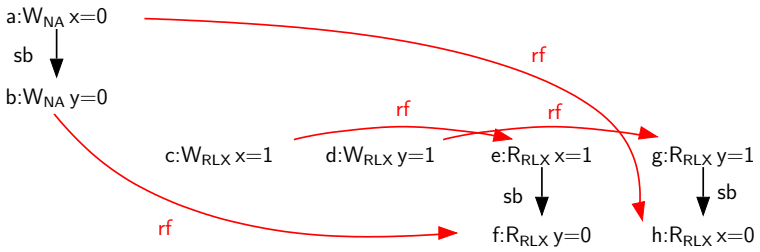


The x86 architecture allows this behaviour: each thread can buffer the write, and then read the initialisation writes from main memory. The analogous program also produces this execution on the Power and ARM architectures. There, the writes need not be propagated before the reads are committed. Again a compiler might spot that the accesses on each thread are unrelated and reorder, allowing the behaviour.

As discussed in Section 2.2, Dekker's algorithm for mutual exclusion relies on the absence of store-buffering behaviour: in its presence the algorithm can allow two clients to simultaneously enter the critical section.

Independent reads of independent writes, IRIW

In this test, presented by Collier [43], we observe a violation of multi-copy atomicity: we see the writes of the writer threads in two different orders on the reader threads. This test is the relaxed C/C++11 analogue of the Power and ARM test from Chapter 2, so the test includes dependencies that (if compiled naively) would prevent thread-local speculation on the hardware.



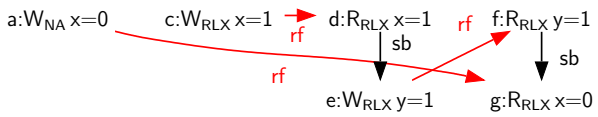

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ x.store(1,relaxed);
  ||| y.store(1,relaxed);
  ||| { r1 = x.load(relaxed);
        r2 = (*(&y+r1-r1)).load(relaxed); }
  ||| { r3 = y.load(relaxed);
        r4 = (*(&x+r3-r3)).load(relaxed); }
  }}}
  return 0;
}
```

Each reader thread sees the writes occur in a different order. The Power and ARM architectures allow this behaviour because they can propagate the writes to the reader threads individually and out of order. C/C++11 also violates multi-copy atomicity here: the execution above is allowed by the memory model. There is no speculation mechanism to disable in C/C++11, nor is there propagation — the execution is allowed because none of the rules of the model forbid it. Note that the dependencies in this example could be optimised away by a C/C++11 compiler.

Write-read causality, WRC

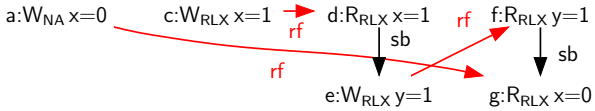
WRC (taken from Boehm and Adve [37]), is similar to MP, in that apparent causality implied by reading across threads is not enough to create order. This time the program demonstrates that a reads-from edge does not order two reads from the same location: the read on the centre thread sees the write on the first thread, whereas the read in the right-hand thread does not.

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ x.store(1,relaxed);
  ||| { r1 = x.load(relaxed);
        y.store(1,relaxed); }
  ||| { r2 = y.load(relaxed);
        r3 = x.load(relaxed); }
  }}}
  return 0;
}
```



In discussing cumulativity, Chapter 2 presented a variant of WRC that included an `lwsync` barrier as well as dependencies on the middle and right-hand threads. On the Power and ARM architectures, dependencies disable the thread-local speculation mechanism and this allowed us to consider the order of propagation of writes to different threads in executions of the example. We return to that example now, albeit without the barrier, written with relaxed atomics.

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ x.store(1,relaxed);
  ||| { r1 = x.load(relaxed);
        y.store(1+r1-r1,relaxed); }
  ||| { r2 = y.load(relaxed);
        r3 = (*(&x+r2-r2)).load(relaxed); }
  }}}
  return 0;
}
```



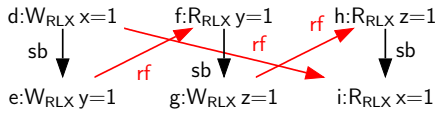
The relaxed outcome above was allowed on Power and ARM, even without thread-local speculation, because the write of `x` can propagate to the middle thread before the write of `y` committed, and then can be propagated after the write of `y` to the right-hand thread. In C/C++11, the dependencies can be compiled away, permitting even more architectural relaxed behaviour.

ISA2

We now consider ISA2, a simplified version of Example 2 from §1.7.1 of the 2009 Power ISA [7], and the second test in Chapter 2 that introduced cumulativity. Again, we consider

the C/C++11 analogue of the test, without the barrier:

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  atomic_int z = 0;
  {{{ { x.store(1,relaxed);
        y.store(1,relaxed); }
  ||| { r1 = y.load(relaxed);
        z.store(1+r1-r1,relaxed); }
  ||| { r2 = z.load(relaxed);
        r3 = (*(&x+r2-r2)).load(relaxed); }
  }}}
  return 0;
}
```

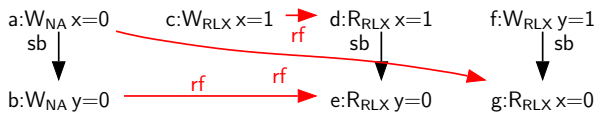


The relaxed outcome above was allowed on Power and ARM, even without thread-local speculation, because the write of `x` can propagate to the middle thread before the write of `y`. Again, the compiler can remove the dependencies, permitting more relaxed behaviour.

Read-write causality, RWC

In this test, from Boehm and Adve [37], the write of `x` on the left-hand thread is read by the middle thread, and the write of `y` on the right-hand thread is not read by the middle thread. The question is whether the right-hand thread should be allowed to read from the initialisation or not. This test is motivated by questions about the order of write propagation on the Power and ARM architectures, and in that regard it is related to WRC. As with WRC, we consider a variant of the test with dependencies that prevent thread-local speculation on the hardware.

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ x.store(1,relaxed);
  ||| { r1 = x.load(relaxed);
        r2 = (*(&y+r1-r1)).load(relaxed); }
  ||| { y.store(1,relaxed);
        r3 = x.load(relaxed); }
  }}}
  return 0;
}
```



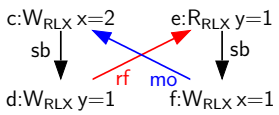
On Power and ARM, we know from the values that are read, that the write of `x` is

propagated to the middle thread before the write of `y`. Even so, the write of `x` can be propagated to the right-hand thread after the write has been committed, and an execution analogous to the one above is allowed. C/C++11 allows this behaviour.

S

S (taken from Maranget et al. [71]) tests whether reads-from and modification order between actions at different locations can form a cycle when combined with sequenced before. The execution below shows that executions with such a cycle are allowed by C/C++11.

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(2,relaxed);
        y.store(1,relaxed); }
  ||| { r2 = y.load(relaxed);
        x.store(1,relaxed); }
  }}}
  return 0;
}
```

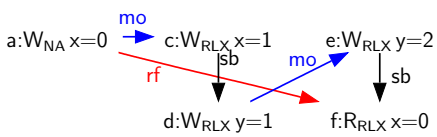


This behaviour is forbidden on x86, but observable on Power and ARM processors: because the writes on the left-hand thread are independent, they can be committed and propagated out of order, allowing this behaviour.

R

This test, taken from Maranget et al. [71], is similar to the message-passing test that checked whether a reads-from edge created ordering between threads, but we replace the reads-from edge with a modification-order edge. It checks whether a modification order edge is sufficient to force a write on one thread to be seen by a read on another.

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(1,relaxed);
        y.store(1,relaxed); }
  ||| { y.store(2,relaxed);
        r1 = x.load(relaxed); }
  }}}
  return 0;
}
```

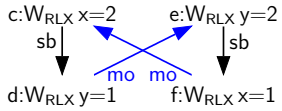


On Power and ARM, the analogue of this test, with coherence commitment order replacing modification order, can produce the relaxed behaviour. This could occur by speculating the read on the right-hand thread, or by propagating the writes out of order. This behaviour is allowed in C/C++11.

2+2W

This test, taken from Maranget et al. [71], highlights that the union of modification order across all locations is not part of happens-before. In this example, modification order union sequenced-before is cyclic. This relaxed behaviour is allowed on C/C++11 and again its analogue is allowed on Power and ARM just by local reordering.

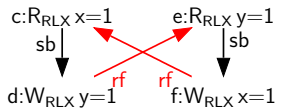
```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(2,relaxed);
        y.store(1,relaxed);}
  ||| { y.store(2,relaxed);
        x.store(1,relaxed);}
  }}}
  return 0;
}
```



Load buffering, LB

In this test, taken from Maranget et al. [71], two reads each appear to read from the future on their sibling thread, creating a cycle in happens-before union reads-from. This behaviour is allowed by C/C++11, Power and ARM, and can be observed on ARM processors. On Power and ARM, this is allowed by committing the accesses out of order on each thread.

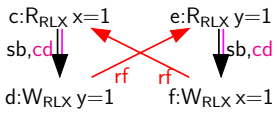
```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { r1 = x.load(relaxed);
        y.store(1,relaxed); }
  ||| { r2 = y.load(relaxed);
        x.store(1,relaxed); }
  }}}
  return 0;
}
```



Self satisfying conditionals, SSC

This test (taken from Section 29.3p11 of the C++11 standard [30]) is a variant of load-buffering where both writes are guarded by conditionals that are satisfied only if the relaxed behaviour is visible. This test is allowed in C/C++11 but forbidden on Power, ARM and x86. Each of the target architectures respects the control-flow dependency order, forbidding the execution.

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { if(x.load(relaxed))
          y.store(1,relaxed); }
  ||| { if(y.load(relaxed))
          x.store(1,relaxed); }
  }}}
  return 0;
}
```



C/C++11 is designed to allow compiler optimisations like common-subexpresion elimination (CSE) that remove control-flow dependencies. There is a tension between allowing this sort of optimisation and forbidding unintuitive behaviours like the one above. As it stands, the standard fails to define this part of the memory model well, and devising a better definition is a difficult problem. See Section 5.10.1 for details.

3.4 Simple release and acquire programs

So far, locks are the only mechanism that can be employed to synchronise multiple threads and avoid races between non-atomics in concurrent code. Locks create very strong synchronisation, and as a consequence require expensive explicit synchronisation on common architectures.

This section introduces the *release_acquire_memory_model* that includes lightweight inter-thread synchronisation through atomic memory accesses. The model presented here applies to programs whose atomic stores use the RELEASE memory order, whose atomic loads use the ACQUIRE order and whose CAS calls use ACQUIRE for failure and ACQ_REL for success.

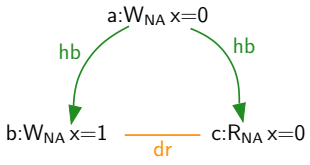
The compilation mappings for the release and acquire atomics introduce explicit synchronisation on the Power and ARM architectures, providing more ordering than the relaxed atomics at some cost to performance. The compiler mappings for the loads and

stores of this fragment of the language are[1]:

C/C++11	X86	Power	ARM	Itanium
`load` ACQUIRE	`MOV` (from memory)	`ld; cmp; bc; isync`	`ldr; teq; beq; isb`	`ld.acq`
`store` RELEASE	`MOV` (into memory)	`lwsync; st`	`dmb; str`	`st.rel`

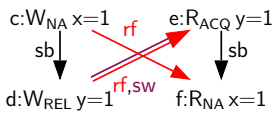
In the C/C++11 memory model, the RELEASE, ACQUIRE and ACQ_REL annotations on atomics cause atomic accesses to *synchronise*, creating happens-before edges coincident with reads-from edges across threads. This synchronisation can be used to avoid data races without the heavy cost of locks and unlocks. Take the example of a data race from Section 3.2:

```
int main() {
  int x = 0;
  {{{ x = 1; ||| r1 = x; }}};
  return 0;
}
```



Previously, locks and unlocks were placed around the racing accesses, creating happens-before between the critical sections and avoiding the race. The program below demonstrates how one can use release-acquire synchronisation to prevent the race using atomic accesses:

```
int main() {
  int x = 0; atomic_int y = 0;
  {{{ { x = 1;
        y.store(1,release); }
  ||| { while (y.load(acquire) == 1);
        r1 = x; }
  }}}
  return 0;
}
```



The fact that an acquire-read is reading from a release write creates the happens-before edge that avoids the race on `x`. The release-acquire atomics support programs that rely on the absence of MP behaviour.

In the Power analogue of this test, there is an `lwsync` between the writes on the left-hand thread that forces the writes to be committed and propagate in order. On the right hand thread, the mapping inserts a dependency followed by an `isync` barrier, and this disables the thread-local speculation mechanism. The ARM mapping inserts barriers that have the same effect, and neither architecture exhibits the relaxed behaviour. The x86

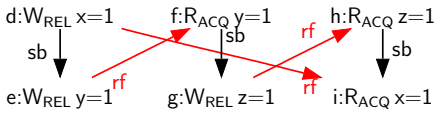
[1]The ARM V8 architecture includes `ld-acq` and `st-rel` instructions. These are strong enough to implement the C/C++11 release and acquire atomics, but they seem to be stronger than necessary, and to have worse performance than the implementation suggested here.

architecture does not exhibit the relaxed behaviour in the MP test. Compilers must not optimise in a way that makes the relaxed outcome possible.

With all atomics restricted to using release and acquire memory orders, much of the relaxed behaviour allowed in the previous section is now forbidden. Message-passing has already been discussed, but the other tests where a reads-from edge failed to provide ordering are now forbidden as well. In particular, ISA2, WRC, S, LB and SSC are no longer allowed. In each case, the barriers and dependencies inserted by the compiler mapping disable out-of-order commitment, out-of-order propagation and read speculation on the Power and ARM architectures. All other relaxed behaviour (SB, RWC, IRIW, R, 2+2W) is still allowed. Note that some of the remaining relaxed behaviours are no longer allowed by the hardware, but the language still allows them, 2+2W for instance.

Let us return to the ISA2 litmus test, amending it to use the release and acquire memory orders, and omitting the dependencies. In this subset of the language, each reads-from edge becomes a happens-before edge, and happens-before is transitive, so the write of `x` in the leftmost thread happens before the read of `x` in the rightmost thread, and the read must read the aforementioned write, rather than the initialisation:

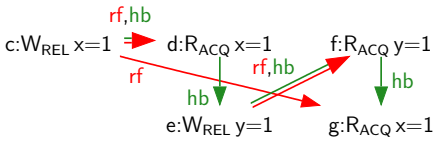
```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  atomic_int z = 0;
  {{{ { x.store(1,release);
        y.store(1,release); }
  ||| { r1 = y.load(acquire);
        z.store(1,release); }
  ||| { r2 = z.load(acquire);
        r3 = x.load(acquire); }
  }}}
  return 0;
}
```



Applying the mapping, the Power and ARM versions of this test have a barrier (`lwsync` or `dmb`, respectively) following the stores and a dependency following the loads, so cumulativity forbids the relaxed outcome on the hardware.

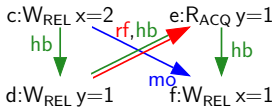
The transitivity of happens-before forbids the relaxed behaviour in the WRC litmus test too, and the Power and ARM mapped versions of the test are forbidden again because of cumulativity.

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ x.store(1,release);
  ||| { r1 = x.load(acquire);
        y.store(1,release); }
  ||| { r2 = y.load(acquire);
        r3 = x.load(acquire); }
  }}}
  return 0;
}
```



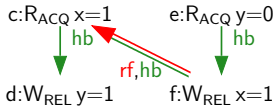
In the release-acquire analogue of S, the reads-from edge becomes a happens-before edge, ordering the two writes in happens-before, forcing modification order to agree, and forbidding the relaxed behaviour.

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(2,release);
        y.store(1,release); }
  ||| { r2 = y.load(acquire);
        x.store(1,release); }
  }}}
  return 0;
}
```



In LB and SSC, the two reads-from edges become happens-before edges, completing a cycle in happens-before. The cycle invalidates the execution: the reads now happen-before the writes that they read from, a violation of the consistency predicate.

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { r1 = x.load(acquire);
        y.store(1,release); }
  ||| { r2 = y.load(acquire);
        x.store(1,release); }
  }}}
  return 0;
}
```



In this model, no new relations have been introduced to the pre-execution or the

execution witness, and both the consistency predicate and the definition of undefined behaviour remain the same as in the relaxed-atomic model. The calculated relations do change, however, to incorporate the new happens-before edges, which are added to the synchronises-with relation. The new version of the synchronises-with relation is given below:

```
let release_acquire_synchronizes_with actions sb asw rf lo a b =
  (tid_of a ≠ tid_of b) ∧
  ( (* thread sync *)
    (a, b) ∈ asw ∨
    (* mutex sync *)
    (is_unlock a ∧ is_lock b ∧ (a, b) ∈ lo) ∨
    (* rel/acq sync *)
    ( is_release a ∧ is_acquire b ∧ (a, b) ∈ rf )
  )
```

Despite the fact that release-acquire programs still allow many sorts of relaxed behaviour, the most unintuitive and difficult behaviours have been forbidden. This leaves the programmer with a much simpler model that can be implemented with higher performance than locks.

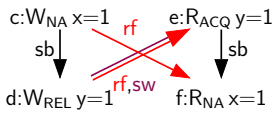
3.5 Programs with release, acquire and relaxed atomics

Because release and acquire atomics come with additional barriers in the compilation mappings, using relaxed accesses where possible will give higher performance on some architectures. This section presentes the *release_acquire_relaxed_memory_model*, that provides both relaxed atomics and release-acquire atomics, so that programmers can synchronise when necessary and use relaxed atomics when not.

3.5.1 Release sequences

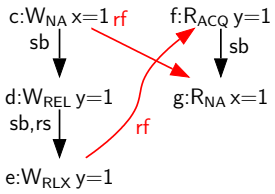
The integration of release and acquire memory orders with the relaxed memory order introduces another level of complexity to the model. So far, the naive combination of rules from the previous two models would be rather weak, and the target architectures provide stronger guarantees. Consider the message-passing example from the last section:

```
int main() {
  int x = 0; atomic_int y = 0;
  {{{ { x = 1;
        y.store(1,release); }
  ||| { while (y.load(acquire) == 1);
        r1 = x; }
  }}}
  return 0;
}
```



On the Power architecture, the compilation mapping places an `lwsync` barrier above the write to the atomic location. It is the fact that the barrier occurs between the non-atomic write and the atomic write that preserves the ordering on the hardware. Inserting a relaxed write to the same location after the release write gives us the following program (the rs edge will be explained below):

```
int main() {
  int x = 0; atomic_int y = 0;
  {{{ { x = 1;
        y.store(1,release);
        y.store(1,relaxed); }
  ||| { r1 = y.load(acquire);
        r2 = x; }
  }}}
  return 0;
}
```



The default Power compilation of the two child threads is given below:

`stw 1,0(x)`	`lwz r1,0(y)`
`lwsync`	`cmpw r1,r1`
`stw 1,0(y)`	`beq LC00`
`stw 1,0(y)`	`LC00:`
	`isync`
	`lwz r2,0(x)`

There is an `lwsync` barrier above the first write to the atomic location. The `lwsync` barrier forces the write to `x` to be committed and propagated to the other thread before program-order-later writes, and therefore does not just order the first write to the atomic location; it serves to order the second write to the atomic location as well. The `lwsync` prevents the out-of-order commitment or propagation of the write of `x` and either write of `y`. The branch-control-isync on the second thread inhibits read speculation. Consequently, it is not possible to see MP relaxed behaviour in the program above. Both ARM and

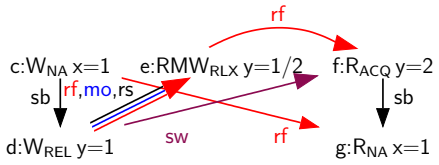
x86 also forbid MP relaxed behaviour in this example with their standard compilation mappings.

C/C++11 provide additional synchronisation to exploit this extra ordering that the hardware guarantees. The additional synchronisation is accounted for by a new calculated relation called the *release sequence*, that for every release-atomic includes some of the program-order-later writes that follow it. The release sequence relation, drawn as `rs` in the previous C/C++11 execution, relates each release to the actions that are part of the sequence. More precisely, each release-write heads a release sequence that contains a chain of modification-order-later writes from the same thread. The chain is broken immediately before the first store from another thread in modification order. This captures some of the writes that are program-order-after the `lwsync` in the compiled code.

Read-modify-writes get special treatment in release sequences, complicating the relation significantly.

The following example illustrates a release sequence that contains a read-modify-write on another thread. An acquire-read reads from the RMW action — because this write is in the release sequence of the release write on the leftmost thread, synchronisation is created between the write-release and the read acquire:

```
int main() {
  int x = 0; atomic_int y = 0;
  {{{ { x = 1;
        y.store(1,release); }
  ||| { cas_weak_explicit(&y,1,2,relaxed,relaxed); }
  ||| { r1 = y.load(acquire);
        r2 = x; }
  }}}
  return 0;
}
```



Here, the programmer has used the relaxed memory order with the CAS, so that on success, the read and write part of the resulting read-modify-write action will have relaxed semantics. With the above formulation of release-sequence, the acquire load will not synchronise with the RMW on the middle thread, and the RMW will not synchronise with the left-hand thread — so far, the release sequence just contains writes from the same thread as the head. As a consequence, this program would be racy, and its behaviour undefined.

Contrast this with the behaviour of this program on the usual processors: x86, Power

and ARM all guarantee that each respective analogous program would see the writes of `x` in the right-hand thread in executions where it reads 1 from `y`. To forbid the relaxed behaviour in the C/C++11 program, one would have to annotate the middle thread's CAS with acquire and release ordering in the event of success. This would induce superfluous synchronisation, and reduce performance. As a consequence, C/C++11 adds some RMW actions from other threads to the release sequence.

More precisely, the release sequence includes a contiguous subsequence of modification order that contains writes from the same thread as the head, and read-modify-writes from any thread. The sequence terminates before the first non-read-modify-write store on a different thread in modification order. The formal definitions that describe the release-sequence relation are given below:

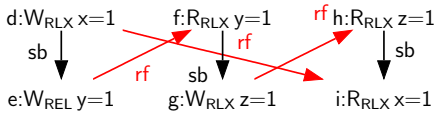
```
let rs_element head a =
  (tid_of a = tid_of head) ∨ is_RMW a
let release_sequence_set actions lk mo =
 { (rel, b) | ∀ rel ∈ actions b ∈ actions |
  is_release rel ∧
  ( (b = rel) ∨
   ( (rel, b) ∈ mo ∧
    rs_element rel b ∧
    ∀ c ∈ actions.
     ((rel, c) ∈ mo ∧ (c, b) ∈ mo) ⟶ rs_element rel c ) ) }
```

The model that covers this sublanguage makes no changes to the consistency predicate or undefined behaviour of the model from the previous section, but does make changes to the calculated relations. A new calculated relation, release-sequences, has been introduced. The release sequence introduces new synchronisation edges, so the definition of synchronises-with has to change to reflect this. Now, an acquire-read synchronises with the head of any release sequence that contains the write it reads from. The definition of synchronises-with below incorporates this change:

```
let release_acquire_relaxed_synchronizes_with actions sb asw rf lo rs a b =
 (tid_of a ≠ tid_of b) ∧
 ( (* thread sync *)
  (a, b) ∈ asw ∨
  (* mutex sync *)
  (is_unlock a ∧ is_lock b ∧ (a, b) ∈ lo) ∨
  (* rel/acq sync *)
  ( is_release a ∧ is_acquire b ∧
   (∃ c ∈ actions. (a, c) ∈ rs ∧ (c, b) ∈ rf) )
 )
```

Dependencies provide no ordering One might expect dependencies to provide a similar guarantee to Power and ARM cumulativity in this fragment of the model, but at the programming language level, the dependencies that one may rely on for ordering are carefully controlled, and they do not. Observe that ISA2 annotated with the release memory order on the writer thread, together with source-level dependencies allows the relaxed behaviour, despite the mapped instructions forbidding it on Power and ARM:

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  atomic_int z = 0;
  {{{ { x.store(1,relaxed);
        y.store(1,release); }
  ||| { r1 = y.load(relaxed);
        z.store(1+r1-r1,relaxed); }
  ||| { r2 = z.load(relaxed);
        r3 = (*(&x+r2-r2)).load(relaxed); }
  }}}
  return 0;
}
```



The WRC test is similar: to allow the compiler to optimise away dependencies, in C/C++11 a release write together with dependencies is not enough to create ordering, even though the relaxed behaviour is forbidden for the mapped Power and ARM code.

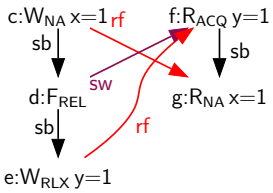
3.6 Programs with release-acquire fences

In the previous section, release sequences were introduced to allow programmers to write code that produced fewer barriers on relaxed architectures. The release-sequence mechanism was motivated by the observation that barriers on target architectures need not be tied to particular memory accesses. The *release_acquire_fenced_memory_model* introduced in this section goes a step further: it includes *fences* that expose barrier-style programming directly. The release and acquire fences map directly to barriers on the Power and ARM processors:

C/C++11	X86	Power	ARM	Itanium
`fence` ACQUIRE	⟨ignore⟩	`lwsync`	`dmb`	⟨ignore⟩
`fence` RELEASE	⟨ignore⟩	`lwsync`	`dmb`	⟨ignore⟩
`fence` ACQ_REL	⟨ignore⟩	`lwsync`	`dmb`	⟨ignore⟩

Release fences In the example in the previous section, relaxed writes in the release sequence were leveraging the barrier implicit in the head of the release sequence to ensure ordering. Fences let the programmer explicitly separate the synchronisation from the memory access. The following example uses explicit release-fence synchronisation:

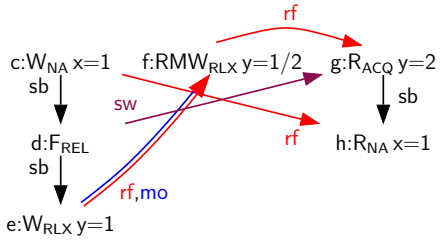
```
int main() {
  int x = 0; atomic_int y = 0;
  {{{ { x = 1;
        fence(release);
        y.store(1,relaxed); }
  ||| { r1 = y.load(acquire);
        r2 = x; }
  }}}
  return 0;
}
```



Here the fence is compiled to an `lwsync` on Power processors or a `dmb` on ARM, and the same architectural mechanism that created ordering in the release sequence example preserves ordering here. In C/C++11, the release fence paired with the relaxed write acts like a release write — they create a synchronises-with edge to the acquire read on the right-hand thread, forbidding the read of zero from `x`.

In a similar way to release writes, release fences provide synchronisation to read-modify-writes on different threads. The following example shows a release fence synchronising with an acquire read that reads a read-modify-write on another thread:

```
int main() {
  int x = 0; atomic_int y = 0;
  {{{ { x = 1;
        fence(release);
        y.store(1,relaxed); }
  ||| { cas_weak_explicit(&y,1,2,relaxed,relaxed); }
  ||| { r1 = y.load(acquire);
        r2 = x; }
  }}}
  return 0;
}
```



There is no release sequence in the example above, because there is no release write in the first thread. The memory model defines a new calculated relation, *hypothetical release sequence*, defined below, to describe the synchronisation that results from release fences. For a given write to an atomic location, the hypothetical release sequence identifies the actions that would be in the release sequence if the write were a release. Hypothetical release sequences give rise to synchronisation in the presence of release fences.

```
let rs_element head a =
  (tid_of a = tid_of head) ∨ is_RMW a

let hypothetical_release_sequence_set actions lk mo =
 { (a, b) | ∀ a ∈ actions b ∈ actions |
   is_atomic_action a ∧
   is_write a ∧
   ( (b = a) ∨
     ( (a, b) ∈ mo ∧
       rs_element a b ∧
       ∀ c ∈ actions.
         ((a, c) ∈ mo ∧ (c, b) ∈ mo) ⟶ rs_element a c ) ) }
```

If a release fence is followed in the same thread by the head of a hypothetical release sequence, and then an acquire action reads from the sequence, then synchronisation is created from the fence to the acquire. This synchronisation is captured by an updated version of the synchronises-with calculation, given below. There are three disjuncts corresponding to fence synchronisation, and the second covers synchronisation between a release-fenced write and an acquire read. The other two disjuncts describe the behaviour of acquire fences, as discussed below.

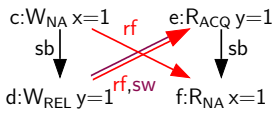
```
let release_acquire_fenced_synchronizes_with actions sb asw rf lo rs hrs a b =
  (tid_of a ≠ tid_of b) ∧
  ( (* thread sync *)
    (a, b) ∈ asw ∨
    (* mutex sync *)
    (is_unlock a ∧ is_lock b ∧ (a, b) ∈ lo) ∨
    (* rel/acq sync *)
    ( is_release a ∧ is_acquire b ∧
      (∃ c ∈ actions. (a, c) ∈ rs ∧ (c, b) ∈ rf) ) ∨
    (* fence synchronisation *)
    ( is_fence a ∧ is_release a ∧ is_fence b ∧ is_acquire b ∧
      ∃ x ∈ actions z ∈ actions y ∈ actions.
        (a, x) ∈ sb ∧ (x, z) ∈ hrs ∧ (z, y) ∈ rf ∧ (y, b) ∈ sb) ∨
```

$$
\begin{aligned}
&(\ \text{is\_fence}\ a \wedge \text{is\_release}\ a \wedge \text{is\_acquire}\ b \wedge \\
&\quad \exists\, x \in \mathit{actions}\ y \in \mathit{actions}. \\
&\qquad (a,\, x) \in \mathit{sb} \wedge (x,\, y) \in \mathit{hrs} \wedge (y,\, b) \in \mathit{rf}\ ) \vee \\
&(\ \text{is\_release}\ a \wedge \text{is\_fence}\ b \wedge \text{is\_acquire}\ b \wedge \\
&\quad \exists\, y \in \mathit{actions}\ x \in \mathit{actions}. \\
&\qquad (a,\, y) \in \mathit{rs} \wedge (y,\, x) \in \mathit{rf} \wedge (x,\, b) \in \mathit{sb})\ )
\end{aligned}
$$

Note that, unlike memory-order annotations on accesses, one fence can create memory ordering through many accesses to various locations.

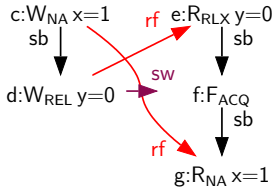
Acquire fences We have seen that release-acquire synchronisation can be used to implement the message-passing programming idiom. Our example used a while loop to check for the write of a flag variable. Recall the following program and execution:

```
int main() {
  int x = 0; atomic_int y = 0;
  {{{ { x = 1;
        y.store(1,release); }
  ||| { while (y.load(acquire) == 1);
        r1 = x; }
  }}}
  return 0;
}
```



Every iteration of the while loop performs an acquire-read of the flag variable. Acquire reads lead to the addition of an `isync` on Power and an `isb` on ARM each of which incur some synchronisation cost. It is unfortunate to have to repeatedly incur the cost of dependencies and barriers when, for synchronisation, only the last read must be an acquire. In the compiler mappings of Chapter 7, the Power architecture implements acquire reads as normal reads followed by an artificial control dependency and an `isync` barrier — referred to as a *ctrl-isync*. Unlike the `lwsync` barrier, discussed in the previous section, this barrier is tied to the access. Still, we need not have the ctrl-isync barrier in the loop. We can promote the ctrl-isync to a stronger `lwsync` barrier, and then we only need make sure that the barrier falls between the atomic read and the non-atomic read in order to preserve ordering. On Power, it would be sufficient to perform a single `lwsync` barrier after the loop, and before the read of the non-atomic. C/C++11 provides an acquire fence that allows programmers to do just that. The following program moves the acquire synchronisation outside of the loop in the standard message-passing idiom:

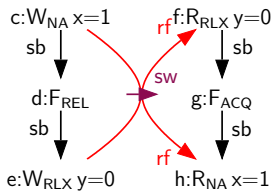
```
int main() {
  int x = 0; atomic_int y = 0;
  {{{ { x = 1;
        y.store(1,release); }
  ||| { while (y.load(relaxed) == 1);
        fence(acquire);
        r2 = x; }
  }}}
  return 0;
}
```



As the execution shows, the acquire fence synchronises with the release write, forbidding MP relaxed behaviour, while performing only low-cost relaxed writes in the body of the loop. Fence synchronisation is captured in the model by adding new edges to synchronises-with. This particular edge is added by the last of the fence disjuncts in the new definition of synchronises-with above. The read y and fence b act together in a similar way to an acquire read in normal release-acquire synchronisation. More precisely: a release action a synchronises with an acquire fence b if there are actions x and y such that x is in the release sequence of a, x is read from by y, and y is sequenced-before b.

Release and acquire fences Release and acquire fences can be used together in the message-passing example to create synchronisation, as in the following program:

```
int main() {
  int x = 0; atomic_int y = 1;
  {{{ { x = 1;
        fence(release);
        y.store(0,relaxed); }
  ||| { y.load(relaxed);
        fence(acquire);
        r1 = x; }
  }}}
  return 0;
}
```



The synchronisation is created by the first disjunct of the fence synchronisation part of synchronises-with above.

3.7 Programs with SC atomics

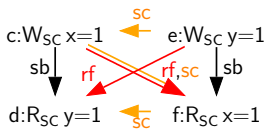
Programs written with the RELAXED, RELEASE and ACQUIRE memory orders admit relaxed behaviours, so only programs that are correct in the presence of those behaviours can be written without locks. Recall that relaxed atomics admitted a broad array of relaxed behaviours, and release-acquire atomics admit a proper subset of those: SB, RWC, IRIW, R, 2+2W. For programs that require an absence of any of these relaxed behaviours, including those that require full sequential consistency, C/C++11 provides the SEQ_CST (SC) memory order. This section introduces the *sc_accesses_memory_model* that includes SC atomic and the SC memory order. The SC atomics are mapped to machine instructions with explicit synchronisation on all architectures:

nocount

C/C++11	X86	Power	ARM	Itanium
`load` SEQ_CST	`MOV` (from memory)	`hwsync; ld; cmp; bc; isync`	`ldr; dmb`	`ld.acq`
`store` SEQ_CST	`MOV` (into memory), `MFENCE`	`hwsync; st`	`dmb; str; dmb`	`st.rel; mf`

The following example shows an execution of a program that would admit store-buffering with relaxed or release-acquire atomics. Here, with the SC memory order, the relaxed behaviour is forbidden:

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(1,seq_cst);
        r1 = y.load(seq_cst); }
  ||| { y.store(1,seq_cst);
        r2 = x.load(seq_cst); }
  }}}
  return 0;
}
```



Returning to the C/C++11 memory model, the execution above displays a new relation over the actions: sc. The sc relation is used to forbid non-sequentially consistent behaviour over the SC-annotated actions, and it is intended to provide a much stronger guarantee for full programs: data-race-free programs whose atomic accesses all have the SEQ_CST memory ordering have no relaxed behaviour — they behave in a sequentially consistent way.

The sc relation totally orders all actions in the execution of the program that are annotated with the SEQ_CST memory order. The sc relation must agree with happens-before and modification order. These requirements are enforced by a new addition to the consistency predicate, the *sc_accesses_consistent_sc* conjunct:

let *sc_accesses_consistent_sc* (*Xo*, *Xw*, (*"hb"*, *hb*) :: _) =
  relation_over *Xo.actions Xw.sc* ∧

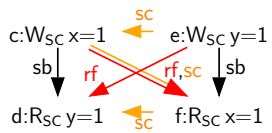
$$
\begin{array}{l}
\text{isTransitive } Xw.sc \wedge \\
\text{isIrreflexive } Xw.sc \wedge \\
\forall\ a \in Xo.actions\ b \in Xo.actions. \\
\quad ((a,\ b) \in Xw.sc \longrightarrow \neg\ ((b,\ a) \in hb \cup Xw.mo)) \wedge \\
\quad (\ ((a,\ b) \in Xw.sc \vee (b,\ a) \in Xw.sc) = \\
\qquad (\ (\neg\ (a = b)) \wedge \text{is\_seq\_cst } a \wedge \text{is\_seq\_cst } b) \\
\quad)
\end{array}
$$

The sc relation is used to restrict the values that may be read by SC reads in another new conjunct of the consistency predicate, *sc_accesses_sc_reads_restricted*, given below. This conjunct makes two separate restrictions on SC reads: one for SC reads that read from SC writes, and another for SC reads that read from non-SC writes.

$$
\begin{array}{l}
\text{let } sc\_accesses\_sc\_reads\_restricted\ (Xo,\ Xw,\ (\text{“}hb\text{”},\ hb) :: \_)\ = \\
\quad \forall\ (w,\ r) \in Xw.rf.\ \text{is\_seq\_cst } r \longrightarrow \\
\quad (\ \text{is\_seq\_cst } w \wedge (w,\ r) \in Xw.sc \wedge \\
\qquad \neg\ (\exists\ w' \in Xo.actions. \\
\qquad\quad \text{is\_write } w' \wedge (\text{loc\_of } w = \text{loc\_of } w') \wedge \\
\qquad\quad (w,\ w') \in Xw.sc \wedge (w',\ r) \in Xw.sc\ )\ ) \vee \\
\quad (\ \neg\ (\text{is\_seq\_cst } w) \wedge \\
\qquad \neg\ (\exists\ w' \in Xo.actions. \\
\qquad\quad \text{is\_write } w' \wedge (\text{loc\_of } w = \text{loc\_of } w') \wedge \\
\qquad\quad (w,\ w') \in hb \wedge (w',\ r) \in Xw.sc\ )\ )
\end{array}
$$

This adds two new restrictions to where SC reads may read from. First, if an SC read, r, reads from an SC write, w, then w must precede r in sc and there must be no other sc-intervening write to the same location. Returning to the store-buffering example, note that this restriction alone is not sufficient to guarantee sequentially consistent behaviour. Recall the example above:

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(1,seq_cst);
        r1 = y.load(seq_cst); }
  ||| { y.store(1,seq_cst);
        r2 = x.load(seq_cst); }
  }}}
  return 0;
}
```



The initialisation of the atomic locations is non-atomic, and non-atomic accesses are not related by the sc relation, so the first conjunct of *sc_accesses_sc_reads_restricted* does

not forbid the store-buffering relaxed behaviour. The second part is needed for SC writes that read from non-SC writes: if w is a non-SC write, then there must not be a write to the same location that happens after w that is sc-before r. With this restriction, store-buffering is forbidden, and as we shall see in Chapter 6, DRF programs without loops or recursion that use only SC atomics do indeed have SC behaviour. This theorem may hold with a weaker restriction, but the proof approach taken in Chapter 6 requires it.

As well as restricting the reads-from relation, SC atomics introduce synchronisation in the same way as release and acquire atomics. In order to reflect this, SC atomics are defined as release and acquire actions in the *is_release* and *is_acquire* functions.

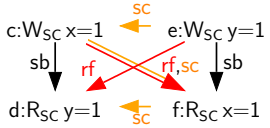
This model has introduced a new relation in the execution witness, as well as conjuncts of the consistency predicate that restrict the behaviour of SC accesses. All other elements of the memory model remain unchanged.

3.7.1 Examining the behaviour of the SC atomics

Now that the parts of the model that define the SC atomics have been introduced, we explore how they forbid the SB, RWC, IRIW, R and 2+2W relaxed behaviours.

Store-Buffering, (SB) We return to the store-buffering example of relaxed behaviour. In this example, there are two non-atomic (and therefore non-SC) initialisation writes that happen before all other actions. The rest of the actions in the execution take the SEQ_CST memory order and are therefore totally ordered by the sc relation. If there is an execution where both threads read from the initialisation writes, then the program admits store-buffering relaxed behaviour. This is not possible in this program, because any total order over the SC actions that agrees with happens-before must order one of the atomic writes before the read at the same location. This places that write after the initialisation in happens-before and before the read in sc, so according to *sc_accesses_sc_reads_restricted*, the atomic read may not read from the initialisation. The following example shows the program in question, and an execution that is allowed:

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(1,seq_cst);
        r1 = y.load(seq_cst); }
  ||| { y.store(1,seq_cst);
        r2 = x.load(seq_cst); }
  }}}
  return 0;
}
```

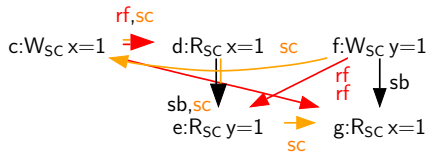


As explained in Chapter 2, store buffering is observable on x86 because the architecture

has thread-local store buffers. We needed to insert `MFENCE`s on each thread to force the buffers to flush and disallow the relaxed behaviour. Note that this is precisely what the compilation mapping specifies for the SC memory order in the test above. For Power and ARM the `sync` and `dmb` barriers specified by the mapping inhibit this behaviour on each.

Read-to-Write Causality, (RWC) When implemented with relaxed or release-acquire atomics, this program would admit relaxed behaviour where the write on the first thread is seen by the read on the second, but not seen by the read on the third thread, despite the read of y taking its value from the earlier write in modification order. With all SC actions, the values read on the middle and right-hand threads imply an sc order that orders the actions of the leftmost thread before the middle, and the middle before the rightmost. Then the initialisation write of x may not be read by the read on the third thread, and the behaviour is forbidden. The following execution shows this sc order:

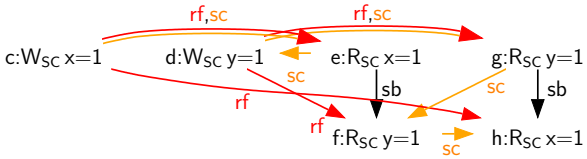
```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ x.store(1,seq_cst);
  ||| { r1 = x.load(seq_cst);
        r2 = y.load(seq_cst); }
  ||| { y.store(1,seq_cst);
        r3 = x.load(seq_cst); }
  }}}
  return 0;
}
```



The relaxed outcome of this test is forbidden for the mapped program on x86, Power and ARM.

Independent Reads of Independent Writes (IRIW) With weaker memory orders, this program produces a relaxed behaviour where the two reading threads see the accesses of the writing threads in opposite orders, each reading one and then zero. With SC atomics, the order of the writes is fixed one way or the other by the sc relation, and the reading threads cannot see the writes in two different orders:

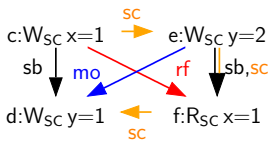
```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ x.store(1,seq_cst);
  ||| y.store(1,seq_cst);
  ||| { r1 = x.load(seq_cst);
        r2 = y.load(seq_cst); }
  ||| { r2 = y.load(seq_cst);
        r3 = x.load(seq_cst); }
  }}}
  return 0;
}
```



The relaxed outcome of this test is forbidden for the mapped program on x86, Power and ARM.

R This test checks whether a modification order edge is sufficient to order a write before a read on a different location. With weaker memory orders it is not, but here sc must agree with modification order, so ordering is created, forbidding the relaxed behaviour:

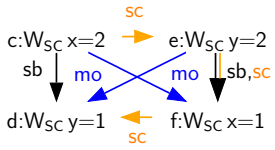
```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(1,seq_cst);
        y.store(1,seq_cst); }
  ||| { y.store(2,seq_cst);
        r1 = x.load(seq_cst); }
  }}}
  return 0;
}
```



The relaxed outcome of this test is forbidden for the mapped program on x86, Power and ARM.

2+2W This program tests whether modification order on one location requires modification ordering on another location to agree. Release-acquire or relaxed atomics admit relaxed behaviour for this test. With SC memory actions, the total sc order must agree with modification order, forbidding the cycle.

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(2,seq_cst);
        y.store(1,seq_cst);}
  ||| { y.store(2,seq_cst);
        x.store(1,seq_cst);}
  }}}
  return 0;
}
```



The relaxed outcome of this test is forbidden for the mapped program on x86, Power and ARM. In fact, Power is stronger than C/C++11 here, and requires only an `lwsync` between the writes on each thread rather than a full `sync`.

3.8 Programs with SC fences

In the compilation mapping, SC atomics have the most expensive implementations, using stronger explicit synchronisation than the other accesses. As a consequence, using SC atomic accesses to forbid relaxed behaviours comes with a significant performance penalty. This section introduces the *sc_fenced_memory_model*, that includes fences with the SEQ_CST memory order, allowing the programmer to forbid SB, R and 2+2W in a more lightweight way. Curiously though, because of details of the Itanium architecture, liberal use of SEQ_CST fences is not enough to regain full sequential-consistency. The mapping of SC fences is given below:

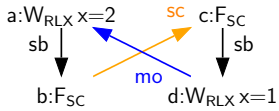
C/C++11	X86	Power	ARM	Itanium
`fence` SEQ_CST	`MFENCE`	`hwsync`	`dmb`	`mf`

The sequentially consistent fences appear in the `sc` relation with the rest of the SC atomics and impose ordering on reads-from and modification order. The conjunct in the consistency predicate that describes the behaviour of SC fences, *sc_fenced_sc_fences_heeded*, given below, is split up into six different cases: three covering fences interacting with reads-from edges, and three covering the interaction with modification order:

```
let sc_fenced_sc_fences_heeded (Xo, Xw, _) =
  ∀ f ∈ Xo.actions f' ∈ Xo.actions
      r ∈ Xo.actions
      w ∈ Xo.actions w' ∈ Xo.actions.
  ¬ ( is_fence f ∧ is_fence f' ∧
      ( (* fence restriction N3291 29.3p4 *)
```

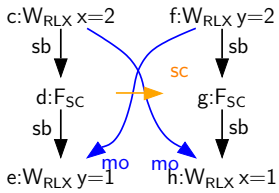
$$
\begin{aligned}
&((w, w') \in Xw.mo \wedge \\
&\quad (w', f) \in Xw.sc \wedge \\
&\quad (f, r) \in Xo.sb \wedge \\
&\quad (w, r) \in Xw.rf \ ) \vee \\
&\texttt{(* fence restriction N3291 29.3p5 *)} \\
&((w, w') \in Xw.mo \wedge \\
&\quad (w', f) \in Xo.sb \wedge \\
&\quad (f, r) \in Xw.sc \wedge \\
&\quad (w, r) \in Xw.rf \ ) \vee \\
&\texttt{(* fence restriction N3291 29.3p6 *)} \\
&((w, w') \in Xw.mo \wedge \\
&\quad (w', f) \in Xo.sb \wedge \\
&\quad (f, f') \in Xw.sc \wedge \\
&\quad (f', r) \in Xo.sb \wedge \\
&\quad (w, r) \in Xw.rf \ ) \vee \\
&\texttt{(* SC fences impose mo N3291 29.3p7 *)} \\
&((w', f) \in Xo.sb \wedge \\
&\quad (f, f') \in Xw.sc \wedge \\
&\quad (f', w) \in Xo.sb \wedge \\
&\quad (w, w') \in Xw.mo \ ) \vee \\
&\texttt{(* N3291 29.3p7, w collapsed first write*)} \\
&((w', f) \in Xw.sc \wedge \\
&\quad (f, w) \in Xo.sb \wedge \\
&\quad (w, w') \in Xw.mo \ ) \vee \\
&\texttt{(* N3291 29.3p7, w collapsed second write*)} \\
&((w', f) \in Xo.sb \wedge \\
&\quad (f, w) \in Xw.sc \wedge \\
&\quad (w, w') \in Xw.mo \ ) \ ) \ )
\end{aligned}
$$

First, two fences can be used to restrict modification order: for any fences f and f', and writes w and w' the consistency predicate requires that there is no cycle with the following shape (this corresponds to the fourth conjunct of *sc_fenced_sc_fences_heeded*):



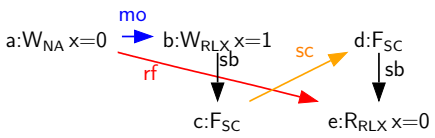
With this rule, SC fences can be used to forbid 2+2W relaxed behaviour, in which modification order over two locations takes part in a cycle with sequenced-before. The following program would exhibit 2+2W if it did not have fences in between the writes on each thread. With the fences, the program does not exhibit the behaviour:

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(2,relaxed);
        fence(seq_cst);
        y.store(1,relaxed);}
  ||| { y.store(2,relaxed);
        fence(seq_cst);
        x.store(1,relaxed);}
  }}}
  return 0;
}
```



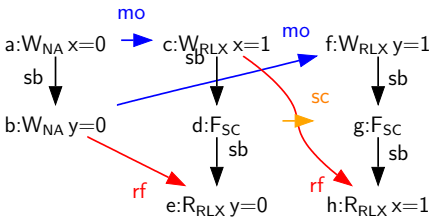
To see that it is forbidden, note that the sc relation is total over the SC actions in the program, so the fences are ordered one way or the other — left to right in the execution above. In this case, the modification order edge over the writes to x that would complete the cycle, from Thread 2 to Thread 1 would also complete a cycle through the sc edge of the fences, and according to the new restriction, executions with such cycles are forbidden.

Fences also restrict the ordering of rf edges across them in a similar way. In particular, given fences f and f', writes w and w' and a read r, the following shape is forbidden within an execution. This corresponds to the first conjunct of *sc_fenced_sc_fences_heeded*.:



Here the fact that the fences order the write before the read mean that the read cannot read from the modification-order-earlier write; it is hidden by the more recent one. This new restriction on the behaviour of executions with SC fences allows us to forbid store-buffering. The following program would exhibit store-buffering if the fences were removed, but with them the relaxed behaviour is forbidden:

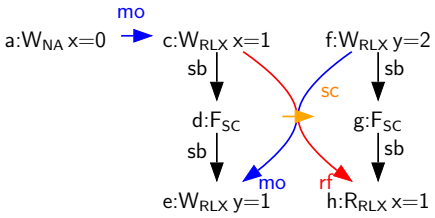
```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(1,relaxed);
        fence(seq_cst);
        r1 = y.load(relaxed); }
  ||| { y.store(1,relaxed);
        fence(seq_cst);
        r2 = x.load(relaxed); }
  }}}
  return 0;
}
```



Again, because sc is total, the fences must be ordered one way or the other — left to right in the execution above. The rf edge that would exhibit the relaxed behaviour in the execution would also complete the forbidden shape, so executions with the relaxed behaviour are forbidden by the fences.

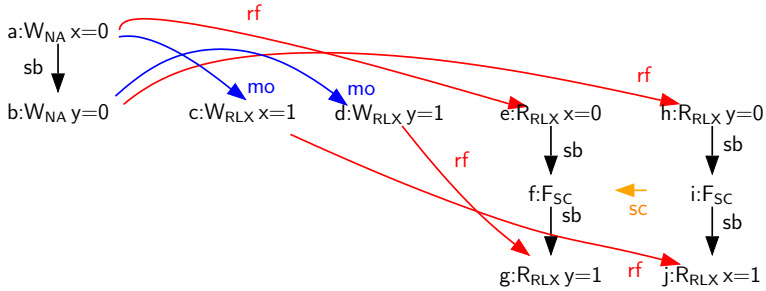
The relaxed behaviour R can be forbidden by SC fences, relying on both of the rules for SC fences. The following program would exhibit R without the fences, but here it is forbidden:

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(1,relaxed);
        fence(seq_cst);
        y.store(1,relaxed); }
  ||| { y.store(2,relaxed);
        fence(seq_cst);
        r1 = x.load(relaxed); }
  }}}
  return 0;
}
```



Again we know that the fences are ordered one way or the other in sc — left to right in the execution above. Depending on which way the fences are ordered, we can invoke the fence ordering of either modification order or reads-from to forbid the execution that contains the relaxed behaviour.

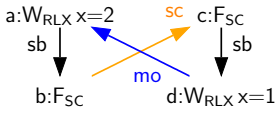
We have seen that SC fences can be used to forbid 2+2W, R and SB, but even inserting SC fences between every two actions is insufficient to rule out IRIW and RWC. Below is a consistent execution of an SC-fenced version of the IRIW litmus test. The SC order does not prevent the relaxed behaviour:




```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ x.store(1,relaxed);
  ||| y.store(1,relaxed);
  ||| { r1 = x.load(relaxed);
        fence(seq_cst);
        r2 = y.load(relaxed); }
  ||| { r2 = y.load(relaxed);
        fence(seq_cst);
        r3 = x.load(relaxed); }
  }}}
  return 0;
}
```

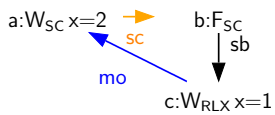
It is interesting to note that with the implementation of the SC fences (`MFENCE` on x86, `sync` on Power, and `dmb` on ARM), inserting a fence between every pair of memory accesses is sufficient to enforce sequential consistency on x86, Power and ARM. But the corresponding fence insertion allows relaxed behaviours on C/C++11. This seems to be a quirk of the language that accommodates an efficient implementation of SEQ_CST fences on the Itanium architecture [54], although rumour suggests that all Itanium implementations are stronger and do not exhibit relaxed behaviour with sufficient fences.

Fence rule derivatives The fences provide additional ordering when mixed with SC accesses of memory. In particular, recall that the consistency predicate forbids the following subgraph:

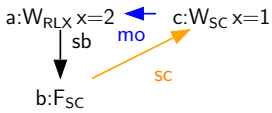


If we replace the left-hand thread's write and fence with a single SC write then we get the following shape, and the subgraph is still forbidden. This corresponds to the fifth

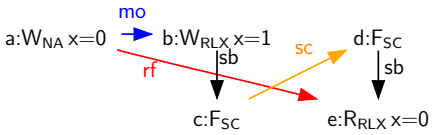
conjunct of *sc_fenced_sc_fences_heeded*.:



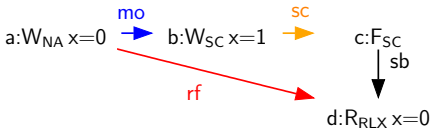
Similarly, replacing the fence and write on the right hand side leaves a different subgraph, and this subgraph is also forbidden. This corresponds to the sixth conjunct of *sc_fenced_sc_fences_heeded*.:



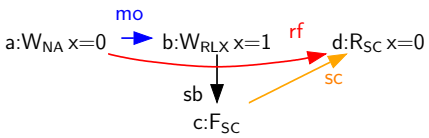
Similarly, recall that fences restrict the behaviour of reads-from by forbidding the following subgraph:



The subgraph with the left hand side replaced, corresponding to the second conjunct of *sc_fenced_sc_fences_heeded*, is forbidden:



So is the subgraph with the right hand side replaced, corresponding to the third conjunct of *sc_fenced_sc_fences_heeded*:



This model has introduced a new conjunct to the consistency predicate that restricts the behaviour of programs that use SC fences. All other elements of the memory model remain unchanged.

3.9 Programs with consume atomics

One of the important distinctions between the semantics of target processor architectures and the C/C++11 language is the differing treatment of dependencies. On processor architectures like Power and ARM, dependencies create ordering that forbids some relaxed

behaviours. In programming these systems, one can sometimes do away with expensive barriers, and rely on dependencies instead. In C/C++11, these dependencies may be compiled away because, so far, the memory model has not recognised them.

This section presents the *with_consume_memory_model* and introduces a new memory order, CONSUME, that the programmer can use to identify memory reads whose data and address dependencies should be left in place to provide memory ordering. Consume reads come with a very low-cost implementation on the Power and ARM architectures: they are implemented in the same way as relaxed atomic reads — no explicit synchronisation needs to be inserted. The only cost comes from the restriction that the compiler must leave all syntactic data and address dependencies of the read in place:

C/C++11	X86	Power	ARM	Itanium
`load` CONSUME	`MOV` (from memory)	`ld` + preserve dependencies	`ldr` + preserve dependencies	`ld.acq`

This approach has its limits: take for instance a program with a chain of dependencies that enters a separately compiled function. Dependencies in this function are required to provide ordering, so compiler optimisations must preserve all data and address dependencies in all functions that might be called separately, or emit hardware synchronisation if those dependencies might have been removed. The language provides a function attribute, `carries_dependency`, that the programmer can use to indicate that a function will be used in a context where its dependency propagation will be taken advantage of.

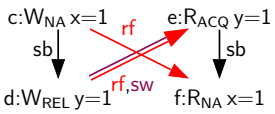
The preservation of dependencies comes at the cost of restricting some compiler optimisations or inserting hardware synchronisation. If a piece of code is never used with a consume atomic, or the ordering that it would provide is not needed, then the programmer can wrap that code with the `kill_dependency` function that tells the compiler to optimise anyway, losing the ordering guarantee that went with the dependency.

The ordering guarantees provided by the memory model are phrased in terms of the syntactic dependencies of the program, as calculated by the thread-local semantics, including function calls with dependency annotations and excluding dependencies that flow through calls to `kill_dependency`.

As well as the new language features for controlling dependencies, the addition of the consume memory order introduces substantial additional complexity to the model: happens-before, the partial order that provides the closest intuition to a global time ordering becomes non-transitive.

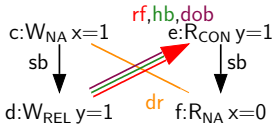
Dependency and dependency-ordered-before Recall the following program: an example of a release-write and an acquire-read creating synchronisation. Here we have synchronisation because the load of y is an acquire, the write of y is a release, and there is a reads-from edge between the two. The synchronisation that is created is extended through sequenced-before to order the write of x before the read of x.

```
int main() {
  int x = 0; atomic_int y = 0;
  {{{ { x = 1;
        y.store(1,release); }
  ||| { r1 = y.load(acquire);
        r2 = x; }
  }}}
  return 0;
}
```



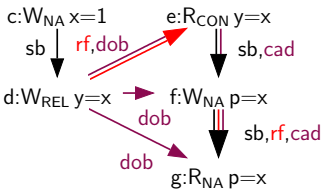
If we change the memory order of the read to be consume instead of acquire, we get different behaviour: the consume read still creates happens-before from the release write to the read consume, but happens-before is no longer extended through sequenced-before, so in the execution, there is no happens-before edge between the write and read of x:

```
int main() {
  int x = 0; atomic_int y = 0;
  {{{ { x = 1;
        y.store(1,release); }
  ||| { r1 = y.load(consume);
        r2 = x; }
  }}}
  return 0;
}
```



Consume reads provide transitive happens-before ordering only to those actions on the same thread that have a syntactic dependency on the consume read that is not explicitly killed, and where parts of the dependency travel through functions, those functions are annotated as dependency preserving. The following example program forbids message-passing relaxed behaviour using a release write, a consume-read, and the dependency from this read to the read of x:

```
int main() {
  int x = 0; atomic_address y = 0;
  {{{ { x = 1;
        y.store(&x,release); }
  ||| { int* p;
        p = y.load(consume);
        r2 = *p; }
  }}}
  return 0;
}
```



The consume atomic obliges the compiler to either emit a barrier, or leave the syntactic

and address dependencies of the consume-load in place. On the Power and ARM architecture, that would leave an address dependency in place in the compilation of the program above. This dependency is sufficient to disable the thread-local speculation mechanism and ensure that the relaxed behaviour is forbidden.

New calculated relations define the limited synchronisation that consume reads produce. The thread-local semantics identifies syntactic address and data dependencies and collects them together in the data-dependence relation, dd, that relates read actions to other memory actions that are dependent on their value (ignoring explicitly killed dependencies, and those that travel through un-annotated functions). A new calculated relation, *carries-a-dependency-to* (cad), is defined below as the union of data-dependence and the thread-local projection of the reads-from relation:

let *with_consume_cad_set actions sb dd rf* = transitiveClosure ($(rf \cap sb) \cup dd$)

Release-consume synchronisation is different to release-acquire synchronisation: it is transitive through the carries-a-dependency-to relation where the compiler is required to leave dependencies in place (or emit a barrier), but not through sequenced before, where the lack of dependencies allows reordering on the Power and ARM target architectures. These new release-to-consume synchronisation edges are called *dependency-ordered-before* edges, dob. We have seen the simple case where the consume-read reads directly from a release write, but the definition of dependency-ordered-before allows the consume to read from writes in a release sequence in a similar way to the synchronises-with relation. The definition of dependency-ordered-before, a new calculated relation, is given below:

let *with_consume_dob actions rf rs cad w a* =
  tid_of $w \neq$ tid_of $a \wedge$
  $\exists\, w' \in actions\ r \in actions.$
    is_consume $r \wedge$
    $(w, w') \in rs \wedge (w', r) \in rf \wedge$
    $(\ (r, a) \in cad \vee (r = a)\ )$

The construction of happens-before changes to incorporate these new dependency-ordered-before edges, and their incomplete transitivity. In this model, happens-before is constructed in two phases: first, we calculate *inter-thread happens-before* , ithb, and then from that we build happens-before.

Inter-thread happens-before, defined below, combines sequenced-before, synchronises-with and dependency-ordered-before edges, carefully omitting sb edges that transitively follow dob edges, while preserving all other transitivity. The omitted edges are precisely those where there is no control or data dependency to provide ordering on the underlying hardware following a consume atomic.

let *inter_thread_happens_before actions sb sw dob* =

let r = $sw \cup dob \cup$ (compose sw sb) in
transitiveClosure ($r \cup$ (compose sb r))

In this model, happens-before, as defined below, is simply inter-thread happens-before union sequenced before, without the transitive closure.

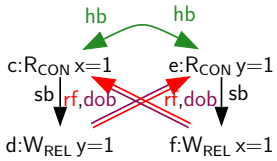
let *happens_before actions sb ithb* =
$sb \cup ithb$

Acyclicity of happens-before Cycles in happens-before are explicitly forbidden by the model. The *consistent_hb* predicate, given below, forbids cycles in ithb, and consequently in happens-before.

let *consistent_hb* (Xo, _, ("*hb*", *hb*) :: _) =
isIrreflexive (transitiveClosure *hb*)

Without the consume memory order, it was not necessary to forbid happens-before cycles explicitly, but with consume, it is possible to construct programs that would give rise to cycles in happens-before, if not explicitly forbidden. Consider the following load-buffering litmus test with consume memory orders on the reads:

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { r1 = x.load(consume);
        y.store(1,release); }
  ||| { r2 = y.load(consume);
        x.store(1,release); }
  }}}
  return 0;
}
```



Here, the consume-reads cause the creation of happens-before edges from the release-writes, but these edges are not transitive through to the write on the same thread. Transitivity from sequenced-before to dependency-ordered-before does give read-to-read happens-before in this example. This generates a cycle in happens-before between the reads, and this otherwise consistent execution violates *consistent_hb*.

Several new calculated relations have been added in this model, the definition of happens-before has changed, and the consistency predicate has a new conjunct. The calculation of undefined behaviour remains the same.

3.10 C/C++11 standard model

This section presents the *standard_memory_model*, a memory model with a tight correspondence to the text of the C++11 standard. This comes with downsides: there are

some elements of the model that are more complicated and difficult to understand than they need be. In particular, this model includes *visible sequences of side effects* , represented in the model by the calculated vsses relation, that restricts the permitted values of atomic reads.

Visible sequences of side effects are intended to identify the set of writes that a given atomic read may read from. Earlier revisions of the standard lacked the coherence requirements introduced in this chapter. Instead, the restriction imposed by vsses was supposed to suffice. Unfortunately, there was some confusion about the interpretation of the text that defined vsses, and some of the interpretations allowed strange behaviours that were unintended by the standardisation committee (see Chapter 5 for full details). The introduction of the coherence requirements makes the restrictions imposed by visible sequences of side effects redundant.

Visible sequences of side effects There is one sequence for each atomic read, that is represented in the model as a relation from members of the visible sequence of side effects to the read. For a given read, the set includes the modification-order-maximal visible side effect and the contiguous subsequence of modification-order-later writes, terminating just before the first write that happens after the read. The definition of the calculated relation, vsses is given below:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{standard\_vsses}\ \mathit{actions}\ \mathit{lk}\ \mathit{mo}\ \mathit{hb}\ \mathit{vse}\ = \\
\quad \{\ (v,\ r)\ |\ \forall\ r \in \mathit{actions}\ v \in \mathit{actions}\ \mathit{head} \in \mathit{actions}\ | \\
\quad\ \mathrm{is\_at\_atomic\_location}\ \mathit{lk}\ r \wedge (\mathit{head},\ r) \in \mathit{vse} \wedge \\
\quad\ \neg\ (\exists\ v' \in \mathit{actions}.\ (v',\ r) \in \mathit{vse} \wedge (\mathit{head},\ v') \in \mathit{mo}) \wedge \\
\quad\ (\ v = \mathit{head} \vee \\
\quad\quad (\ (\mathit{head},\ v) \in \mathit{mo} \wedge \neg\ ((r,\ v) \in \mathit{hb}) \wedge \\
\quad\quad\ \forall\ w \in \mathit{actions}. \\
\quad\quad\quad ((\mathit{head},\ w) \in \mathit{mo} \wedge (w,\ v) \in \mathit{mo}) \longrightarrow \neg\ ((r,\ w) \in \mathit{hb}) \\
\quad\quad\ ) \\
\quad\ ) \\
\quad\}
\end{array}
$$

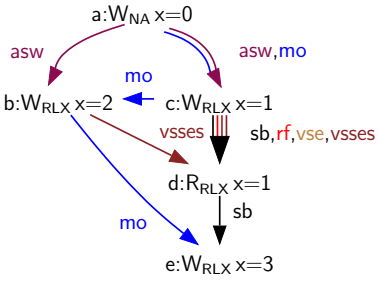
The atomic reads-from condition changes to require atomic reads to read from the visible sequence of effects. The *standard_consistent_atomic_rf* predicate below captures this requirement:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{standard\_consistent\_atomic\_rf}\ (\mathit{Xo},\ \mathit{Xw},\ \_ ::\ \_ ::\ \_ ::\ (\text{“}\mathit{vsses}\text{”},\ \mathit{vsses}) ::\ \_)\ = \\
\quad \forall\ (w,\ r) \in \mathit{Xw.rf}.\ \mathrm{is\_at\_atomic\_location}\ \mathit{Xo.lk}\ r \wedge \mathrm{is\_load}\ r \longrightarrow \\
\quad\quad (w,\ r) \in \mathit{vsses}
\end{array}
$$

The following examples illustrate the vsses relation. In the first example, consider the read on the right hand thread. The write of value 1 is the only visible side effect

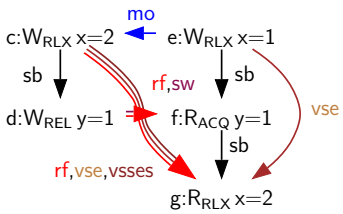
of the read. This write forms the head of the visible sequence of side effects, and is consequently related to the read by the vsses relation, as are each of the actions in the contiguous subsequence of modification order that follows the visible side effect, up to, but not including the first write that happens after the read. The vsses relation in this execution forbids the load of x from reading from the sb-following write:

```
int main() {
  atomic_int x = 0;
  {{{ x.store(2, relaxed);
  ||| {
        x.store(1, relaxed);
        r = x.load(relaxed);
        x.store(3, relaxed);
      }
  }}};
  return 0;
}
```



If there are multiple visible side effects of a given read, then the sequence only starts at the latest one in modification order. As a consequence, in the following execution, the store of 1 to x is not in the visible sequence of side effects:

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ {
        x.store(2, relaxed);
        y.store(1, release);
      }
  ||| {
        x.store(1, relaxed);
        r = y.load(acquire);
        r = x.load(relaxed);
      }
  }}};
  return 0;
}
```



The vsses relation is added to the calculated relations in this model, and the consistency predicate has changed. All other elements of the model remain the same.

Chapter 4

Explore the memory model with Cppmem

This chapter presents joint work with Scott Owens, Jean Pichon, Susmit Sarkar, and Peter Sewell.

The memory model presented in this thesis is relatively complex; understanding the definitions alone is an involved task. As a consequence, calculating the possible outcomes of small test programs by hand takes some time, and it is difficult to be sure that one has applied all of the rules correctly throughout every execution. In the early development of the model, happens-before could be cyclic, there were no coherence guarantees, the definition of visible-sequences of-side-effects was unclear, and seq_cst atomics did not behave as intended. This meant that incorrect executions were allowed, and calculating them by hand was time consuming and prone to error.

The model is written in mechanised formal mathematics, and its constituent predicates are computable, so we can write tools that use the model directly. This chapter describes Cppmem: an exhaustive execution generator for litmus test programs — given a small program, it returns all of the possible consistent executions of the program, and identifies any executions that contain faults. Cppmem presents these executions as graphs, and can display all of the relations of the model, including faults between actions that cause undefined behaviour.

Cppmem exhaustively calculates the set of executions, so the user is sure to see every corner case of their program, but this comes at the cost of speed: Cppmem becomes unusable on all but the smallest programs. As a consequence, it is of no use for testing realistic programs, but is useful for probing the memory model, for communication and for education. In particular, Cppmem has been used to teach masters students the memory model, it was invaluable during the development of the formal model, when prospective changes were being discussed and needed to be tested and explained to the standardisation committee, and it has been used by ARM, Linux and GCC developers to explore the

behaviour of the memory model.

4.1 Architecture

CPPMEM calculates the executions of a program in a series of steps that follow the structure of the memory model:

1. A thread-local semantics is used to calculate a set of pre-executions. Rather than enumerating a very large set with a pre-execution for every choice of read values, as in the model, CPPMEM calculates pre-executions with symbolic values constrained by conditionals in the chosen path of control flow.

2. For each pre-execution, the set of possible execution witnesses are enumerated, making read values concrete where there are reads from edges, creating a set of partially-symbolic candidate executions.

3. The consistency predicate is used to filter the candidate executions, keeping only those that are consistent.

4. For each remaining execution pair, all faults are calculated.

CPPMEM has a hard-coded thread-local semantics that accepts a very small subset of the C++11 language — just enough to write simple litmus tests. This thread-local semantics is embodied in a function that is recursive on program syntax, that is very similar to the thread-local semantics presented in Appendix 3.1.

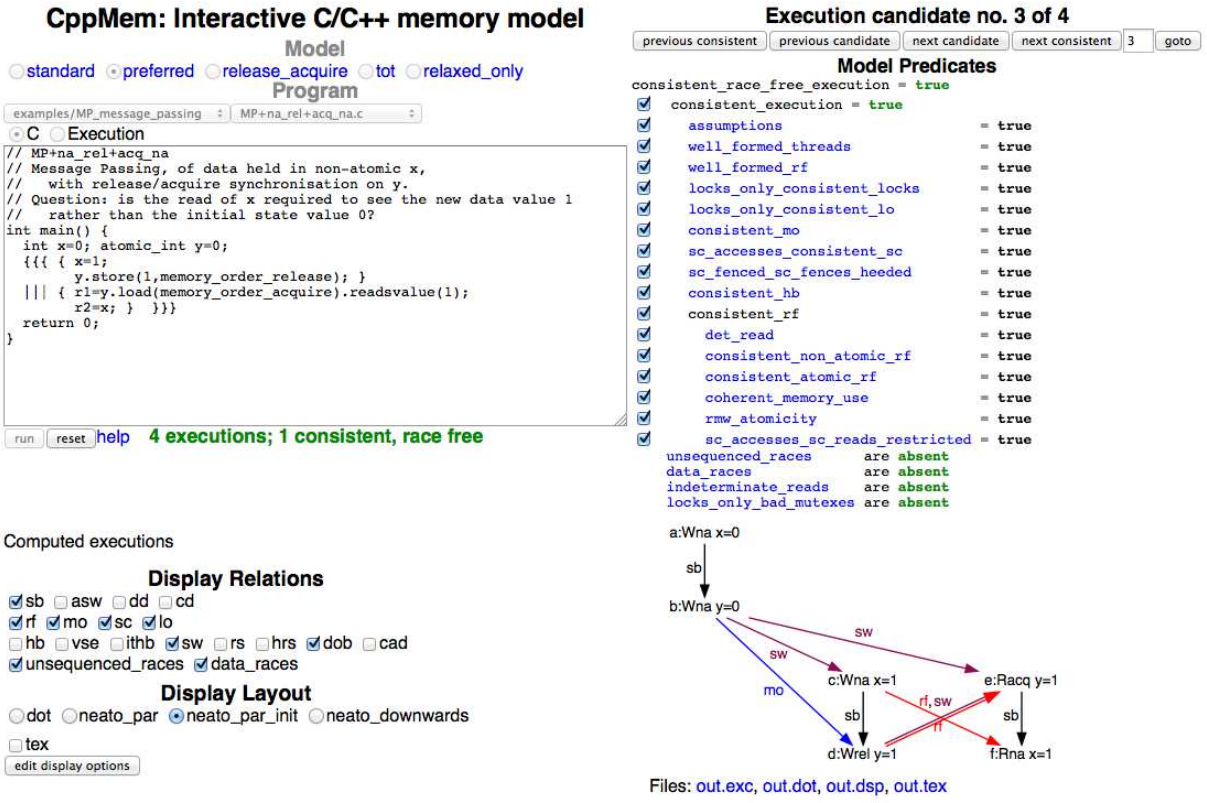
The enumeration of execution-witness relations for each pre-execution is intentionally naive, because the tool is designed to produce all candidate executions; the permutations of total relations like modification order or SC order are exhaustively enumerated with no pruning. This allows the user to explore inconsistent executions as well as consistent ones, but leads to a huge explosion of the search space of the tool, and accounts for its notably poor performance on programs with more than a handful of SC atomic accesses. More concretely, in the worst case, for each pre-execution of n actions, there are $(n!)^2$ candidate executions.

A different approach was explored with J. C. Blanchette, T. Weber, S. Owens, and S. Sarkar [32], where the Nitpick counterexample generator (based on an underlying SAT solver) was used to find consistent executions of litmus tests. Performance degraded on tests that only use relaxed atomics, but there was a significant performance improvement over CPPMEM on tests that use the SC atomics. Testing showed that CPPMEM scales well when using relaxed atomics, but when SC atomics are used, tests of 13 actions or more took longer than 10^4 seconds to process. On the other hand, Nitpick scaled well on the SC tests, processing 13-action tests in 40 seconds, and 22 action tests in 977 seconds.

4.2 Interface

The formal memory model is written in Lem, which is compiled to OCaml code. CPPMEM uses this code in its calculation of executions. A small change to the memory model can be incorporated into CPPMEM by rerunning Lem and recompiling the tool. CPPMEM can even provide a choice of several memory models for comparison.

CPPMEM presents executions as graphs — in fact, all of the execution graphs in this thesis are produced by CPPMEM. Together with graphical output, the tool has a convenient public web interface — we include a screenshot below:



The interface provides several useful features. At the highest level, it comprises four panels: a program text box (top left), an execution graph (bottom right), a summary of the calculation of the memory model over the current execution (top right), and a set of display options for the execution graph (bottom left).

Typical use of the web interface involves choosing a particular memory model from the radio button list at the top left (preferred corresponds to the with-consume-memory-model), and then either choosing a program from the drop down menus, or inserting one in to the text box. Pressing the "run" button causes the enumeration of executions, and on completion an execution graph of one of the candidate executions will appear in the bottom right.

At the top right, there are controls for exploring the enumerated executions. One can page through the candidate executions, or through the subset of those that is consistent. Below these controls, there are the predicates of the memory model, with an indication of whether they hold for the current execution. Each predicate name is a link that points to generated HTML that presents its definition. The toggle boxes allow the suppression of each predicate: if the box is un-checked, then the consistent executions are recalculated, ignoring any unchecked conjuncts.

The toggle boxes at the bottom left allow the user to show or hide relations in the execution, and the layout radio buttons choose different graph layout algorithms. There is a toggle box for exporting the Latex output of Cppmem in the execution graph, and one can edit more detailed display options by clicking the "edit display options" button.

At the bottom right, there are four links to different output files for the execution with four different file extensions, `exc`, `dot`, `dsp` and `tex`. The `tex` file is a Latex compatible representation of the graph. The `dot` file is a native format of Graphviz, the graph layout engine. The `dsp` file contains the display options that have been chosen. Finally, the `exc` file contains a representation of the current execution, and can be used as input to Cppmem in the left-side text input box by changing its radio button from "C" to "Execution".

Command-line batch mode In addition to the web interface, Cppmem has a command-line interface with a batch mode. This is useful both for printing a number of tests for display in a document, and for testing collections of tests. Some lightweight validation of the model was carried out by manually assembling a collection of litmus tests that follows the key Power and ARM tests, and then comparing the outcomes of those tests to what is allowed on the underlying processors.

Chapter 5

Design alterations and critique

This chapter discusses problems that were identified in the standard. The problems include serious omissions of important restrictions, inconsistencies in the definition and open questions in the design of relaxed-memory languages. We start with problems that were fixed before the standard was ratified, then discuss problems under consideration for future revisions, and conclude with remaining problems and open questions.

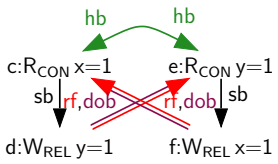
5.1 Acyclicity of happens-before

In early revisions of the standard, happens-before could be cyclic in some executions. This violated the core intuition of the model, and much of the consistency predicate did not make sense in the presence of a cycle. Without this restriction, the standard model from the previous chapter does admit executions with happens-before cycles. The standardisation committee was surprised to learn that cycles were allowed, and certainly did not intend to allow them.

The following example program and execution was discovered when considering the semantics of consume atomics. The execution below is undesirable because it has a cycle in happens-before, arising from the use of consume atomics.

The execution features load-buffering behaviour, but here the loads are annotated with consume memory orders and the stores take release order. As a consequence, dependency-ordered-before edges are generated from the stores to the loads, but they do not transitively carry through sequenced-before due to the lack of dependency:

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { r1 = y.load(consume);
        x.store(1, release); }
  ||| { r2 = x.load(consume);
        y.store(1, release); }
  }}}
  return 0;
}
```



Recall the definition of inter-thread-happens-before from the standard model:

```
let inter_thread_happens_before actions sb sw dob =
  let r = sw ∪ dob ∪ (compose sw sb) in
    transitiveClosure (r ∪ (compose sb r))
```

Happens-before is not transitive from dependency-ordered-before edges to sequenced before edges. This lack of transitivity stops the writes from being ordered after the reads at the same location (which would make the execution inconsistent). However,dependency-ordered-before edges are transitive through the composition of a sequenced before edge and another dependency-ordered-before edge, so in the example above, each read is related to the other in happens-before, and there is a cycle.

The issue was first highlighted in report N3125 [73] to WG21 for C++, and was taken to a meeting as a comment by the Great Britain national body: issue 10 from N3102 [51]. The comment suggested adding the following text to Section **1.10**:

1.10p12

> [...]The implementation shall ensure that no program execution demonstrates a cycle in the "happens before" relation. [*Note:* This would otherwise be possible only through the use of consume operations. — *end note*]

This change was accepted before ratification of C++11, and subsequently made its way into WG14 Defect Report #401 [22] for C, and was proposed for inclusion in a Technical Corrigendum. The issue is now marked as closed by WG14 [1], but it has not yet appeared in a C11 Technical Corrigendum

5.2 Coherence axioms

Early versions of the standard lack the CoWR and CoRW coherence requirements described in Section 3.3 and listed in Section **1.10p15-18** of the standard, and left CoWW coherence implicit in the definition of modification order. They rely on the restrictions imposed by visible-sequences-of-side-effects and an absence of CoRR coherence violations to dictate which writes a particular read can consistently read from. The corresponding memory model is essentially the same as the model presented in the last chapter, except imposing only the CoRR and CoWW parts of the *coherent_memory_use* requirement.

Depending on one's interpretation of the standard, the new coherence restrictions either make the memory model more restrictive, or they leave the model equivalent. The text in question is from paragraph **1.10p14**, the definition of visible-sequence-of-side-effects:

1.10p14

> The *visible sequence of side effects* on an atomic object M, with respect to a value computation B of M, is a maximal contiguous subsequence of side effects in the modification order of M, where the first side effect is visible with respect to B, and for every side effect, it is not the case that B happens before it. The value of an atomic object M, as determined by evaluation B, shall be the value stored by some operation in the visible sequence of M with respect to B. [...]

The ambiguity here is the sense in which the word "maximal" is used in the first sentence. Is this the maximal subsequence by inclusion, is it the sequence that is begun by the maximal visible-side-effect in modification order, or does it mean that the sequence must extend as far as possible from a given visible side effect? This question was posed to the standardisation committee at the very beginning of the process of formalisation. They replied that they meant maximal-by-inclusion. As a consequence, early models used this interpretation, and comments suggesting clarification of this, and the other uses of the word maximal throughout the standard were included in many documents submitted to the standardisation committee [29, 20, 74, 51, 83, 24]. In retrospect, and despite the response of the standardisation committee, this interpretation seems flawed.

Firstly, the standard uses precisely the same phrase in the definition of release sequences:

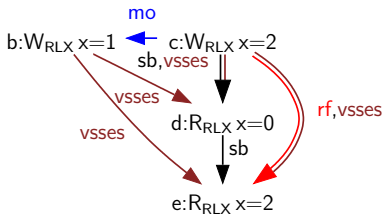
1.10p6

> A *release sequence* headed by a release operation A on an atomic object M is a *maximal contiguous subsequence of side effects in the modification order of* M, where the first operation is A, and every subsequent operation
>
> - is performed by the same thread that performed A, or
> - is an atomic read-modify-write operation.

Here, the committee was clear that they mean that the sequence should extend as far as possible in modification order for a given release.

Secondly, if the visible sequences of side effects of a particular read include the maximal-length subsequence, according to **1.10p14**, then some perverse executions are permitted. The following example would be allowed with the maximal-length interpretation of **1.10p14**, and without enforcing the CoRW and CoWR coherence requirements:

```
int main() {
  atomic_int x = 0;
  {{{ { x.store(1,relaxed); }
  ||| { x.store(2,relaxed);
        r1 = x.load(relaxed);
        r2 = x.load(relaxed);
      }
  }}}
  return 0;
}
```



Here, the initialisation write acts as the visible side effect that heads a visible sequence of side effects that contains both of the atomic writes. The loads each read from writes in their visible sequence of side effects, in accordance with CoRR coherence.

This execution is perverse: from the perspective of the second thread, it writes the location `x`, but fails to see that write locally, reading a modification-order-earlier write to the same location. This execution contains a CoWR coherence violation between actions `b`, `c` and `d`. The x86, Power and ARM architectures all forbid violations of CoWR coherence (in fact the coherence requirements follow hardware coherence [16], with thread-local program order replaced by happens-before), and allowing the compiler to reorder actions `c` and `d` would seem perverse. Furthermore, discussion with the C++11 standardisation committee suggested that they do not intend to allow this sort of execution [111].

This execution would not be allowed if we take the other interpretation of maximal in **1.10p14**: the visible sequence of side effects starts at the modification-order-maximal

visible side effect. Whichever interpretation one takes, the addition of the coherence axioms allows us to do away with visible sequences of side effects, and the confusion that they introduce. The coherence axioms neatly describe the intent of the standardisation committee, and even the models with visible sequences of side-effects impose both CoRR and CoWW coherence.

I suggested that the standardisation committee adopt the CoWR and CoRW coherence requirements, and that they make the CoWW requirement explicit. The changes to the wording of section **1.10** were suggested in N3136 [111], and are now part of the published standard. The change added the following text to Section **1.10**:

1.10p15–1.10p20

> If an operation A that modifies an atomic object M happens before an operation B that modifies M, then A shall be earlier than B in the modification order of M. [*Note:* This requirement is known as write-write coherence. — *end note*]
>
> If a value computation A of an atomic object M happens before a value computation B of M, and A takes its value from a side effect X on M, then the value computed by B shall either be the value stored by X or the value stored by a side effect Y on M, where Y follows X in the modification order of M. [*Note:* This requirement is known as read-read coherence. — *end note*]
>
> If a value computation A of an atomic object M happens before an operation B on M, then A shall take its value from a side effect X on M, where X precedes B in the modification order of M. [*Note:* This requirement is known as read-write coherence. — *end note*]
>
> If a side effect X on an atomic object M happens before a value computation B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M. [*Note:* This requirement is known as write-read coherence. — *end note*]
>
> [*Note:* The four preceding coherence requirements effectively disallow compiler reordering of atomic operations to a single object, even if both operations are relaxed loads. This effectively makes the cache coherence guarantee provided by most hardware available to C++ atomic operations. — *end note*]

[*Note:* The visible sequence of side effects depends on the "happens before" relation, which depends on the values observed by loads of atomics, which we are restricting here. The intended reading is that there must exist an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the "happens before" relation derived as described above, satisfy the resulting constraints as imposed here. — *end note*]

5.3 Visible sequences of side effects are redundant

Chapter 6 will describe mechanised proofs of a set of equivalences over the formal models of Chapter 3. One of the results will show that visible sequences of side effects are made redundant by the addition of the coherence axioms. This fact was presented to the C and C++ standardisation committees, and they both saw the value in the substantial simplification afforded by simply removing visible sequences of side effects. In C, Defect Report 406 [24] suggests this change, and in C++, the issue was raised in N3833 [9] and was incorporated into the draft of the next standard in N3914 [82]. The change to the wording is the same for both languages; the passage that defines the sequences is replaced with the following:

The value of an atomic object M, as determined by evaluation B, shall be the value stored by some side effect A that modifies M, where B does not happen before A.

In the model, the *standard_consistent_rf* predicate is replaced with:

$$\begin{aligned}&\textsf{let } \mathit{consistent\_atomic\_rf}\ (\mathit{Xo},\ \mathit{Xw},\ (\text{“}\mathit{hb}\text{”},\ \mathit{hb}) :: \_)\ =\\ &\quad \forall\ (w,\ r) \in \mathit{Xw.rf}.\ \text{is\_at\_atomic\_location}\ \mathit{Xo.lk}\ r \wedge \text{is\_load}\ r \longrightarrow\\ &\qquad \neg\ ((r,\ w) \in \mathit{hb})\end{aligned}$$

5.4 Erroneous claim of sequential consistency

The standard makes the claim that data-race-free programs whose atomics are all annotated with the `seq_cst` memory order will exhibit sequentially consistent behaviour:

29.3p8

> [*Note:* `memory_order_seq_cst` ensures sequential consistency only for a program that is free of data races and uses exclusively `memory_order_seq_cst` operations. [...]— *end note*]

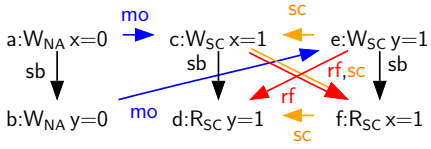
This property, articulated by Boehm and Adve [37], is central to the design of the memory model: programmers ought to be able to use a simplified model if they do not wish to use the more complicated high-performance concurrency features. Unfortunately, this claim was false before the introduction of the second disjunct in the implication of the *sc_accesses_sc_reads_restricted* predicate:

$$
\begin{aligned}
&\textsf{let } \mathit{sc\_accesses\_sc\_reads\_restricted}\ (Xo,\ Xw,\ (\text{“}hb\text{”},\ hb) :: \_)\ = \\
&\quad \forall\ (w,\ r) \in Xw.rf.\ \text{is\_seq\_cst}\ r \longrightarrow \\
&\quad\ (\ \text{is\_seq\_cst}\ w \wedge (w,\ r) \in Xw.sc \wedge \\
&\quad\quad \neg\ (\exists\ w' \in Xo.actions. \\
&\quad\quad\quad \text{is\_write}\ w' \wedge (\text{loc\_of}\ w = \text{loc\_of}\ w') \wedge \\
&\quad\quad\quad (w,\ w') \in Xw.sc \wedge (w',\ r) \in Xw.sc\ )\ ) \vee \\
&\quad\ (\ \neg\ (\text{is\_seq\_cst}\ w) \wedge \\
&\quad\quad \neg\ (\exists\ w' \in Xo.actions. \\
&\quad\quad\quad \text{is\_write}\ w' \wedge (\text{loc\_of}\ w = \text{loc\_of}\ w') \wedge \\
&\quad\quad\quad (w,\ w') \in hb \wedge (w',\ r) \in Xw.sc\ )\ )
\end{aligned}
$$

This part of the predicate restricts reads annotated with the `seq_cst` memory order that read from writes that are not annotated with the `seq_cst` memory order. Such reads must not read from a write that happens before an SC write to the same location that precedes the read in SC order.

Without this restriction, the model allows executions that are direct counterexamples of the theorem from **29.3p8**. The example below (discovered when attempting a prove the theorem), is the familiar store-buffering example, with all atomic accesses annotated with the `seq_cst` memory order. In the execution on the right, the reads read from the initialisation writes despite the strong choice of memory order.

```
int main() {
  atomic_int x = 0; atomic_int y = 0;
  {{{ { x.store(1,seq_cst);
        r1 = y.load(seq_cst);
      }
  ||| { y.store(1,seq_cst);
        r2 = x.load(seq_cst);
      }
  }}}
  return 0;
}
```



This execution exhibits non-sequentially consistent behaviour, so for the model without the additional restriction on SC reads, this serves as a counterexample to the claim in the standard. Early drafts of the standard made this claim of sequentially consistent behaviour yet failed to restrict `seq_cst` reads on non-`seq_cst` writes sufficiently.

The counterexample to the claim of sequential-consistency was presented in POPL [28], and I discussed possible solutions with members of the C++11 standardisation committee. The conclusion of that discussion was then summarised by Hans Boehm in issue 2034 of the defect report N3822 [80], including a change of wording to paragraph **29.3p3** of the standard. The layout of the text was then refined in N3278 [45] into the form that exists in the published standard. The changes added the second bullet of the list in **29.3p3**:

29.3p3

> There shall be a single total order S on all `memory_order_seq_cst` operations, consistent with the "happens before" order and modification orders for all affected locations, such that each `memory_order_seq_cst` operation B that loads a value from an atomic object M observes one of the following values:
>
> - the result of the last modification A of M that precedes B in S, if it exists, or
> - if A exists, the result of some modification of M in the visible sequence of side effects with respect to B that is not `memory_order_seq_cst` and that does not happen before A, or
> - if A does not exist, the result of some modification of M in the visible sequence of side effects with respect to B that is not `memory_order_seq_cst`.

This solution is sufficient to return sequential consistency to programs that use only `seq_cst` atomics with some additional restrictions. This property is expressed formally in

Lem with a mechanised proof in HOL, and is presented in detail in Chapter 6, formally validating this central design goal of the memory model.

This solution restricts `seq_cst` reads from non-`seq_cst` writes using the happens-before relation. This is sufficient to provide sequentially consistent behaviour to some programs, as desired, but it is a weaker than it might be, and is not symmetrical with the restrictions on `seq_cst` fences. Recall that `seq_cst` fences restrict the reads-from relation, forbidding reads that follow a `seq_cst` fence in sequenced before from reading a *modification order* predecessor of earlier `seq_cst` writes. The restriction on `seq_cst` imposed in **29.3p3** is weaker, formulating its restriction with happens-before rather than modification order. If the model were strengthened to use modification order here, it could be simplified, and the compilation mappings of Chapter 2 would remain correct.

5.5 Missing SC-fence restrictions

As discussed in Chapter 3, the standard seems to miss some restrictions on executions that use SC fences. The additional restrictions are imposed in the model in the *sc_fenced_sc_fences_heeded* predicate.

Discussions with the C and C++ standardisation committees have been positive, and they believe these restrictions were simply omitted. C Defect Report #407 [21], and C++11 Defect Report issue 2130 [23] include the suggestion that the following two passages are added to the standard:

> For atomic modifications A and B of an atomic object M, if there is a `memory_order_seq_cst` fence X such that A is sequenced before X, and X precedes B in S, then B occurs later than A in the modification order of M.

> For atomic modifications A and B of an atomic object M, if there is a `memory_order_seq_cst` fence Y such that Y is sequenced before B, and A precedes Y in S, then B occurs later than A in the modification order of M.

Following discussion and redrafting the change has been accepted for incorporation in the draft of the next revision of the C++ standard [23].

5.6 Undefined loops

In C/C++11 programmers are required to avoid programs that have undefined behaviour, and in return the standard carefully defines how the remaining programs will behave. In this way, undefined behaviour becomes a set of restrictions on the sorts of program that can be written in the language. On the whole, the standard is relatively fair, and programs with undefined behaviour are programs one would not wish to write. Paragraph **1.10p24** goes further, and forbids a whole host of useful programs:

1.10p24

> The implementation may assume that any thread will eventually do one of the following:
>
> - terminate,
> - make a call to a library I/O function,
> - access or modify a volatile object, or
> - perform a synchronization operation or an atomic operation.
>
> [*Note:* This is intended to allow compiler transformations such as removal of empty loops, even when termination cannot be proven. — *end note*]

Although it does not explicitly use the words "undefined behaviour", this paragraph requires programmers to make sure their programs meet the criteria. There is no explicit definition of the behaviour of a program that does not meet this requirement, and Section **1.3.24** of the C++11 standard makes it clear (as did the standardisation committee) that such programs have undefined behaviour:

1.3.24

> [. . .]Undefined behavior may be expected when this International Standard omits any explicit definition of behavior or when a program uses an erroneous construct or erroneous data.[. . .]

Unfortunately, this stops the programmer from writing many reasonable programs, and is likely to provide undefined behaviour to many existing programs. For example: imagine a proof assistant that is asked to prove an assertion that is in fact false. In some executions, this proof assistant will get lost in a never-ending search for a proof, as they are wont to do. If in its search, it performs no calls to library I/O, does not access a

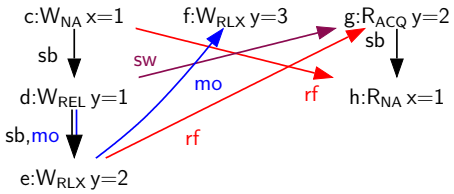
volatile object, and performs no synchronisation or atomic accesses, then returning "true" is a behaviour that conforms to the standard. This is perverse.

Despite protests, this paragraph remains in the standard. It is a monument to the importance the standardisation committee places on sequential compiler optimisations. The paragraph exists to enable the optimiser to move code across loops without the need to prove that they terminate. This is a pragmatic position: a standard that required major changes to existing compilers or harmed performance would simply be ignored. It is unfortunate that the specification does not model the intended behaviour here, and instead provides undefined behaviour. This leaves the specification sound, but it means we cannot use it to reason about some reasonable programs.

5.7 Release writes are weaker than they might be

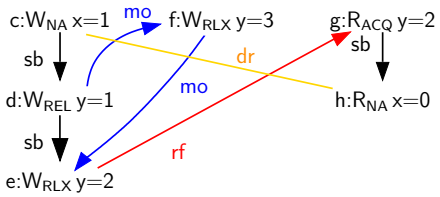
The semantics of release writes is largely provided by the definition of release sequences, the relation that allows reads from modification-order-later writes to synchronise with prior release writes. The implementability of this synchronisation mechanism relies on details of the Power and ARM processors that happen to provide more ordering than the language exploits. Consider the following example that will illustrate this discrepancy:

```
int main() {
  int x = 0; atomic_int y = 0;
  {{{ { x = 1;
        y.store(1,release);
        y.store(2,relaxed); }
  ||| { y.store(3,relaxed); }
  ||| { y.load(acquire);
        r1 = x; }
  }}}
  return 0;
}
```



We will consider several executions of this program. The first is the execution above where Thread 3 reads from the relaxed write on Thread 1 and the write on Thread 2 is not modification-order-between the release and the relaxed writes on thread 1. In this execution, the relaxed write on Thread 1 is in the release sequence of the release write, the acquire load synchronises with the release on Thread 1, and there is no data race. The correct behaviour is guaranteed on the Power architecture because the compiler will insert an `lwsync` barrier before the release write, and that will ensure propagation of the non-atomic write before the later atomic writes on Thread 1.

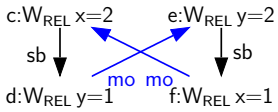
Next consider the execution below where the write on Thread 2 intervenes in modification order between the release and relaxed writes on Thread 1, and the acquire load reads the relaxed store on Thread 1.



Now the release sequence terminates before the store on Thread 2, there is no synchronisation, and there is a race between the non-atomic load and store. The same architectural mechanism that ensured the correct behaviour in the previous execution still applies here, so the behaviour of this program on Power will be the same as the first execution, where there was synchronisation. The C/C++11 memory model is weaker here than it need be: release sequences could be extended to include all program-order-later writes.

There is another way in which release writes are weaker than their implementation on processors. On Power, it is sufficient to place `lwsync` barriers between the writes in the 2+2W test, an addition that corresponds to making the second write on each thread a release:

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { x.store(2,release);
        y.store(1,release);}
  ||| { y.store(2,release);
        x.store(1,release);}
  }}}
  return 0;
}
```



The relaxed behaviour is allowed in C/C++11, and it is necessary to annotate all writes with the `seq_cst` memory order to forbid the behaviour. It is not clear whether programmers rely on the absence of this behaviour, but if they do, strengthening the treatment of release writes would be important for performance.

5.8 Carries-a-dependency-to may be too strong

The memory model extends the dependency-ordered-before relation, that captures synchronisation through consume atomics, through the dependencies captured by the carries-

a-dependency-to relation. cad. The cad relation, described in Chapter 3 and Appendix A, captures syntactic dependency in the program. The definition allows programmers to insert false dependencies into their program as in the following example program:

```
int main() {
  int x = 0;
  atomic_int y = 0;
  {{{ { x = 1;
        y.store(1,release); }
  ||| { r1 = y.load(consume);
        r2 = *(&x ^ r1 ^ r1); }
  }}}
  return 0;
}
```

Although the definition of syntactic dependency in the standard is very clear, compilers can easily recognise this sort of dependency as false, and might optimise it away. Programmers of Linux are not intended to use purely syntactic dependency to enforce ordering, but rather a richer notion of dependency that does not recognise false dependencies. The following section discusses another repercussion of the treatment of dependencies in the standard.

5.9 The use of consume with release fences

The standard does not define synchronisation resulting from the combination of release fences and consume atomics. As a consequence, the formal model does not include such synchronisation. The most economical way of making writes engage in synchronisation is to use release fences. Consume-reads are the cheapest way to synchronise reads. The ability to combine the two would enable the programmer to write synchronised code with less overhead. This seems like an oversight rather than a design decision. A future report to the standardisation committee will suggest wording that provides this synchronisation.

5.10 Thin-air values: an outstanding language design problem

In formalising the C/C++11 memory model, I revealed serious problems with the specification, suggested solutions to many of them, and those have been adopted by the language specification. However, there is one major problem outstanding: thin-air values.

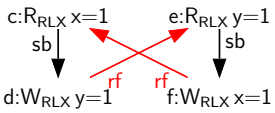
The problem of how to restrict thin-air values in relaxed-memory-model programming languages is a difficult one, and an old one. Much of the complication of the Java memory model was intended to solve a similar problem, but gave rise to a complicated, and

ultimately faulty specification (see Section 5.10.3). In this chapter we explain the problem as precisely as possible, we show that the restriction of thin-air values in C/C++11 invalidates the accepted implementations of atomic accesses, and we explore alternative solutions, making some concrete suggestions.

5.10.1 Thin-air values, informally

Precisely defining thin-air values is the very problem that the language faces, so instead of a definition of the term, this chapter offers a series of examples that explore the design space for the memory model's restriction of thin-air values. Before the first example of an execution with a thin-air value, recall the load-buffering test, repeated below. The model allows the read in each thread to read from the write in the other, completing a cycle in sb union rf. This behaviour is not only allowed by the C/C++11 memory model, it is also permitted for analogous programs on both the ARM and IBM Power architectures, and observable on ARM processors. If load buffering were forbidden by the language, the lightweight implementations of relaxed accesses would have to be strengthened for Power and ARM.

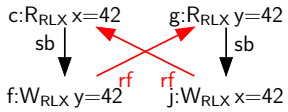
```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { r1 = x.load(relaxed);
        y.store(1,relaxed); }
  ||| { r2 = y.load(relaxed);
        x.store(1,relaxed); }
  }}}
  return 0;
}
```



Allowed by Power and ARM architectures, and observed on ARM, so C/C++11 should allow this.

Many of the examples of thin-air values in this section will, at their core, contain the cyclic shape of this load-buffering execution. The first example of a thin-air value, LB+data+data, augments the basic load-buffering program by adding data-dependencies from the writes to the reads in each thread:

```
int main() {
  atomic_int x = 0;
  atomic_int y = 0;
  int a1,a2;
  {{{ { a1 = x.load(relaxed);
        y.store(a1,relaxed); }
  ||| { a2 = y.load(relaxed);
        x.store(a2,relaxed); }
  }}}
  return 0;
}
```



Forbidden by x86, Power and ARM architectures. C/C++11 should forbid this.

As the formal model stands, any single value may be read by the two atomic reads. The designers of the memory model intend this execution of the program to be forbidden, and the standard imposes a restriction in **29.3p9** that is intended to forbid this sort of execution. In fact, in **29.3p10** the standard says explicitly that **29.3p9** forbids it:

29.3p10

> [...]
>
> ```
> // Thread 1:
> r1 = y.load(memory_order_relaxed);
> x.store(r1, memory_order_relaxed);
>
> // Thread 2:
> r2 = x.load(memory_order_relaxed);
> y.store(r2, memory_order_relaxed);
> ```
>
> [**29.3p9** implies that this] may not produce `r1 = r2 = 42` [...]

This execution of the program violates programmer intuition: the fact that the written value is dependent on the read value implies a causal relationship, as do the reads-from edges. Together these edges form a cycle, and that cycle is what seems to violate intuition and separate this from more benign executions. Analogous programs on x86, Power and ARM processors would all forbid this execution, and the standard explicitly forbids this execution. Moreover, allowing executions of this sort for some types, pointers for example, would allow us to write programs that violate run-time invariants by, say, forging pointers — see Section 5.10.3 for a discussion of Java's safety guarantees in the context of the thin-

air problem. It seems clear then that this sort of thin-air execution should be forbidden.

There is already a notion of dependency that captures that present above: the *data-dependency* relation, dd. One might imagine a restriction that forbade cycles in the union of dd and rf. It would be formulated as a new conjunct of the consistency predicate:

$$\text{let } \mathit{thin\_air1}\ (Xo,\ Xw,\ \_)\ =\quad \text{isIrreflexive (transitiveClosure } (\mathit{dd} \cup \mathit{rf}))$$

The execution above would be forbidden by this new conjunct, but there are other facets of the out-of-thin-air problem that it would not fix. For example, in **29.3p11**, the standard expresses a desire (not a requirement) that control dependencies should, under certain circumstances, forbid similar executions:

29.3p11

> [*Note:* The requirements do allow `r1 == r2 == 42` in the following example, with `x` and `y` initially zero:
>
> ```
> // Thread 1:
> r1 = x.load(memory_order_relaxed);
> if (r1 == 42)
>    y.store(r1, memory_order_relaxed);
>
> // Thread 2:
> r2 = y.load(memory_order_relaxed);
> if (r2 == 42)
>    x.store(42,memory_order_relaxed);
> ```
>
> However, implementations should not allow such behavior. — *end note*]

In the example, we call the execution that ends with `r1 == r2 == 42` a *self-satisfying conditional*: the conditional statements are mutually satisfied. Self-satisfying conditionals are another sort of thin-air value, and they are at the core of the current problems with the language. Allowing self satisfying conditionals harms compositionally (See Chapter 8), but as we shall see, forbidding them is not straightforward.

29.3p11 is a note. Notes have no force, but are typically used for instructive examples, or further exposition of the normative text. In this case, the note includes an instruction to implementers: they "should" not allow such behaviour. The word "should" carries a specific definition in the context of an ISO standard, and means that the sentence

imposes no restriction. Instead, this prose expresses a convention that the memory model designers wish implementers to follow, but that programmers should not rely on. The designers do not want programmers to have to consider such executions, and seem to be abusing the word "should" to avoid providing a complete specification. This unhappy state of affairs is an admission that the designers do not know how to solve the problem of out-of-thin-air-values.

If we apply our prospective thin-air restriction to the example above, we see that there is no cycle in `dd` union `rf`, but if there were a relation that captured syntactic control dependency, `cd`, then the execution that produced `r1 == r2 == 42` would have a cycle in `rf` and `cd`. Of the two examples of thin-air values above, the first involves passing a value through a cycle that includes data-dependencies, and the second involves a cycle including control dependencies. If all of those dependencies were captured in a single relation, `dep`, then both executions would easily be forbidden by disallowing cycles in the union of `dep` and `rf` in a new conjunct of the consistency predicate:

$$\mathsf{let}\ \mathit{thin\_air2}\ (\mathit{Xo},\ \mathit{Xw},\ \_)\ =\quad \text{isIrreflexive (transitiveClosure } (\mathit{dep} \cup \mathit{rf}))$$

At first, this solution seems reasonable. The memory models of typical target hardware architectures do forbid such cycles, so if a C/C++11 program were translated to analogous machine code for a given processor, then no executions with out-of-thin-air-values would be observed.

Unfortunately, forbidding dependency cycles at the language level has serious drawbacks. The users of the C and C++ languages have come to expect high performance, and compiler optimisations make a significant contribution to that end. Consider how forbidding dependency cycles at the language specification level might be implemented: the compiler might preserve all data and control dependencies so that executions resulting from dependency cycles at the language level would translate to executions with dependency cycles on the hardware, where they would be forbidden. Then, any optimisation that removes dependencies would be forbidden, including well used optimisations like hoisting out of a loop, or common subexpression elimination. Understandably, forbidding optimisations is an anathema to the standardisation committee, so preservation of all dependencies in compilation is not a viable proposal.

The thin-air restriction should allow compiler optimisations over concurrent code, but optimisation of relaxed atomic accesses will interact with that restriction. We look at an example of a program with relaxed atomics where the compiler might optimise. The program below has an if-statement where either branch will write forty-two:

```
void main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { r1 = x.load(relaxed); \\ reads 42
        if (r1 == 42)
          y.store(r1,relaxed);
      }
  ||| { r2 = y.load(relaxed); \\ reads 42
        if (r2 == 42)
          x.store(42,relaxed);
        else
          x.store(42,relaxed);
      }
  }}}
}
```

Forbidden by x86, Power and ARM architectures.

Allowed on hardware with thread-local optimisations.

C/C++11 should allow this.

It seems that a compiler should be allowed to optimise this program by replacing the if-statement in Thread 2 with a single write, removing the control dependency, and transforming the program into the one below:

```
void main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { r1 = x.load(relaxed); \\ reads 42
        if (r1 == 42)
          y.store(r1,relaxed);
      }
  ||| { r2 = y.load(relaxed); \\ reads 42
        x.store(42,relaxed);
      }
  }}}
}
```

Allowed by Power and ARM architectures.

C/C++11 should allow this.

The analogue of the optimised execution would be allowed to read forty-two in both

threads according to the Power and ARM architectures, so the language must permit that execution. Therefore, in order to permit the optimisation, the thin-air restriction must allow the relaxed behaviour in the original program. Contrast the un-optimised program above with the example from **29.3p11** in the standard, repeated below:

```
void main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { r1 = x.load(relaxed); \\ reads 42
        if (r1 == 42)
          y.store(r1,relaxed);
      }
  ||| { r2 = y.load(relaxed); \\ reads 42
        if (r2 == 42)
          x.store(42,relaxed);
      }
  }}}
}
```

Forbidden by x86, Power and ARM architectures.

Forbidden on hardware with thread-local optimisations.

Allowing this breaks compositional reasoning.

C/C++11 should forbid this.

The execution where both writes see forty-two is intended to be forbidden, but that execution is identical to the execution of the previous program, where we might optimise. This brings out a subtle point about dependencies: some of them should be respected and preserved, like the ones in **29.3p11**, and others should be ignored in order to permit compiler optimisations. Moreover, we cannot decide which dependencies to respect at the level of a single execution. That is a significant observation, because the rest of the memory model describes the behaviour of single executions. This means the problem cannot be solved by simply adding another conjunct to the consistency predicate; the solution will need to use information from the wider context of the program. In fact, Section 5.10.4 will show that the standard's current thin-air restriction (phrased as another conjunct of the consistency predicate) is broken. First, we describe some possible solutions, and consider Java's approach.

5.10.2 Possible solutions

One could have the optimiser leave dependencies in place only where there might be a dependency cycle. That might appear to be reasonable: dependency cycles can only exist between atomic accesses that are given the weakest memory order parameters, and such accesses within a program are likely to be rare. The problem is that compilation is not a whole program affair. Instead, parts of the program are compiled separately and then linked together. This can lead to optimisations removing dependencies from some functions, only for those functions to be placed in a dependency cycle. Consider the following example where the function `f` is compiled in a separate compilation unit:

```
void f(int a, int* b) {
  if (a == 42)
    *b = 42;
  else
    *b = 42;
}

void main() {
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { r1 = x.load(relaxed); \\ reads 42
        f(r1,&r2);
        y.store(r2,relaxed);
      }
  ||| { r3 = y.load(relaxed); \\ reads 42
        f(r3,&r4);
        x.store(r4,relaxed);
      }
  }}}
}
```

Forbidden by x86, Power and ARM architectures.

C/C++11 should forbid this.

This execution of the program contains a dependency cycle, and would be forbidden by the naive thin-air restriction. At the stage where `f` is compiled, the compiler cannot know whether `f` will be used in a context where its dependency matters, so the compiler would be forbidden from optimising. This suggests that simply forbidding dependency cycles would come at a cost to performance: compilers would be forced to preserve dependencies in the compilation of most functions. This is assumed to be too great a cost, so the

thin-air restriction ought to be weaker than forbidding all syntactic dependency cycles.

Although out-of-thin-air values have not been precisely defined, it is clear that the designers of the memory model intend some executions featuring thin-air values to be forbidden, so some additional restriction is necessary. It is also clear that the weaker that condition is, the fewer optimisations it will forbid. To produce a viable thin-air restriction, one must identify which dependencies must be preserved in an execution. There seem to be two parties who could be given this task, one is the specification designer, and the other is the programmer.

The specification chooses the dependencies Suppose the specification is to select the dependencies that are to be respected in the execution of a program. A natural way to do this would be to introduce a new relation, `dep`, that captures precisely these edges. Then, the consistency predicate can be augmented so that it forbids executions with cycles in `dep` union `rf`. The choice of the `dep` relation could range from the empty set, so that thin-air values are allowed, to all syntactic dependencies, so that dependency removing optimisations are forbidden — the ideal is somewhere in between.

One might imagine a specification that identifies false dependencies between reads and writes by performing a thread-local analysis, in much the same way a compiler does. The thread-local analysis would then be tightly coupled to which compiler optimisations were allowed. If more elaborate whole-program optimisations were to be permitted, then the dependency analysis would have to take place on the whole program instead. One would have to take care not to simply build into the specification a set of compiler optimisations that are allowed. This would both complicate the specification and hamstring the optimiser. Instead, one should try to abstract the effect of reasonable optimisations in such a way that would allow future optimisations.

The programmer chooses the dependencies Another approach would be to have the programmer choose which dependencies are to be respected, and collect those in the `dep` relation. The consistency predicate would again gain a conjunct forbidding cycles in `dep` union `rf`. Compilers would be obliged to leave any dependencies between `dep`-related actions in place. The hardware would then forbid thin-air values that result from `dep`-annotated dependency cycles. These annotations could be attached to functions, so in the previous example, we would annotate `f` with a dependency edge from argument `a` to argument `b`. Then the thin air execution would have a dependency cycle, and could be forbidden in the consistency predicate.

Programs that lack enough annotation would simply permit thin-air behaviour. This might seem reasonable: Java needed to restrict thin-air values to satisfy its security guarantees, and here there are none. Providing well-defined behaviour to programs with thin-air values does come at a cost however: such programs must be compiled faithfully, so this choice acts as a constraint on optimisations and implementation choices. If instead

programs with thin-air values were undefined, the optimiser would be able to rely on their absence.

To stop the specification from allowing thin-air values, a new kind of undefined behaviour could be introduced to require programmers to sufficiently annotate their programs. Thin-air values could be made a cause of undefined behaviour. This brings us back to the difficult problem of defining thin-air values, but in this case it would be a reasonable approximation to define them as cycles in syntactic dependency union rf in consistent executions.

This approach would be easy for the programmer to understand, but there are several potential pitfalls. First, all programs with cycles between relaxed atomics in syntactic dependency union rf would have to be annotated, and optimising such programs would be forbidden. This would forbid reasonable optimisations of atomic accesses, but not non-atomics. Second, this suggestion would be unacceptable if it required annotation of much more code outside of the concurrency library, yet the previous example makes it clear that wider annotation could well be required. It may be that concurrency libraries rarely ever pass values to their clients without synchronising, in which case annotating dependencies in the client would be unnecessary. The suitability of this approach hinges on what sort of concurrent code programmers will write.

The solutions proposed here all have drawbacks, and none has support from the designers of the C/C++11 specifications.

5.10.3 Java encountered similar problems

In many ways the Java memory model attempts to solve the same problems as the C/C++11 memory model: to facilitate efficient implementation on various relaxed memory processors at the same time as permitting compilers to optimise. At its highest level, the Java memory model is similar to the *drfSC* memory model: in the absence of races, Java provides sequentially consistent behaviour. There is a key difference however: the treatment of program faults like races. C/C++11 can simply provide faulty programs with undefined behaviour, whereas Java's security guarantees require that faulty programs receive well-defined behaviour that preserves said guarantees. One of those security guarantees requires that references only be accessible by parts of the program that have explicitly been given the reference — a disparate part of the program should not be able to conjure up a reference out of thin air. Consider the following example, taken from Manson's thesis [69]. In this example, the program has a data race that threatens to produce a reference out of thin air:

```
Initially, x = null, y = null
o is an object with a field f that refers to o
```

```
Thread1     Thread~2
r1 = x;     r3 = y;
r2 = x.f;   x = r4;
y = r2;

r1 == r2 == o is not an acceptable behaviour
```

This program has a race and does not execute in a sequentially consistent model, but rather the relaxed model that Java provides to racy programs. Even so, the memory model must forbid the execution in which the object `o` is read, in order to preserve Java's security guarantees. On the other hand, the relaxed memory model does allow load-buffering behaviour in racy programs; to do otherwise would require stronger implementations, reducing performance on relaxed hardware. The Java memory model is therefore faced with the same thin-air problem as C/C++11: how can a memory model that allows load-buffering behaviour forbid thin-air values while permitting compiler optimisations?

At the core of the Java memory model is an analogue of the C/C++11 consistency predicate, albiet with fewer constraints, permitting all reads to read from happens-before-unrelated writes like atomic reads may in C/C++11. The constraint on the analogue of the reads-from relation is so lax, in fact, that Java permits executions that contain the coherence violation shapes. Valid executions must satisfy this consistency predicate, as well as an intricate out of thin air condition.

Thin-air values are forbidden by allowing only executions where one can incrementally build up a happens-before-down-closed prefix of the execution, so that in the limit, all of the execution's actions are covered. Actions are *committed* to this prefix one-by-one by showing the existence at each step of a similar *justifying execution* that is allowed by a slightly stronger version of the memory model. In this sequence of justifying executions, each execution must share the same actions and relations over the prefix as the execution in the limit, but may differ dramatically otherwise. Entirely different control flow choices might be made by parts of the justifying execution outside of the prefix. The model that judges these justifying executions strengthens the consistency predicate by requiring that reads outside of the prefix read from a write that happens before them.

If Java has a well-formed thin-air restriction, then why should C/C++11 not simply adopt it? There are two reasons: firstly, there are optimisations that Java's solution does not permit that should be allowed. Oracle's own Hotspot compiler violates the Java memory model by performing disallowed compiler optimisations [101]. Secondly, the model is difficult to reason about: in order to understand how a program might behave, one must imagine a sequence of justifying executions, each potentially very different and diverging further and further from an original, more constrained execution. If one wants to check whether a particular execution is allowed or not, it is a matter of judging whether such a sequence exists.

Java introduces a great deal of complication attempting to solve the thin-air problem, but it is ultimately unsuccessful. The C/C++11 memory model ought to be a simpler case — there are no security guarantees to satisfy, and conventions can be imposed on the programmer, with undefined behaviour if they are ignored. The next section shows that the restriction that is included in C/C++11 does restricts some executions that must be allowed.

5.10.4 The standard's thin-air restriction is broken

The thin air restriction imposed by the standard is given in **29.3p9**:

29.3p9

> An atomic store shall only store a value that has been computed from constants and program input values by a finite sequence of program evaluations, such that each evaluation observes the values of variables as computed by the last prior assignment in the sequence. The ordering of evaluations in this sequence shall be such that:
>
> - if an evaluation B observes a value computed by A in a different thread, then B does not happen before A, and
> - if an evaluation A is included in the sequence, then every evaluation that assigns to the same variable and happens before A is included.

This text was added very early in the development of the standard, and survives unchanged since its introduction to drafts in 2007 [34]. The standard provides examples that illustrate the application of the restriction in **29.3p10**:

29.3p10

> [*Note:* The second requirement disallows "out-of-thin-air" or "speculative" stores of atomics when relaxed atomics are used. Since unordered operations are involved, evaluations may appear in this sequence out of thread order. For example, with x and y initially zero,

```
 // Thread 1:
r1 = y.load(memory_order_relaxed)
x.store(r1, memory_order_relaxed);

// Thread 2:
r2 = x.load(memory_order_relaxed);
y.store(42, memory_order_relaxed);
```

is allowed to produce `r1 = r2 = 42`. The sequence of evaluations justifying this consists of:

```
 y.store(42, memory_order_relaxed);
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);
r2 = x.load(memory_order_relaxed);
```

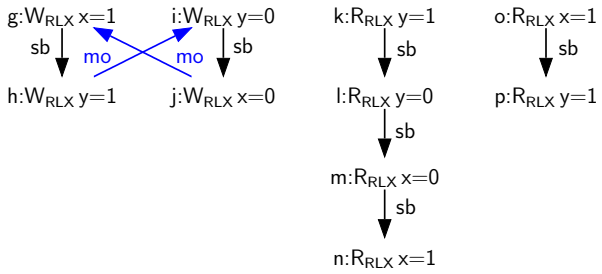
On the other hand,

```
 // Thread 1:
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);

// Thread 2:
r2 = x.load(memory_order_relaxed);
y.store(r2, memory_order_relaxed);
```

may not produce `r1 = r2 = 42`, since there is no sequence of evaluations that results in the computation of `42`. In the absence of "relaxed" operations and read-modify-write operations with weaker than `memory_order_acq_rel` ordering, the second requirement has no impact. — *end note*]

There are several problems with this restriction. First of all, by the admission of the standard itself in **29.3p11**, it is weaker than intended. Secondly, this requirement is incorrectly phrased in terms of evaluations rather than memory accesses. Perhaps more fatally, this requirement imposes a restriction that would not be satisfied by the industry-standard implementation of the atomic reads and writes on Power processors: a stronger implementation would be needed, decreasing performance. To see that this is the case, consider the following program and execution:

```
int main() {
  atomic_int x = 2;
  atomic_int y = 2;
  atomic_int z = 0;
  atomic_int* A[3] = {&x,&y,&z};
  int r1,r2,r3,r4,r5,r6,sum;
  {{{ { x.store(1,relaxed);   // write y index
        y.store(1,relaxed); } // write y index
  ||| { y.store(0,relaxed);   // write x index
        x.store(0,relaxed); } // write x index
  ||| { r1 = y.load(relaxed);         // read y index
        r2 = (A[r1])->load(relaxed);   // read x index
        r3 = (A[r2])->load(relaxed);   // read x index
        r4 = (A[r3])->load(relaxed); } // read y index
  ||| { r5 = x.load(relaxed);         // read y index
        r6 = (A[r5])->load(relaxed); }   // read y index
  }}};
  sum = 100000*r1 + 10000*r2 + 1000*r3 + 100*r4 + 10*r5 + r6
  z.store(sum,relaxed);
  return 0;
}
```



This program gives rise to an execution where Thread 1 and 2 produce 2+2W behaviour, a cycle in sb union mo. Two other threads are reading from this cycle: Thread 3 reads the writes in the order of the cycle, from Thread 1's write of `y` to its write of `x`, and Thread 4 reads Thread 1's write of `x` and then its write of `y`. Threads 3 and 4 both have address dependencies between each pair of reads in the sequence. After the threads have completed, there is a write of `z` whose value is calculated from all of the previous reads.

Now we shall try to find a sequence of evaluations that satisfies **29.3p9** for the store to location `z`. The value stored to `z` is computed from all of the values read in Threads 3 and 4, so the reads in those threads must be in the sequence, as must the writes that they read from in Threads 1 and 2. Note that the dependencies between the reads in Threads 3 and 4 mean that the ordering in the threads must be preserved in the sequence. Each read must read from the most recent write in the sequence, so we can start to work out

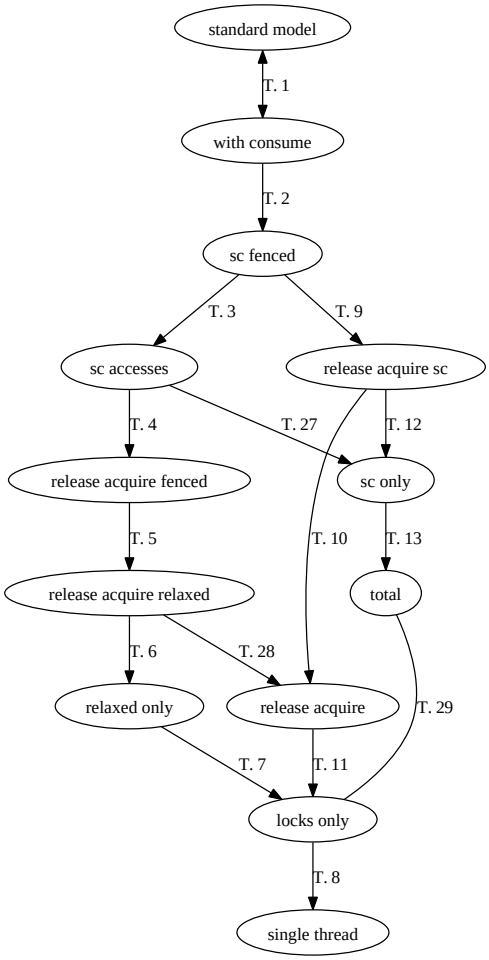
some constraints. Thread 3 requires the writes to appear in the following order in the sequence: `y=1`, `y=0`, `x=0`, `x=1`. This contradicts the values that are read on Thread 4, which reads `x=1`, and then reads `y=1`, so this execution violates **29.3p9**, and should be forbidden.

Using the industry standard mapping of C/C++11 atomic accesses to Power accesses, this execution of the program is allowed, so in order to forbid it, a stronger choice of implementation would be necessary. This is a clear indication that **29.3p9** is too strong a restriction, that it is not what the designers intended, and that the treatment of thin-air values in the standard is broken.

Chapter 6

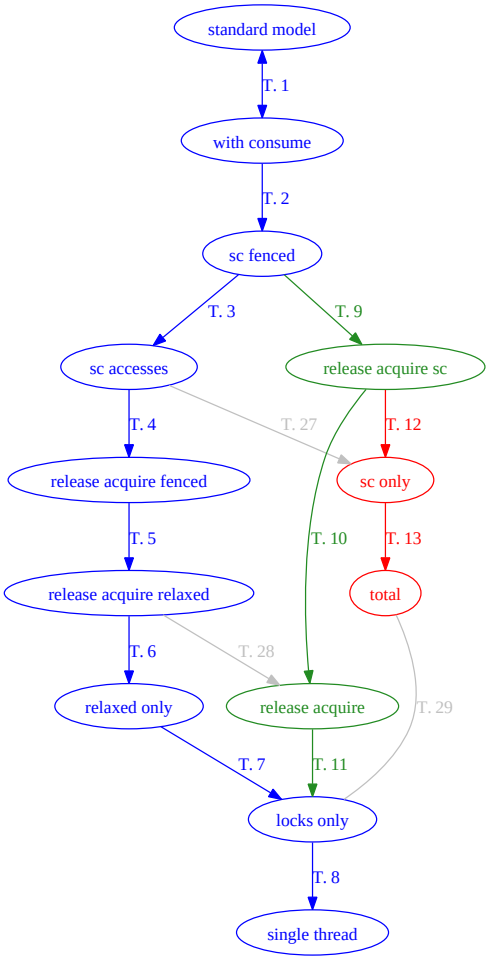
Model meta-theory

This chapter relates the models of Chapter 3 to one another, setting out the conditions under which one can eschew the complexity of a richer model in favour of a simpler one. These relationships take the form of theorems over the Lem definitions of the memory model (see Appendix C), all shown to hold in HOL4 (the proof scripts can be found at the thesis web page [2]), that state the condition that a program must satisfy in order to have equivalence between two models. In most cases of equivalence, this condition is simply the stronger model condition, but in one case there is no need for a model condition, and in another, neither model condition subsumes the other. The following directed graph represents the model equivalences, with directed edges corresponding to the subsumption of model conditions. The edge that points in both directions corresponds to a direct equality of the model conditions, and the undirected edge requires a condition stronger than either model condition.



The equivalences provide several useful results. Theorem 1 shows that part of the specification is redundant, as it is subsumed by the rest of the model. The rest of the equivalences show that one can consider a simpler model if the program does not use all of the features of the model. Theorems 1, 2, 9, 12 and 13 establish one of the design goals of the language, for a restricted set of programs, without loops or recursion. This is the design goal described by Boehm and Adve [37]: data-race-free code that uses only regular reads and writes, locks, and the sequentially-consistent atomic accessor functions will execute with sequentially consistent semantics.

The chapter begins with a precise formulation of what it means for two models to be equivalent, and then, using the graph above to organise the structure, presents the equivalence results and key insights from the proofs. These results are presented in three paths through the graph, highlighted in blue red and green below:



The blue path starts with a proof that visible sequences of side effects are redundant, in Section 6.2 and then contains a sequence of increasingly simple models that apply to smaller and smaller sublanguages, in Section 6.3. The proofs of the equivalences that the blue path represents are all relatively straightforward: reducing the set of features covered by each sublanguage makes the machinery that governed the unused features redundant.

The green path considers sublanguages without relaxed atomics. This simplifies the memory model significantly: the model no longer needs to keep track of release sequences, and thin-air behaviour is not allowed. The proofs of equivalence in this section are more involved: we must show that any happens-before edge that is induced by reading from a release sequence is covered by other happens-before edges in the execution.

The red path includes the analogue of the result described by Boehm and Adve [37], established for programs without loops or recursion over the full C/C++11 memory model: data-race-free programs that use only the SC atomics, locks and non-atomic accesses have SC behaviour. The proof of this relationship is the most involved in this chapter.

6.1 Defining the equivalence of models

The behaviour of a program according to one of the memory models given in Chapter 3 is either a set of candidate executions or undefined behaviour. Intuitively, two models are equivalent if, for any program, they either both produce the same set of candidate execu-

tions, or they both produce undefined behaviour. The type of candidate executions may differ between two models: they may use different calculated relations in their consistency judgement, or even different relations in their execution witness. There are, however, elements of the type of candidate executions that remain the same; both the pre-execution, that describes the actions of one path of control flow, and the reads-from relation are common across all models. Together these elements of the candidate execution represent a reasonable notion of the observable interaction with memory. A candidate execution X is a triple comprised of a pre-execution, an execution witness, and a list of calculated relations. The function below projects the pre-execution and the reads-from relation:

```
let rf_observable_filter X = {(Xo, Xw.rf) | ∃ rl. (Xo, Xw, rl) ∈ X}
```

As it happens, most of the models share the same execution witness, and can be related by a stronger notion of equivalence. For these models the filter that projects the entire execution witness is used:

```
let observable_filter X = {(Xo, Xw) | ∃ rl. (Xo, Xw, rl) ∈ X}
```

These projections are used to construct a function that generates the observable behaviour of a given program when run under a specific model. Each function produces either a set of projected executions, or a constant denoting undefined behaviour. The first produces the behaviour for the reads-from projection, with an arbitrary memory model taken as a parameter:

```
let rf_behaviour M condition opsem (p : PROGRAM) =
  let consistent_executions =
    { (Xo, Xw, rl) |
        opsem p Xo ∧
        apply_tree M.consistent (Xo, Xw, rl) ∧
        rl = M.relation_calculation Xo Xw } in
  if condition consistent_executions ∧
    ∀ X ∈ consistent_executions.
      each_empty M.undefined X
  then rf_Defined (rf_observable_filter consistent_executions)
  else rf_Undefined
```

The second produces the behaviour of the more precise projection that retains the whole execution witness:

```
let behaviour M condition opsem (p : PROGRAM) =
  let consistent_executions =
    { (Xo, Xw, rl) |
```

opsem p Xo ∧
apply_tree *M.consistent* (*Xo*, *Xw*, *rl*) ∧
rl = *M.relation_calculation Xo Xw* } in
if *condition consistent_executions* ∧
∀ *X* ∈ *consistent_executions*.
each_empty *M.undefined X*
then Defined (observable_filter *consistent_executions*)
else Undefined

Each of the equivalence results share a similar form: under some condition *cond* over the operational semantics and the program, the models M_1 and M_2, with model conditions P_1 and P_2 are equivalent. Here, equivalence is expressed as the equality of behaviour for any thread-local semantics and program:

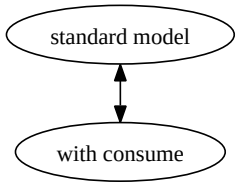
(∀ *opsem p*.
cond *opsem p* ⟶
(rf_behaviour M_1 P_1 *opsem p* = rf_behaviour M_2 P_2 *opsem p*))

6.2 Visible sequences of side effects are redundant in the standard

The first equivalence is between the *standard_memory_model* (§3.10), and the *with_consume_memory_model* (§3.9). This equivalence simplifies the memory model presented in the standard by showing that a complicated part of the specification is redundant and can be omitted to no ill-effect. Throughout this chapter, each result from the overview graph will be presented with the edge from the overview that represents the equivalence, the detailed formulation of the necessary condition for the equivalence, and the equivalence result itself. In this case, the equivalence applies to all programs and choices of operational semantics, so there is no necessary condition, and the more precise equivalence that uses the full execution-witness projection is used:

Theorem 1.

(∀ *opsem p*.
(*behaviour with_consume_memory_model true_condition opsem p* =
behaviour standard_memory_model true_condition opsem p))



When proving this equivalence, it is useful to note that the two memory models differ very little: the undefined behaviour, calculated relations and consistency predicates of

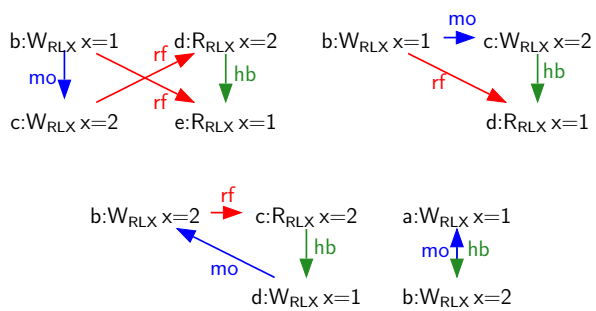
the two models are almost identical. The only difference is the inclusion in the *standard_memory_model* model of the calculated relation, *standard_vsses*, identifying the *visible sequences of side effects* of each atomic load, and an accompanying requirement that atomic loads read from a predecessor in this relation. This relation was introduced in Section 3.10, and is justified with text from the standard in Appendix A. It is intended to represent the set of writes that a particular read may read from. Chapter 5 argues that the language used to define it in the standard is ambiguous, and it does not achieve its goal of identifying the precise set of writes that may be read from. Recall the definition of visible sequences of side effects:

$$
\begin{aligned}
&\textsf{let } \mathit{standard\_vsses}\ \mathit{actions}\ \mathit{lk}\ \mathit{mo}\ \mathit{hb}\ \mathit{vse}\ = \\
&\quad \{\ (v,\ r)\ |\ \forall\ r \in \mathit{actions}\ v \in \mathit{actions}\ \mathit{head} \in \mathit{actions}\ | \\
&\qquad \text{is\_at\_atomic\_location}\ \mathit{lk}\ r \wedge (\mathit{head},\ r) \in \mathit{vse} \wedge \\
&\qquad \neg\ (\exists\ v' \in \mathit{actions}.\ (v',\ r) \in \mathit{vse} \wedge (\mathit{head},\ v') \in \mathit{mo}) \wedge \\
&\qquad (\ v = \mathit{head} \vee \\
&\qquad\quad (\ (\mathit{head},\ v) \in \mathit{mo} \wedge \neg\ ((r,\ v) \in \mathit{hb}) \wedge \\
&\qquad\qquad \forall\ w \in \mathit{actions}. \\
&\qquad\qquad\quad ((\mathit{head},\ w) \in \mathit{mo} \wedge (w,\ v) \in \mathit{mo}) \longrightarrow \neg\ ((r,\ w) \in \mathit{hb}) \\
&\qquad\quad) \\
&\qquad) \\
&\quad \}
\end{aligned}
$$

This relation creates edges to each read at an atomic location from each visible side effect, and from all modification order successors of those visible side effects up to, but not including (recall that hb is acyclic) any writes that happen after the read. The consistency predicate then requires that atomic loads read from one of the writes that this relation identifies:

$$
\begin{aligned}
&\textsf{let } \mathit{standard\_consistent\_atomic\_rf}\ (\mathit{Xo},\ \mathit{Xw},\ \_ ::\ \_ ::\ \_ ::\ (\text{“}\mathit{vsses}\text{”},\ \mathit{vsses}) ::\ \_)\ = \\
&\quad \forall\ (w,\ r) \in \mathit{Xw.rf}.\ \text{is\_at\_atomic\_location}\ \mathit{Xo.lk}\ r \wedge \text{is\_load}\ r \longrightarrow \\
&\qquad (w,\ r) \in \mathit{vsses}
\end{aligned}
$$

This conjunct of the consistency predicate is made redundant by the coherence axioms, which forbid executions that contain any of the shapes below:



The *with_consume_memory_model* omits the requirement that atomic loads read from a write related by *standard_vsses*, and instead requires only that atomic loads do not read a later write in happens-before.

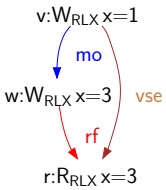
We prove that for any execution, the consistency predicates have the same outcome. Because the calculation of undefined behaviour has not changed, this property implies the equality of the behaviour of the two models. We split the equality of outcome into two implications.

Standard-model consistency implies simplified-model consistency It is clear that any consistent execution in the *standard_memory_model* is a consistent execution in the *with_consume_memory_model*, because the new restriction on atomic loads is already enforced (*standard_vsses* does not relate writes to loads that happen after them).

Simplified-model consistency implies standard-model consistency It remains to show that any consistent execution in the *with_consume_memory_model* is a consistent execution in the *standard_memory_model*.

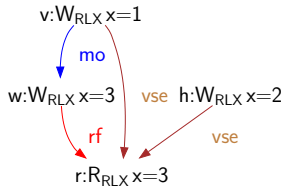
Suppose, seeking a contradiction, that there is a consistent execution in the *with_consume_memory_model* that is not a consistent execution in the *standard_memory_model*. That execution must fail the consistency predicate in the *standard_memory_model* by violating the *standard_consistent_atomic_rf* conjunct. Then there is an edge in rf relating an action, w, to a load at an atomic location, r, but the rf edge is not coincident with a *standard_vsses* edge. The rest of the consistency predicate holds, so the rf edge implies that w is a write at the same location as r.

Now, we identify an action v that is related to r by *standard_vsses*, forming a contradiction. The existence of the rf edge from w to r implies the existence of a visible side effect, v of r by the *det_read* conjunct of the consistency predicate. If v and w are not equal, then (v, w) must be in mo, otherwise v would happen before r, by the definition of visible side effect, and (w, v) would be in mo, by consistency of of mo, and there would be a CoWR coherence violation. So far, we have identified the following actions and edges:



Now seek the mo-maximal element of a set, A, containing the visible side effects of r that are either equal to w or appear earlier in mo. Certainly v is a member of A so the set is non-empty. In addition, A is a subset of the prefix of w union the singleton set w. Consistency implies both that modification order is acyclic, and that it has finite prefixes, so we know that A is finite. Then the non-empty finite set A has a maximal element,

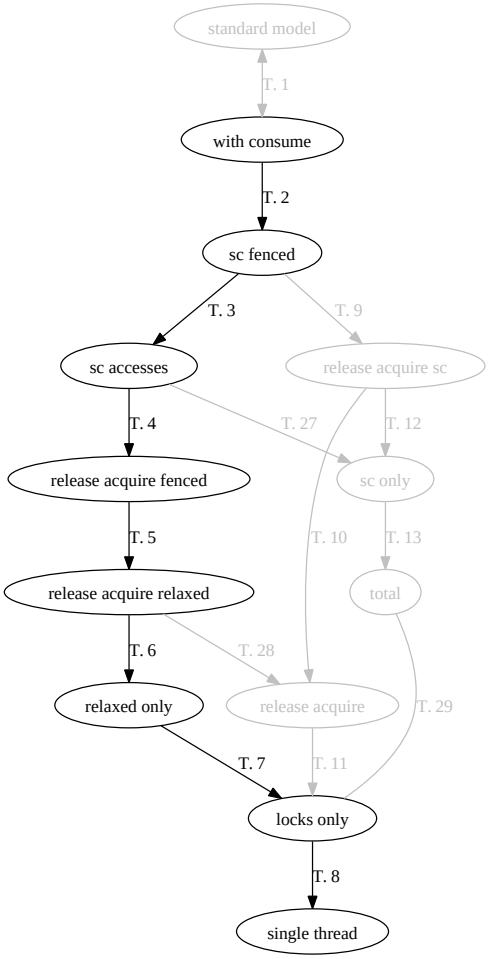
h, in the acyclic relation mo. The visible side effect h is then either equal to w, or is an mo-maximal mo-predecessor of w.



Returning to the definition of *standard_vsses*, we show that w is related to r with the head action h. Note that h has no mo-later visible side-effects, otherwise there would be a CoWR coherence violation between that side-effect, w and r. We have that (r, w) is not in happens-before from the consistency predicate of the *with_consume_memory_model*. For any mo-intervening action, w', between h and w, if there is a happens-before edge from r to w' then that edge completes a CoRW cycle between r, w' and w. All together, this implies that h is related to r by *standard_vsses*, a contradiction, proving the equivalence holds, and showing that visible sequences of side-effects are a needless complication in an already intricate memory model.

6.3 Avoid advanced features for a simpler model

The equivalences presented in this section are collected together because their necessary conditions are simple: they restrict the use of concurrency features, and make no other restrictions on programs. The proofs of these equivalences are correspondingly straightforward. The graph below shows the sequence of equivalences covered in this section in the context of the rest of the results:



The sequence of equivalences will start from the simplified *with_consume_memory_model* that imposes no restrictions on programs, and remove features one step at a time, eventually reaching the *single_thread_memory_model*.

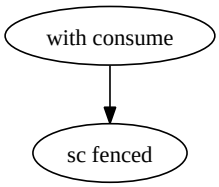
Programs without consume atomics This equivalence relates the simplified model from the previous section, *with_consume_memory_model*, presented formally in Section 3.9, to a model for programs that do not use the consume memory order, the *sc_fenced_memory_model* from Section 3.8. The model condition requires that programs do not use the consume memory order:

```
let sc_fenced_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    ∀ a ∈ Xo.actions.
      match a with
      | Lock _ _ _ _ → true
      | Unlock _ _ _ → true
      | Load _ _ mo _ _ → (mo ∈ {NA, Acquire, Relaxed, Seq_cst})
      | Store _ _ mo _ _ → (mo ∈ {NA, Release, Relaxed, Seq_cst})
```

```
  | RMW _ _ mo _ _ _ → (mo ∈ {Acq_rel, Acquire, Release, Relaxed, Seq_cst})
  | Fence _ _ mo → (mo ∈ {Release, Acquire, Relaxed, Seq_cst})
  | Blocked_rmw _ _ _ → true
  end
```

THEOREM 2.

```
(∀ opsem p.
  statically_satisfied sc_fenced_condition opsem p ⟶
  (behaviour sc_fenced_memory_model sc_fenced_condition opsem p =
   behaviour with_consume_memory_model true_condition opsem p))
```



The consume memory order provides language-level synchronisation without requiring the insertion of a hardware barrier on the Power and ARM architectures. It was introduced in Section 3.9, adding significant complexity to the C/C++11 memory model. Without it, happens-before would be far simpler to formulate, and would be transitive — an important property for a relation that evokes a temporal intuition. Happens-before is defined with auxiliary definitions in the *with_consume_memory_model*, with additional calculated relations *cad* and *dob*.

The purpose of this complexity is to define happens-before edges resulting from consume reads as transitive only in the presence of thread-local dependency, *dd*. In the *with_consume_memory_model* model, happens-before is defined as follows:

```
let inter_thread_happens_before actions sb sw dob =
  let r = sw ∪ dob ∪ (compose sw sb) in
    transitiveClosure (r ∪ (compose sb r))


let happens_before actions sb ithb =
  sb ∪ ithb
```

Without any consume actions, the calculated relation *dob* is empty, and the relatively complicated definitions of *inter_thread_happens_before* and *happens_before* above can be simplified to the following:

```
let no_consume_hb sb sw =
  transitiveClosure (sb ∪ sw)
```

We will show that this simplified version of happens-before is equivalent to the more complicated version above. It is straightforward that the simplified version is a superset of the more complex definition, so for equivalence, it remains to show that any edge in *no_consume_hb* is also present in *happens_before*. The HOL4 proof proceeds by considering

each edge in *no_consume_hb* as a path of edges that alternate between sb and sw edges. Any path made up entirely of sb edges in a consistent execution is itself an sb edge, because sb is transitive, and is part of *happens_before*. Any path made up of sw edges is a path of edges in the relation r from the definition of *inter_thread_happens_before*, and is also in *happens_before*. Similarly, any path that alternates between *sb* and *sw*, starting with an *sw* edge, is in r and *happens_before*. Paths that start with an *sb* edge are included by the union of the composition of *sb* and *sw* edges from *inter_thread_happens_before*. This covers all possible edges in *no_consume_hb*, so the relations are equal.

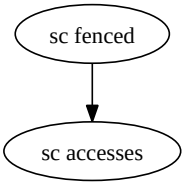
The *sc_fenced_memory_model* shares the same undefined behaviour and consistency predicate as the *with_consume_memory_model*, but its calculated relations are simpler, omitting *cad* and *dob* and taking the simpler version of happens-before presented above.

Programs without SC fences The next equivalence, between the *sc_fenced_memory_model* from Section 3.8 and the *sc_accesses_memory_model* from Section 3.7, applies to programs that do not use fences with the `seq_cst` memory order:

```
let sc_accesses_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    ∀ a ∈ Xo.actions.
      match a with
      | Lock _ _ _ _ → true
      | Unlock _ _ _ → true
      | Load _ _ mo _ _ → (mo ∈ {NA, Acquire, Relaxed, Seq_cst})
      | Store _ _ mo _ _ → (mo ∈ {NA, Release, Relaxed, Seq_cst})
      | RMW _ _ mo _ _ _ → (mo ∈ {Acq_rel, Acquire, Release, Relaxed, Seq_cst})
      | Fence _ _ mo → (mo ∈ {Release, Acquire, Relaxed})
      | Blocked_rmw _ _ _ → true
      end
```

THEOREM 3.

```
(∀ opsem p.
  statically_satisfied sc_accesses_condition opsem p ⟶
  (behaviour sc_accesses_memory_model sc_accesses_condition opsem p =
   behaviour sc_fenced_memory_model sc_fenced_condition opsem p))
```



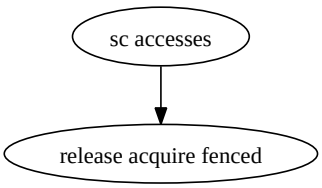
The *sc_accesses_memory_model* expresses the semantics of such programs, simplifying the *sc_fenced_memory_model* by removing the *sc_fenced_sc_fences_heeded* conjunct from the consistency predicate. The *sc_fenced_sc_fences_heeded* conjunct imposes no restriction on programs that do not feature SC fences, and the rest of the model is the same, so proving equivalence of the two models is trivial.

Programs without SC atomics Removing all SC atomics permits further simplification of the memory model. The *release_acquire_fenced_memory_model*, from Section 3.6, is equivalent to the *sc_accesses_memory_model*, from Section 3.7, in the absence of SC atomics:

```
let release_acquire_fenced_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    ∀ a ∈ Xo.actions.
      match a with
      | Lock _ _ _ _ → true
      | Unlock _ _ _ → true
      | Load _ _ mo _ _ → (mo ∈ {NA, Acquire, Relaxed})
      | Store _ _ mo _ _ → (mo ∈ {NA, Release, Relaxed})
      | RMW _ _ mo _ _ _ → (mo ∈ {Acq_rel, Acquire, Release, Relaxed})
      | Fence _ _ mo → (mo ∈ {Release, Acquire, Relaxed})
      | Blocked_rmw _ _ _ → true
      end
```

THEOREM 4.

```
(∀ opsem p.
  statically_satisfied release_acquire_fenced_condition opsem p ⟶
 (behaviour release_acquire_fenced_memory_model release_acquire_fenced_condition opsem p =
  behaviour sc_accesses_memory_model sc_accesses_condition opsem p))
```



Without SC atomics, the sc order is no longer needed, and restrictions based on it can be omitted. The *release_acquire_fenced_memory_model* drops two conjuncts from the consistency predicate: *sc_accesses_consistent_sc* and *sc_accesses_sc_reads_restricted*. These predicates impose no restriction in the absence of SC atomics, and the rest of the model remains the same, so showing equivalence with the *sc_accesses_memory_model* is trivial.

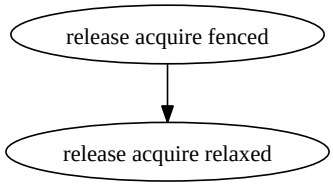
Programs without fences Without fences, the calculation of sw can be simplified substantially. The *release_acquire_relaxed_memory_model*, from Section 3.5, is equivalent to the *release_acquire_fenced_memory_model*, from Section 3.6, for programs without fences:

```
let release_acquire_relaxed_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
```

```
∀ a ∈ Xo.actions.
  match a with
  | Lock _ _ _ _ → true
  | Unlock _ _ _ → true
  | Load _ _ mo _ _ → (mo ∈ {NA, Acquire, Relaxed})
  | Store _ _ mo _ _ → (mo ∈ {NA, Release, Relaxed})
  | RMW _ _ mo _ _ _ → (mo ∈ {Acq_rel, Acquire, Release, Relaxed})
  | Fence _ _ _ → false
  | Blocked_rmw _ _ _ → true
  end
```

THEOREM 5.

```
(∀ opsem p.
  statically_satisfied release_acquire_relaxed_condition opsem p ⟶
 (behaviour release_acquire_relaxed_memory_model release_acquire_relaxed_condition opsem p =
  behaviour release_acquire_fenced_memory_model release_acquire_fenced_condition opsem p))
```



The mathematical machinery that supports fence synchronisation is relatively complex, and omitting fences makes it unnecessary. The *release_acquire_relaxed_memory_model* has the same consistency predicate and undefined behaviour as the *release_acquire_fenced_memory_model*, but the calculated relations that make up the sw relation can be simplified. First, the hypothetical-release-sequence relation, *hrs*, can be omitted: it is only used for fence synchronisation. Then the definition of sw can be simplified to remove conjuncts that apply only to fences. Again, proving this equivalence is trivial, because the differences in the models have no effect for programs without fences.

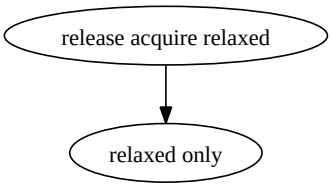
Programs without release or acquire atomics Without the release and acquire memory orders, programs can no longer use atomic accesses to synchronise, and again the sw relation can be simplified. The *relaxed_only_memory_model* of Section 3.3 is equivalent to the *release_acquire_relaxed_memory_model* of Section 3.5 for programs that do not use the release and acquire memory orders:

```
let relaxed_only_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    ∀ a ∈ Xo.actions.
```

```
match a with
| Lock _ _ _ _ → true
| Unlock _ _ _ → true
| Load _ _ mo _ _ → mo ∈ {NA, Relaxed}
| Store _ _ mo _ _ → mo ∈ {NA, Relaxed}
| RMW _ _ mo _ _ _ → mo ∈ {Relaxed}
| Fence _ _ _ → false
| Blocked_rmw _ _ _ → true
end
```

THEOREM 6.

$(\forall\ opsem\ p.$
$\quad statically\_satisfied\ relaxed\_only\_condition\ opsem\ p \longrightarrow$
$\quad (behaviour\ relaxed\_only\_memory\_model\ relaxed\_only\_condition\ opsem\ p =$
$\quad behaviour\ release\_acquire\_relaxed\_memory\_model\ release\_acquire\_relaxed\_condition\ opsem\ p))$



Without release atomics, there can be no release-sequences, so the calculated relation rs will be empty, and can be removed. Furthermore, the atomic synchronisation conjunct can be omitted from the sw relation, which becomes:

```
let locks_only_sw actions asw lo a b =
  (tid_of a ≠ tid_of b) ∧
  ( (* thread sync *)
    (a, b) ∈ asw ∨
    (* mutex sync *)
    (is_unlock a ∧ is_lock b ∧ (a, b) ∈ lo)
  )
```

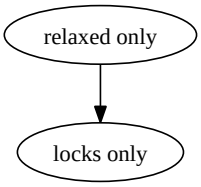
The consistency predicate and calculation of undefined behaviour remain the same, and it is just the set of calculated relations that change. The relations that have changed have only omitted constituent parts that were empty in suitably restricted programs, so equivalence is straightforward.

Programs without atomics For programs that do not use atomic locations, the *relaxed_only_memory_model* of Section 3.3 is equivalent to the *locks_only_memory_model* of Section 3.2:

```
let locks_only_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    ∀ a ∈ Xo.actions.
      match (loc_of a) with
      | Nothing → false
      | Just l → (Xo.lk l ∈ {Mutex, Non_Atomic})
      end
```

THEOREM 7.

```
(∀ opsem p.
  statically_satisfied locks_only_condition opsem p ⟶
  (behaviour locks_only_memory_model locks_only_condition opsem p =
   behaviour relaxed_only_memory_model relaxed_only_condition opsem p))
```



The *locks_only_memory_model* describes the semantics of such programs, and is greatly simplified. Without atomics, the modification-order relation, mo, is empty, and the restrictions that apply to atomics or mo edges become unnecessary. The following conjuncts of the consistency predicate are omitted:

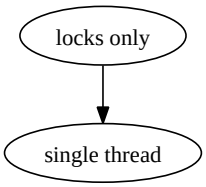
- *consistent_mo*,
- *consistent_atomic_rf*,
- *coherence_memory_use* and
- *rmw_atomicity*.

Again, equivalence is easy to show, because the parts of the model that have been lost impose no restriction over the restricted set of programs.

single-threaded programs The final step in the sequence equivalences applies to single-threaded programs that use none of the concurrency features. For these programs, the *single_thread_memory_model* of Section 3.1 is equivalent to the *locks_only_memory_model* of Section 3.2:

```
let single_thread_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
  ∃ b ∈ Xo.actions. ∀ a ∈ Xo.actions.
    (tid_of a = tid_of b) ∧
    match (loc_of a) with
    | Nothing → false
    | Just l → (Xo.lk l = Non_Atomic)
    end
```

THEOREM 8.
$(\forall\ opsem\ p.$
$\quad statically\_satisfied\ single\_thread\_condition\ opsem\ p \longrightarrow$
$\quad (behaviour\ single\_thread\_memory\_model\ single\_thread\_condition\ opsem\ p =$
$\quad\ behaviour\ locks\_only\_memory\_model\ locks\_only\_condition\ opsem\ p))$

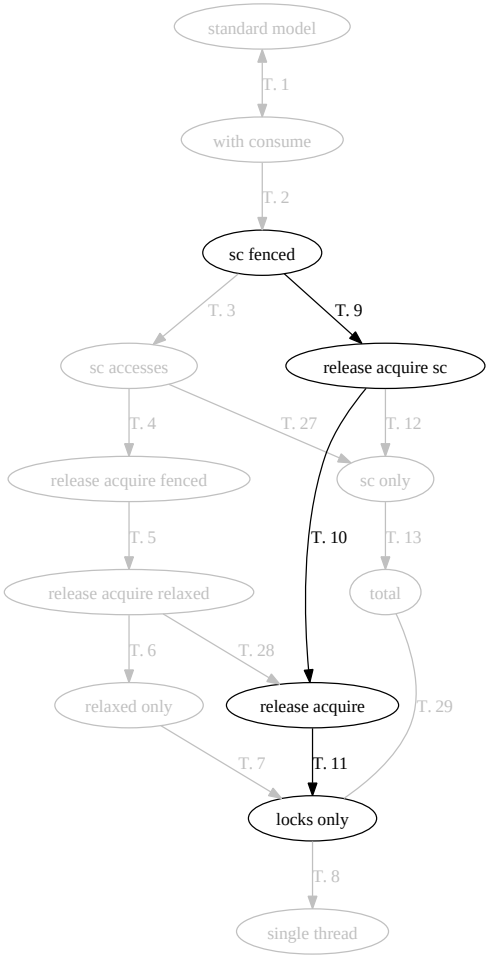


The *single_thread_memory_model* includes several simplifications. The calculation of undefined behaviour no longer needs to identify data races because they are the interaction of memory access from more than one thread. There is no need to check for incorrectly used mutexes because the memory model only applies to programs that do not use mutex locations. There is no synchronisation at all in this model because there is only a single thread, so the sw relation is no longer needed, and happens-before is equal to the sequenced-before relation. Sequenced before is already required to be acyclic, so there is no need to check that again with the *consistent_hb* conjunct of the consistency predicate. Lock order, lo, no longer has any effect on consistency, so the *locks_only_consistent_lo* and *locks_only_consistent_locks* conjuncts of the consistency predicate can be omitted.

The proof of equivalence with all of these omissions is again straightforward, because the parts that are left out place no restrictions on programs that satisfy the condition.

6.4 Synchronisation without release sequences

This section draws together a set of equivalences that simplify the release-acquire synchronisation mechanism for programs that do not use relaxed atomic accesses. Without relaxed, there is no need for the complexity of the release sequence, and if the initialisation of an atomic always happens before all accesses of it, thin-air executions (see Chapter 5) are forbidden. These results make up the following path through the overview graph:



As well as avoiding relaxed atomic accesses, the models presented in this section require some modest discipline in the use of atomic initialisation: *atomic_initialisation_first* requires that the program perform only a single initialisation at each atomic location, and that it create happens-before edges from the initialisation to any writes at that location. The precise definition of the restriction, and motivation for its existence are provided below. The C++11 imposes a similar restriction on programmers: it allows them to initialise an atomic only once. C11 appears to be more liberal however, allowing programmers to re-initialise atomics.

The first equivalence relates the *sc_fenced_memory_model* of Section 3.8 to a new memory model: the *release_acquire_SC_memory_model*, that does not have release sequences. Recall that the release sequence allows reads from relaxed atomics to cause synchronisation to mo-prior release atomics. Release sequences are defined as follows:

```
let rs_element head a =
    (tid_of a = tid_of head) ∨ is_RMW a


let release_sequence_set actions lk mo =
  { (rel, b) | ∀ rel ∈ actions b ∈ actions |
```

is_release rel $\wedge$
($(b = rel) \vee$
  ($(rel,\ b) \in mo \wedge$
    rs_element rel b $\wedge$
    $\forall\ c \in actions.$
      $((rel,\ c) \in mo \wedge (c,\ b) \in mo) \longrightarrow$ rs_element rel c)) }

Actions in the sequence are either on the same thread as the release, or they are read-modify-write actions. The sequence is headed by a release action, and is made up of a contiguous subset of mo. The sequence ends before the first action on a different thread that is not a read-modify-write.

In the *sc_fenced_memory_model*, the sw relation is calculated using release sequences — acquire actions that read from some write, synchronise with the head of any release sequence that contains the write (a version of sw without fence synchronisation is shown for clarity):

```
let release_acquire_relaxed_synchronizes_with actions sb asw rf lo rs a b =
  (tid_of a ≠ tid_of b) ∧
  ( (* thread sync *)
    (a, b) ∈ asw ∨
    (* mutex sync *)
    (is_unlock a ∧ is_lock b ∧ (a, b) ∈ lo) ∨
    (* rel/acq sync *)
    ( is_release a ∧ is_acquire b ∧
      (∃ c ∈ actions. (a, c) ∈ rs ∧ (c, b) ∈ rf) )
  )...
```

It is a tempting hypothesis that, in the absence of relaxed-memory-order operations, release sequences have no effect, but this is not true without also imposing an additional restriction. Non-atomic writes of atomic locations are allowed, and can be within a release sequence. An acquire load that reads such a write will create a happens-before edge in the model with release sequences, but not in the other. This discrepancy means that some programs are racy in the release-acquire model, but are not in the model with release sequences.

We introduce a condition, *atomic_initialisation_first*, that restricts to programs where these models are equivalent, and where thin-air behaviour is not allowed. The following definitions are from the *release_acquire_SC_memory_model*, that does without release sequences. This model will motivate the condition that must be imposed on programs to achieve equivalence.

In this new model, the consistency predicate and undefined behaviour will remain the same, but the calculated relations will change, omitting release sequences, and adopting a simpler sw calculation:

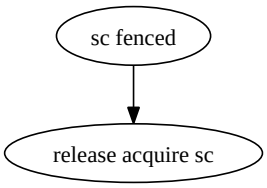
```
let release_acquire_synchronizes_with actions sb asw rf lo a b =
  (tid_of a ≠ tid_of b) ∧
  ( (* thread sync *)
    (a, b) ∈ asw ∨
    (* mutex sync *)
    (is_unlock a ∧ is_lock b ∧ (a, b) ∈ lo) ∨
    (* rel/acq sync *)
    ( is_release a ∧ is_acquire b ∧ (a, b) ∈ rf )
  )
```

The model condition for this equivalence imposes *atomic_initialisation_first*, and also requires that programs use no relaxed atomic accesses. Note the absence of the release and acquire memory orders from those allowed for read-modify-write actions. Those memory orders only synchronise part of the read-modify-write access, the write and the read part respectively. This would leave the other part of the access with relaxed semantics, so we forbid those memory orders. The release and acquire fences have also been dropped in this model; they are no longer useful in the absence of relaxed atomics.

```
let release_acquire_SC_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    atomic_initialisation_first (Xo, Xw, rl) ∧
    ∀ a ∈ Xo.actions.
      match a with
      | Lock _ _ _ _ → true
      | Unlock _ _ _ → true
      | Load _ _ mo _ _ → (mo ∈ {NA, Acquire, Seq_cst})
      | Store _ _ mo _ _ → (mo ∈ {NA, Release, Seq_cst})
      | RMW _ _ mo _ _ _ → (mo ∈ {Acq_rel, Seq_cst})
      | Fence _ _ mo → (mo ∈ {Seq_cst})
      | Blocked_rmw _ _ _ → true
      end
```

THEOREM 9.

```
(∀ opsem p.
  statically_satisfied release_acquire_SC_condition opsem p ⟶
  (behaviour sc_fenced_memory_model sc_fenced_condition opsem p =
   behaviour release_acquire_SC_memory_model release_acquire_SC_condition opsem p))
```

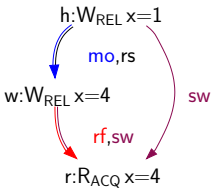


Now we show that this equivalence holds. Seeking equality in the absence of relaxed atomics, and expecting failure, compare the relations that result from substituting the two different sw relations into happens before:

```
let no_consume_hb sb sw =
    transitiveClosure (sb ∪ sw)
```

Seeking happens-before equality First note that *release_acquire_synchronizes_with* is a subset of *release_acquire_relaxed_synchronizes_with* (every release heads its own release sequence) and this inclusion can be lifted to the respective happens-before relations. The converse is not true however: the release sequence allows a read of a read-modify-write to synchronise with a seemingly unrelated release-write on a different thread. To show equality, it is sufficient to show that every synchronises-with edge in the model with release-sequences is covered by a happens-before edge in the model without.

Consider an arbitrary synchronises-with edge between a release write, h, and an acquire read, r. Then there exists some action w in the release sequence of h such that r reads from w:

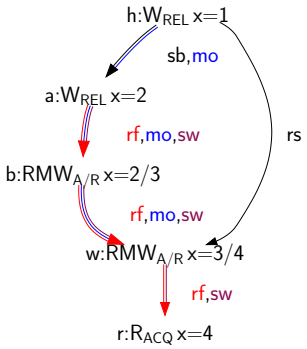


This shape generates a sw edge from h to r in the *sc_fenced_memory_model*. To show that there is a happens-before edge from h to r in the *release_acquire_SC_memory_model*, first consider the case where none of the actions in the release sequence is an initialisation write. Then, w must be a release, and r must be an acquire, and there is an sw edge, and consequently a happens-before edge (w, r). If w is equal to h, then the existence of the hb edge has been shown, so consider the case where they are different. Happens-before is transitve in this model, so it is sufficient to show that there is a happens-before egde from h to w.

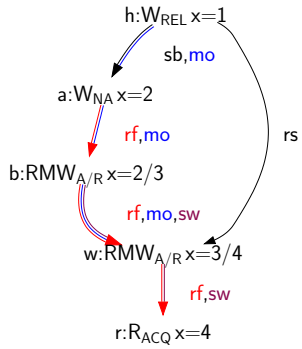
If w is on the same thread as h then *indeterminate_sequencing* from the consistency predicate guarantees that there is a sequenced before edge between the two actions. That edge must agree with modification order, and therefore, there is a happens-before edge from w to h as required.

In the case that w is not on the same thread as h, then w is a read-modify-write action. Consistency implies that modification order has finite prefixes, and that it is acyclic. The set of actions that are mo-before w where all actions are on a different thread from h is then either empty, or finite, and we can chose the minimal action in mo. Either way, we have an action, b, that must be a read-modify-write, whose immediate mo predecessor, a,

is on the same thread as h. By *rmw_atomicity*, b must read from a, and because neither is an initialisation, and there are no relaxed atomics, the two actions synchronise. Next, either a equals h or *indeterminate_sequencing* implies that there is a happens-before edge from h to a. Then either b is equal to r, and transitivity completes a happens-before edge from h to r, or there is a contiguous finite sequence of read-modify-write actions in mo between b and r. Each of these creates a happens-before edge to its predecessor, and again transitivity provides us with the required happens-before edge. The following example illustrates the case where neither h and a, nor b and r are equal:



Then we have the happens-before edge we required in the case that none of the writes in the release sequence were the initialisation. If initialisation writes are allowed in the release sequence, then it may not be possible to build a happens-before edge in the *release_acquire_SC_memory_model* coincident with every synchronises-with edge in the *sc_fenced_memory_model*. There are several examples that illustrate this: in the first, the non-atomic initialisation write takes the position of a in the diagram above — the last action on the same thread as h before a series of read-modify-write actions that are eventually read by r. Then there is no happens-before edge from a to its successor, and the chain of happens-before edges is broken:



Similarly, r may read directly from a, but in the case that a is an initialisation, then it is not a release, and again the chain of happens-before edges is broken.

It is sufficient then, for equivalence, to ensure that initialisation writes cannot appear in any release sequence in which they might be read by a read-modify-write action. This

is a dynamic property that can only be checked by calculating the set of consistent executions. It would be preferable to have a property that provides this guarantee that can be statically checked. *atomic_initialisation_first* is stronger than necessary, but it is sufficient for equivalence, and seems like a reasonable discipline to require of a programmer. It requires that for every non-atomic store at an atomic location, a, and any other write at the same location, b, b is not a non-atomic write and b follows a in sequenced-before union additional-synchronises-with:

```
let atomic_initialisation_first (Xo, _, _) =
  ∀ a ∈ Xo.actions b ∈ Xo.actions.
    is_at_atomic_location Xo.lk a ∧ is_NA_store a ∧
    is_write b ∧ (loc_of a = loc_of b) ∧ (a ≠ b) ⟶
    ((a, b) ∈ transitiveClosure (Xo.sb ∪ Xo.asw)) ∧ ¬ (is_NA_store b)
```

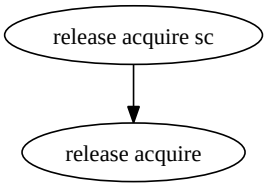
This property implies that there is a single initialisation of each atomic location, and that it happens before all writes to the same location. Equivalence of the *release_acquire_SC_memory_model* and the *sc_fenced_memory_model* has been established in HOL4 with this condition, following the argument provided above for the equivalence of the happens-before relations.

Causal consistency — the release-acquire model Without SC atomics or fences, the model becomes conceptually simple, providing only the release-acquire atomics with the straightforward definition of sw given above. The following equivalence relates the *release_acquire_memory_model* of Section 3.4 to the *release_acquire_SC_memory_model* defined above. The equivalence is straightforward to prove; the parts of the model that deal with SC atomics and fences are simply omitted.

```
let release_acquire_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    atomic_initialisation_first (Xo, Xw, rl) ∧
    ∀ a ∈ Xo.actions.
      match a with
      | Lock _ _ _ _ → true
      | Unlock _ _ _ → true
      | Load _ _ mo _ _ → (mo ∈ {NA, Acquire})
      | Store _ _ mo _ _ → (mo ∈ {NA, Release})
      | RMW _ _ mo _ _ _ → mo = Acq_rel
      | Fence _ _ _ → false
      | Blocked_rmw _ _ _ → true
      end
```

THEOREM 10.

$(\forall\ opsem\ p.$
$\quad statically\_satisfied\ release\_acquire\_condition\ opsem\ p \longrightarrow$
$\quad (behaviour\ release\_acquire\_memory\_model\ release\_acquire\_condition\ opsem\ p =$
$\quad\ behaviour\ release\_acquire\_SC\_memory\_model\ release\_acquire\_SC\_condition\ opsem\ p))$

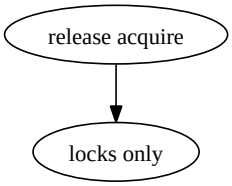


The model without atomics The final equivalence in this sequence of models relates the *release_acquire_memory_model* to the *locks_only_memory_model*. Observe that the condition on this equivalence subsumes the previous one — *atomic_initialisation_first* only applies to atomic locations. Again, this equivalence is straightforward, and simply removes the parts of the model that apply to atomic accesses.

```
let locks_only_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    ∀ a ∈ Xo.actions.
      match (loc_of a) with
      | Nothing → false
      | Just l → (Xo.lk l ∈ {Mutex, Non_Atomic})
      end
```

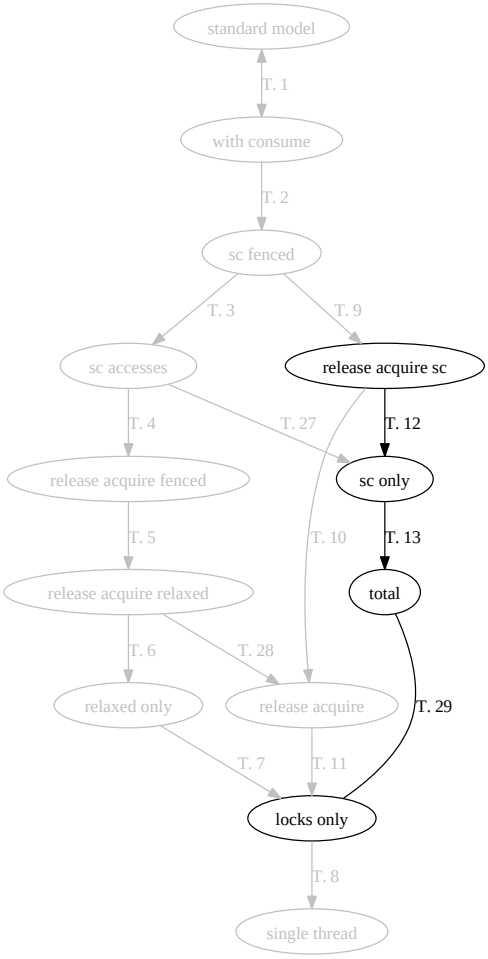
THEOREM 11.

$(\forall\ opsem\ p.$
$\quad statically\_satisfied\ locks\_only\_condition\ opsem\ p \longrightarrow$
$\quad (behaviour\ SC\_memory\_model\ SC\_condition\ opsem\ p =$
$\quad\ behaviour\ locks\_only\_memory\_model\ locks\_only\_condition\ opsem\ p))$



6.5 SC behaviour in the C/C++11 memory model

This section presents the equivalence between the C/C++11 memory model and a sequentially consistent memory model. The following overview graph shows the sequence of equivalences that provide this result:



Boehm and Adve described a desirable property of the C+11 memory model [37]: race free programs that use only the SC atomics, mutexes and non-atomic accesses have SC behaviour. They proved this result, by hand, for a precursor of the C++11 memory model. This precursor model was much simpler than C++11: it has a single total order over all atomic accesses, rather than per-location modification orders and coherence requirements, and it did not have non-atomic initialisation, a complication that invalidated this property in early drafts of the standard (see Chapter 5 for details).

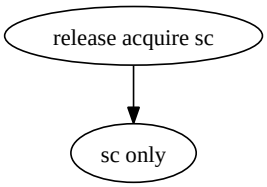
The results presented in this section establish, in HOL4, the property described by Boehm and Adve for the C++11 memory model, as ratified, albeit for a limited set of programs: those that only have finite pre-executions (so not those with recursion or loops).

The first equivalence in this sequence is a simple one — it removes the release and acquire atomics, leaving only atomics with the SC memory order:

```
let SC_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    atomic_initialisation_first (Xo, Xw, rl) ∧
    ∀ a ∈ Xo.actions.
      match a with
```

```
| Lock _ _ _ _ → true
| Unlock _ _ _ → true
| Load _ _ mo _ _ → (mo ∈ {NA, Seq_cst})
| Store _ _ mo _ _ → (mo ∈ {NA, Seq_cst})
| RMW _ _ mo _ _ _ → (mo ∈ {Seq_cst})
| Fence _ _ mo → false
| Blocked_rmw _ _ _ → true
end
```

THEOREM 12.
(∀ *opsem p*.
statically_satisfied SC_condition opsem p ⟶
(*behaviour SC_memory_model SC_condition opsem p* =
behaviour release_acquire_SC_memory_model release_acquire_SC_condition opsem p))



The *SC_memory_model* is new, but differs only slightly from the *release_acquire_SC_memory_model* defined above. Release-acquire synchronisation remains in the model, because SC atomics do create release-acquire synchronisation. The part of the consistency predicate that deal with SC fences is omitted as it places no restriction on the restricted set of programs. As a consequence, the equivalence proof is straightforward.

The totally ordered memory model

All of the memory models described so far have been based on partial orders. There has not been a global order of memory accesses that matches the intuition of time passing. The next equivalence will relate the C/C++11 memory model to such a model, and in so doing, validate a central tenet of the memory model design.

In order to state the proof of equivalence, we must define a sequentially consistent model that matches the following note from the standard, and is similar to the totally ordered model of Boehm and Adve. The note claims a property holds of the C/C++11 memory model design: that suitably restricted programs exhibit sequentially consistent behaviour.

1.10p21

> [. . .][*Note:* It can be shown that programs that correctly use mutexes and *memory_order_seq_cst* operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as "sequential consistency". However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result must perform an undefined operation. — *end note*]

The note refers to the sequentially consistent memory model of Lamport [60], presented in Chapter 2, where all memory accesses are interleaved, and loads from memory read the most recent write in this interleaving.

We present an axiomatic formulation of SC that follows Boehm and Adve. Rather than a partial happens-before relation, this model is governed by a total order over the actions, tot, that forms part of the execution witness. The structure of the model is the same as the previous models: there is a consistency predicate, calculated relations, and undefined behaviour. The model ensures that tot is a total order over the actions a new conjunct of the consistency predicate, *tot_consistent_tot*. It also ensures that tot agrees with sb and asw, and has finite prefixes:

```
let tot_consistent_tot (Xo, Xw, _) =
    relation_over Xo.actions Xw.tot ∧
    isTransitive Xw.tot ∧
    isIrreflexive Xw.tot ∧
    isTrichotomousOn Xw.tot Xo.actions ∧
    Xo.sb ⊆ Xw.tot ∧
    Xo.asw ⊆ Xw.tot ∧
    finite_prefixes Xw.tot Xo.actions
```

The tot relation replaces the host of partial relations that governed previous models. Take, for instance, the lock-order relation: in this model, the condition on locks is adjusted slightly so that tot can be used to similar effect:

```
let tot_consistent_locks (Xo, Xw, _) =
    (∀ (a, c) ∈ Xw.tot.
      is_successful_lock a ∧ is_successful_lock c ∧ (loc_of a = loc_of c)
```

$$\longrightarrow$$
$$(\exists\ b \in Xo.actions.\ (\text{loc\_of}\ a = \text{loc\_of}\ b) \wedge \text{is\_unlock}\ b \wedge (a,\ b) \in Xw.tot \wedge (b,\ c) \in Xw.tot))$$

Where in previous model, the happens-before relation was used to decide read values, in this model tot is used. The only calculated relation is a version of the visible-side-effect relation that uses tot in place of happens-before. For each read, it relates the most recent write to the read.

This model simplifies the semantics of reads by unifying the parts of the model that govern non-atomic loads, atomic loads and read-modify-writes. As part of this unification, the *det_read* condition changes slightly, to apply to all actions rather than just loads.

```
let tot_det_read (Xo, Xw, _ :: ("vse", vse) :: _) =
  ∀ r ∈ Xo.actions.
    (∃ w ∈ Xo.actions. (w, r) ∈ vse) =
    (∃ w' ∈ Xo.actions. (w', r) ∈ Xw.rf)
```

In the previous model, there are five conditions that identify the writes that may be read by the different sorts of read action:

- *consistent_non_atomic_rf*
- *consistent_atomic_rf*
- *coherent_memory_use*
- *rmw_atomicity*
- *sc_accesses_sc_reads_restricted*

In this model, the coherence requirements are no longer necessary, because there is only a single total order, and the four remaining conditions can be unified into a single one that applies to all reads:

```
let tot_consistent_rf (Xo, Xw, _ :: ("vse", vse) :: _) =
  ∀ (w, r) ∈ Xw.rf. (w, r) ∈ vse
```

The tot relation has replaced the mo, lo, sc and hb relations, and consequently this model omits the conditions from the consistency predicate of the *SC_memory_model* that checked the consistency of those relations:

- *locks_only_consistent_lo*
- *consistent_mo*
- *sc_accesses_consistent_sc*

- *consistent_hb*

Furthermore, there is no need to assume that each of the partial relations has finite prefixes. In this model it is assumed only that rf has finite prefixes:

```
let tot_assumptions (Xo, Xw, _) =
  finite_prefixes Xw.rf Xo.actions
```

The final differences between this model and the last are in the calculation of undefined behaviour. The total model shares the same calculation of unsequenced races and indeterminate reads, but the identification of bad mutex use changes slightly, and the definition of data races is quite different. The *tot_bad_mutexes* function first projects out a relation equivalent to lock order from tot, and then uses that to check the for violating mutex actions with the same predicate as the previous models:

```
let tot_bad_mutexes (Xo, Xw, _) =
  { a | ∀ a ∈ Xo.actions |
    let lo = { (a, b) | ∀ a ∈ Xo.actions b ∈ Xo.actions |
               ((a, b) ∈ Xw.tot) ∧ (loc_of a = loc_of b) ∧
               is_at_mutex_location Xo.lk a
             } in
    ¬ (locks_only_good_mutex_use Xo.actions Xo.lk Xo.sb lo a)}
```

It would defeat the purpose of the simplifications made here if one had to calculate races in terms of the previous models. Instead, the definition of races in this model follows the standard definition of races in sequentially consistent models similar to the one used in Boehm and Adve's paper [37]. Two distinct actions at the same location form a data race if at least one of them is a write, they are on different threads, at least one of them is not atomic, they are not ordered by asw, and the two actions are adjacent in tot:

```
let tot_data_races (Xo, Xw, _) =
    { (a, b) | ∀ a ∈ Xo.actions b ∈ Xo.actions |
       ¬ (a = b) ∧ (loc_of a = loc_of b) ∧ (is_write a ∨ is_write b) ∧
       (tid_of a ≠ tid_of b) ∧
       ¬ (is_atomic_action a ∧ is_atomic_action b) ∧
       ¬ ((a, b) ∈ Xo.asw) ∧
       (a, b) ∈ Xw.tot ∧
       ¬ (∃ c ∈ Xo.actions. ((a, c) ∈ Xw.tot) ∧ ((c, b) ∈ Xw.tot)) }
```

The condition necessary for the equivalence has two parts. The first, *atomic_initialisation_before_all*, is stronger than *atomic_initialisation_first*, requiring atomic reads to be ordered after the initialisation, as well as writes:

```
let atomic_initialisation_before_all (Xo, _, _) =
  ∀ a ∈ Xo.actions b ∈ Xo.actions.
    is_at_atomic_location Xo.lk a ∧ is_NA_store a ∧
    (loc_of a = loc_of b) ∧ (a ≠ b) ⟶
    ((a, b) ∈ transitiveClosure (Xo.sb ∪ Xo.asw)) ∧ ¬ (is_NA_store b)
```

The second condition will be used to enable finite induction over sets of executions of increasing size, ensuring that, in the limit, the set of finite executions covers all of the possible executions of the program. This is guaranteed by requiring a finite bound on the size of the action sets of all executions of the program:

```
let bounded_executions (Xs : SET CANDIDATE_EXECUTION) =
  ∃ N. ∀ (Xo, Xw, rl) ∈ Xs.
    finite Xo.actions ∧
    size Xo.actions < N
```

This condition limits the scope of the result significantly: programs that include loops or recursion do not satisfy the condition. Litmus tests without loops do however, so the equivalence result will apply to the litmus tests in Chapter 3, and those in the literature. We adopt the condition to simplify the proof of equivalence.

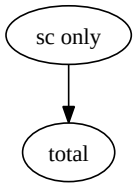
An alternative approach would be to define a relation over execution prefixes, show that it is well founded, and then use well-founded induction over prefixes of increasing size. This alternative approach would require additional assumptions that restrict the structure of the program, in order to avoid an infinite chain of happens-before-unrelated threads, breaking well-foundednesss. This might be achieved by adding thread-creation events to the model: on creating new threads there would be an event on the parent thread that would be related to thread-start events on the children. We would then require a parent to only create a finite number of threads with a single thread creation event. This assumption seems to be satisfied by the syntax of the language, and we conjecture that, with suitable assumptions, the theorem would hold over infinite executions.

With the definition of the totally ordered model and its conditions, it is possible to formally state the equivalence that **1.10p21** claims.

```
let tot_condition (Xs : SET CANDIDATE_EXECUTION) =
  bounded_executions Xs ∧
  ∀ (Xo, Xw, rl) ∈ Xs.
    atomic_initialisation_before_all (Xo, Xw, rl) ∧
    ∀ a ∈ Xo.actions.
      match a with
      | Lock _ _ _ _ → true
      | Unlock _ _ _ → true
```

```
  | Load _ _ mo _ _ → (mo ∈ {NA, Seq_cst})
  | Store _ _ mo _ _ → (mo ∈ {NA, Seq_cst})
  | RMW _ _ mo _ _ _ → (mo ∈ {Seq_cst})
  | Fence _ _ mo → false
  | Blocked_rmw _ _ _ → true
  end
```

THEOREM 13.



$$
\begin{aligned}
&(\forall\ opsem\ p.\\
&\quad opsem\_assumptions\ opsem \wedge\\
&\quad statically\_satisfied\ tot\_condition\ opsem\ p \longrightarrow\\
&\quad (rf\_behaviour\ SC\_memory\_model\ SC\_condition\ opsem\ p =\\
&\quad\ rf\_behaviour\ tot\_memory\_model\ tot\_condition\ opsem\ p))
\end{aligned}
$$

Equivalence proof overview

This proof of equivalence is rather more involved than those that have already been presented. The form of candidate executions differs between the two models: they have different execution witnesses and different calculated relations, so the equivalence that projects only rf from the execution witness is used. The proof relies on several assumptions on the thread-local semantics that enable induction over partial executions.

At the highest level, the proof involves showing that one can translate an execution from one model into an execution in the other. These translations will rely on the absence of undefined behaviour, and are supported by complementary proofs showing that undefined behaviour in one model implies undefined behaviour in the other.

The translation from a total order execution to a partial one is straightforward: each of the partial relations, lo, mo and sc is projected out of tot, and a translated execution witness gathers these translated relations together. In the other direction, a linearisation of the union of hb and sc forms the translated total order of the execution witness. Then, as we shall see in greater detail later, in the absence of faults, these translations produce consistent executions in their target models.

The treatment of faults that lead to undefined behaviour forms a large part of the proof effort. The line of argument relies on the notion of a *prefix*, a partial execution that includes all actions that are ordered before any action within the prefix. In either direction, the general form of the proof follows the same series of steps. Given a faulty execution in one model:

1. choose some order over the actions and find a minimal fault in that order,
2. construct a prefix of that fault that contains no other faults,
3. show that this prefix is consistent,

4. translate the prefix into an execution prefix in the other model,

5. show that this new execution prefix is consistent in the other model, using the fact that its progenitor is fault free,

6. extend this execution, adding the faulty action from the original execution, forming a new consistent prefix,

7. show that there is a fault in the extended prefix,

8. and finally, complete the extended prefix to make a full consistent execution with a fault.

Given a fault in one model, this sequence of steps witnesses a (possibly quite different) consistent execution in the other model that has a fault. This shows that a program with undefined behaviour under one model has undefined behaviour in the other.

The C/C++11 standard does not define prefixes of candidate executions or provide support for induction over such artefacts. In order to prove this equivalence, it is necessary to make several assumptions that enable this sort of reasoning. The next section presents these assumptions.

Assumptions on the thread-local semantics

The proof involves an induction that shows that a racy prefix with undefined behaviour can be extended to a full consistent execution with undefined behaviour. This section describes the assumption that we make over the thread-local semantics in order to provide support for this induction. It begins with the formal definition of an execution prefix.

A prefix is a set of actions that identifies a part of a pre-execution of a program. The prefix must be *downclosed*, that is, for any action in the prefix, all sb or asw predecessors are also in the prefix. The following definition captures this notion when used with the relation *sbasw*, also defined below:

let

$$downclosed\ A\ R\ =\ \forall\ a\ b.\ b \in A \wedge (a,\ b) \in R \longrightarrow a \in A$$

let $sbasw\ Xo\ =$ transitiveClosure $(Xo.sb \cup Xo.asw)$

The set of actions that identifies a prefix must be a finite subset of the actions of some pre-execution that the thread-local semantics produces from the program. The definition of a prefix is then:

let $is\_prefix\ opsem\ p\ Xo\ A\ =$
$opsem\ p\ Xo \wedge A \subseteq Xo.actions \wedge$ downclosed A (sbasw Xo) $\wedge$ finite A

The induction will proceed by adding actions to a prefix one at a time. A prefix of increased size will have to be *sbasw*-downclosed, so only actions that follow actions from the prefix in sb or asw, with no *sbasw*-intervening actions can be added. We precisely define these actions as the *fringe_set*: the minimal set of actions in *sbasw* that are not already contained in the prefix.

$$\textsf{let}\ \mathit{fringe\_set}\ \mathit{Xo}\ A\ =\ \text{minimal\_elements}\ (\setminus\ \mathit{Xo}.\mathit{actions}\ A)\ (\text{sbasw}\ \mathit{Xo})$$

The inductive step will require the addition of actions from the fringe-set. It will then be necessary to show that the extended prefix is consistent. If the action that is added is a read or a lock, it may have a value or outcome that is not consistent with the rest of the prefix. In that case, we assume *receptiveness*: that there is another execution produced by the thread-local semantics, where the new action reads a different value or produces a different lock outcome. In order to define this property of the thread-local semantics, we define a new relationship between pre-executions, *same-prefix*. This relationship is parameterised by an action set. It requires that the prefixes of the two pre-executions over the given set are equal, and so are their fringe sets:

$$\begin{array}{l}\textsf{let}\ \mathit{same\_prefix}\ \mathit{Xo}_1\ \mathit{Xo}_2\ A\ = \\ \quad \textsf{let}\ AF\ =\ A \cup \text{fringe\_set}\ \mathit{Xo}_1\ A\ \textsf{in} \\ \quad (\text{pre\_execution\_mask}\ \mathit{Xo}_1\ AF = \text{pre\_execution\_mask}\ \mathit{Xo}_2\ AF) \wedge \\ \quad (\text{fringe\_set}\ \mathit{Xo}_1\ A = \text{fringe\_set}\ \mathit{Xo}_2\ A)\end{array}$$

Now we can define receptiveness: for a given prefix of a pre-execution and action in the fringe-set of that prefix, the thread-local semantics admits pre-executions for every possible value of the action (or for a blocked outcome if the action is a lock), each of which has the same prefix and fringe set, differing only in the value of the action. The definition of receptiveness relies on the auxiliary definitions *replace-read-value* and *pre-execution-plug*, that respectively update the value of an action (or update the lock outcome to blocked), and replace one action with another in the action set and relations of a pre-execution. The definition of receptiveness is then:

```
let
  receptiveness opsem =
∀ p Xo A a.
  is_prefix opsem p Xo A ∧
  a ∈ fringe_set Xo A ∧
  (is_read a ∨ is_successful_lock a)
  ⟶
  ∀ v.
    let a′ = replace_read_value a v in
    ∃ Xo′.
      is_prefix opsem p Xo′ A ∧
      a′ ∈ fringe_set Xo′ A ∧
      same_prefix Xo′ (pre_execution_plug Xo a a′) A
```

It is also necessary to assume that the thread-local semantics produces only pre-executions that satisfy the well-formed-threads predicate:

```
let
  produce_well_formed_threads (opsem : OPSEM_T) =
  ∀ Xo p. ∃ Xw rl. opsem p Xo ⟶ well_formed_threads (Xo, Xw, rl)
```

These assumptions are collected together in the *opsem_assumptions* predicate:

```
let opsem_assumptions opsem =
  receptiveness opsem ∧
  produce_well_formed_threads opsem
```

Now the model condition and assumptions have been defined, we can present the equivalence theorem precisely:

```
THEOREM 1. (∀ opsem p.
  opsem_assumptions opsem ∧
  statically_satisfied tot_condition opsem p ⟶
  (rf_behaviour SC_memory_model SC_condition opsem p =
   rf_behaviour tot_memory_model tot_condition opsem p))
```

Top-level case split

At the highest level, there are two cases: the case where either model identifies a racy execution of the program, and the case where neither does. In the racy case, the proof proceeds by identifying a racy execution in the other model, so that the behaviour of the program is undefined in both models. In the race-free case, we need to show that there

exists a consistent execution in the other model with the same pre-execution and the same reads-from map. The racy and non-racy cases can each be split into two directions; this leaves us with four top level cases to consider.

In the discussion that follows, the *release_acquire_SC_memory_model* will be referred to as the *HB-model* and the *tot_memory_model* will be referred to as the *SC-model*. Then the four cases to prove are:

1. For any consistent execution of a race-free program in the HB-model, there is a consistent execution in the SC-model that shares the same pre-execution and reads-from map.

2. For any consistent execution of a race-free program in the SC-model, there is a consistent execution in the HB-model that shares the same pre-execution and reads-from map.

3. For any racy program in the SC-model, there is a consistent execution in the HB-model that has a race.

4. For any racy program in the HB-model, there is a consistent execution in the SC-model that has a race.

A Defined HB Execution Corresponds to a Consistent SC Execution

In this case, we start with a consistent execution in the HB-model, $X = (Xo, Xw, rl)$, with a finite action set, of a program with no undefined behaviour, and we need to construct a consistent execution in the SC-model, X', that shares the same pre-execution and reads-from map. We start by proving properties of lock order, reads-from and happens-before.

LEMMA 2. lo *restricted to the successful locks and unlocks is a subset of* hb

Proof. [This describes `lo_then_hb` in the formal proofs [2]] Without loss of generality, consider two arbitrary mutex actions, a and b, where a precedes b in hb. There are four cases for the four combinations of lock and unlock: unlock-lock, lock-lock, lock-unlock, and unlock-unlock.

The unlock-lock case can be further split into the case where the two actions are from different threads, and the case where they are on the same thread. The different thread case follows directly from the definitions of synchronises-with and happens-before. The same thread case follows from the indeterminate-sequencing conjunct of the HB-model's consistency predicate.

In the lock-lock case, the consistent-locks conjunct of the HB-model's consistency predicate implies that there is an intervening unlock action. Using the assumption that lock order has finite prefixes, identify the lo-minimal unlock between the two successful locks, c. This unlock happens-before b by the unlock-lock case established above. Now

note that there is no undefined behaviour, and as a consequence, there exists a successful lock, d, that is sequenced before c with no lock-order-intervening unlock between the two. Now d must equal a, otherwise, consistency would require that there exist an unlock at the same location lo-between d and a, and also lo-between d and c, a contradiction. This implies that there is a sequenced before edge from a to c, completing a happens-before edge from a to b, as required.

In the lock-unlock case, we use the lack of undefined behaviour to identify a successful lock, c that precedes b in lock order and sequenced-before. Then either c is equal to a or by the consistent-locks conjunct, a precedes c in lock order, and the lock-lock case above implies that a happens before c, establishing through transitivity a happens-before edge from a to b as required.

In the unlock-unlock case, we again use the lack of undefined behaviour to identify a successful lock, c that precedes b in lock order and sequenced before, such that there is no lo-intervening unlock between the two. This implies that a precedes c in lo, and we have that a happens before c by the unlock-lock case above. Transitivity then implies that a happens before b, as required.

All four cases have been shown to hold, so the lemma holds. □

Lemma 3. *Any* rf *edge between actions at an atomic location is also a* hb *edge.*

Proof. [This describes `rf_then_hb` in the formal proofs [2]] Name the reads-from related actions w and r. If w and r are on the same thread, then the indeterminate-sequencing and consistent-rf conjuncts of the HB-model's consistency predicate imply that they are ordered by happens-before. If they are on different threads and w is non-atomic, then the *atomic_initialisation_before_all* conjunct of the model condition implies that w happens before r. If w is atomic then it must be a release, r must be an acquire and the two actions synchronise, and w happens before r, as required. □

Lemma 4. *There exists a strict linear order, tot, over the actions of* Xo *that contains both the happens-before and SC order of* X*.*

Proof. [This describes `hb_consistent_fault_free_then_tot_order` in the formal proofs [2]] Define the relation *hbsc* as follows:

$$hbsc = (hb \cup sc)^+$$

Now we shall prove that the reflexive closure of *hbsc*, *hbscr*, can be extended to a linear order. First we must show that *hbscr* is a partial order over the actions of the pre-execution. Clearly the relation is transitive and reflexive, it remains to show that the domain and range of the relation is the actions and that the relation is antisymmetric.

The domain and range of *hbscr* is the actions *hbsc* is made up of relations whose domain and range are restricted the actions of the pre-execution by the consistency predicate of the HB-model. The domain and range of *hbscr* is therefore the same actions.

***hbscr* is antisymmetric** Again, this relies on the consistency predicate of the HB-model, this time the *sc_accesses_consistent_sc* predicate, that forbids happens-before and SC-order from having opposing edges. This, coupled with transitivity and acyclicity of both relations provides us with antisymmetry of *hbscr*.

We have established that *hbscr* is a partial order, so we can extend it to a reflexive total order *totr*, and then project the strict linear order *tot* from that. □

THEOREM 5. *There exists a strict linear order, tot, over the actions of Xo such that the candidate execution X' comprising Xo, Xw.rf and tot is a consistent execution in the SC-model.*

Proof. [This describes `hb_consistent_fault_free_then_tot_consistent` in the formal proofs [2]] First use Lemma 4 to identify *tot*, a strict linear order over the actions of Xo that contains both the happens-before and SC order of X.

Now consider the conjuncts of the consistency predicate of the SC-model. Several of the conjuncts are trivially satisfied because neither the relations they consider nor the conjuncts themselves have changed. This includes the *well_formed_threads* and *well_formed_rf* conjuncts. The *tot_assumptions* conjunct is trivially satisfied by the stronger *assumptions* conjunct that holds in the HB-model. The *tot_consistent_tot* conjunct is satisfied by construction: *tot* is a strict linear order over the actions of Xo. Three conjuncts remain to be shown for consistency: *tot_consistent_locks*, *tot_det_read* and *tot_consistent_rf*.

The *tot_consistent_locks* conjunct is defined as follows:

$$
\begin{aligned}
&\mathsf{let}\ \mathit{tot\_consistent\_locks}\ (Xo,\ Xw,\ \_)\ = \\
&\quad (\forall\ (a,\ c) \in Xw.\mathit{tot}. \\
&\qquad \text{is\_successful\_lock}\ a \wedge \text{is\_successful\_lock}\ c \wedge (\text{loc\_of}\ a = \text{loc\_of}\ c) \\
&\qquad \longrightarrow \\
&\quad (\exists\ b \in Xo.\mathit{actions}.\ (\text{loc\_of}\ a = \text{loc\_of}\ b) \wedge \text{is\_unlock}\ b \wedge (a,\ b) \in Xw.\mathit{tot} \wedge (b,\ c) \in Xw.\mathit{tot}))
\end{aligned}
$$

To show this holds, we must show that any two successful locks at the same location have a tot-intervening unlock at the same location. Without loss of generality, consider two arbitrary locks, a and c, where a precedes c in tot. There must be a lock-order edge between the two actions by the consistent-lock-order conjunct of the HB-model's consistency predicate. Lemma 2 implies that this lock order edge is also a happens-before edge, and happens before is a subset of tot. tot is transitive and irreflexive, so we know that a precedes c in lock order. Recall the consistent-locks conjunct of the HB-model's consistency predicate:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{locks\_only\_consistent\_locks}\ (\mathit{Xo},\ \mathit{Xw},\ \_)\ = \\
\quad (\forall\ (a,\ c) \in \mathit{Xw.lo}. \\
\quad\ \ \text{is\_successful\_lock}\ a \wedge \text{is\_successful\_lock}\ c \\
\quad\ \ \longrightarrow \\
\quad\ \ (\exists\ b \in \mathit{Xo.actions}.\ \text{is\_unlock}\ b \wedge (a,\ b) \in \mathit{Xw.lo} \wedge (b,\ c) \in \mathit{Xw.lo}))
\end{array}
$$

The existence of the lo-intervening unlock, together with Lemma 2, establishes *tot_consistent_locks*.

The *tot_det_read* conjunct is defined as follows:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{tot\_det\_read}\ (\mathit{Xo},\ \mathit{Xw},\ \_ :: (\text{“}\mathit{vse}\text{”},\ \mathit{vse}) :: \_)\ = \\
\quad \forall\ r \in \mathit{Xo.actions}. \\
\quad\ \ (\exists\ w \in \mathit{Xo.actions}.\ (w,\ r) \in \mathit{vse}) = \\
\quad\ \ (\exists\ w' \in \mathit{Xo.actions}.\ (w',\ r) \in \mathit{Xw.rf})
\end{array}
$$

First note that there is no undefined behaviour, so there can be no reads in the execution without a corresponding reads-from edge. Visible-side-effect edges only relate writes to reads, so if there exists a visible side effect of some action, it is a read, and there must be a write related to the read by a reads-from edge. It remains to show that if there is a reads from edge to a read, then that read has a visible side effect.

Given a reads-from edge to a read r, data-race freedom together with Lemma 3 imply that the reads from edge is coincident with a happens-before edge. That in turn implies that there is a tot edge between the two actions. Finiteness of the action set implies that we can find the maximal write at the location of r that precedes it in tot. This write is a visible side effect of r, as required.

The *tot_consistent_rf* conjunct is defined as follows:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{tot\_consistent\_rf}\ (\mathit{Xo},\ \mathit{Xw},\ \_ :: (\text{“}\mathit{vse}\text{”},\ \mathit{vse}) :: \_)\ = \\
\quad \forall\ (w,\ r) \in \mathit{Xw.rf}.\ (w,\ r) \in \mathit{vse}
\end{array}
$$

Label the actions related by the reads-from edge as w and r. The reads from edge from w to r, together with data-race freedom and Lemma 3 implies that w happens before r. Then we know that w precedes r in tot by the construction of tot.

Suppose, for a contradiction, that there is a tot intervening write to the same location, c, between w and r. The reads and writes cannot be at a non-atomic location: if they were, then the lack of data races would imply happens-before edges coincident with the tot edges, and the consisten-rf conjunct of the HB-model's consistency predicate would be violated. The actions must therefore be at an atomic location. By the *atomic_initialisation_before_all* conjunct of the model condition, only c is not a non-atomic write, and c and r must be SC actions ordered by the SC-order from c to r. Regardless of whether w is an SC write or a non-atomic write, the reads-from edge from w to r contradicts the sc-reads-restricted conjunct of the HB-model's consistency predicate. This

implies that there is no tot intervening write, c, to the same location, between w and r, and that w is a visible side effect of r, as required.

This completes the consistency predicate. □

A Defined SC Execution Corresponds to a Consistent HB Execution

In this case, we start with a consistent execution in the SC-model, $X = (Xo, Xw, rl)$, with a finite action set, of a program with no undefined behaviour, and we need to construct a consistent execution in the HB-model, X', that shares the same pre-execution and reads-from map.

We start by defining a new execution witness Xw_p, that will form part of the candidate execution in the HB-model. The components of Xw_p are simply projections of tot. There are three projections; the first is a projection of tot to the lock and unlock actions at each location:

$$
\begin{array}{l}
\mathsf{let}\ lo\_p\ tot_0\ Xo\ = \\
\quad \{(a,\ b) \mid (a,\ b) \in tot_0 \wedge (a \neq b) \wedge (\text{loc\_of}\ a = \text{loc\_of}\ b) \wedge \\
\qquad\quad (\text{is\_lock}\ a \vee \text{is\_unlock}\ a) \wedge (\text{is\_lock}\ b \vee \text{is\_unlock}\ b) \wedge \\
\qquad\quad \text{is\_at\_mutex\_location}\ Xo.lk\ a \\
\quad \}
\end{array}
$$

The second is a projection of tot to the write actions at each location:

$$
\begin{array}{l}
\mathsf{let}\ mo\_p\ tot_0\ Xo\ = \\
\quad \{(a,\ b) \mid (a,\ b) \in tot_0 \wedge \text{is\_write}\ a \wedge \text{is\_write}\ b \wedge (a \neq b) \wedge \\
\qquad\quad (\text{loc\_of}\ a = \text{loc\_of}\ b) \wedge \text{is\_at\_atomic\_location}\ Xo.lk\ a \\
\quad \}
\end{array}
$$

The third is a projection of tot to the SC actions:

$$
\begin{array}{l}
\mathsf{let}\ sc\_p\ tot_0\ = \\
\quad \{(a,\ b) \mid (a,\ b) \in tot_0 \wedge (a \neq b) \wedge \text{is\_seq\_cst}\ a \wedge \text{is\_seq\_cst}\ b\}
\end{array}
$$

The execution witness Xw_p is then defined as:

$$
\begin{array}{l}
\mathsf{let}\ Xw\_p\ tot_0\ Xo\ rf_0\ = \\
\quad \langle\!|\ rf\ =\ rf_0; \\
\qquad mo\ =\ \text{mo\_p}\ tot_0\ Xo; \\
\qquad sc\ =\ \text{sc\_p}\ tot_0; \\
\qquad lo\ =\ \text{lo\_p}\ tot_0\ Xo; \\
\qquad ao\ =\ \{\}; \\
\qquad tot =\ \{\}; \\
\quad |\!\rangle
\end{array}
$$

LEMMA 6. *Any edge in the reads-from relation is coincident with a visible side effect in the calculated relations of the projected execution witness, X'.*

Proof. [This describes `rf_then_hb_vse_thm` in the formal proofs [2]] Call the actions related by the reads-from relation w and r. The tot-consistent-rf conjunct of the SC-model's consistency predicate implies that w is a visible side effect of r with respect to the tot relation.

First we show that w happens before r. First consider the case where w and r are on the same thread. There is no undefined behaviour, so there can be no unsequenced races, and w and r must be ordered by sequenced before. Sequenced before is a subset of tot by the tot-consistent conjunct of the consistency predicate, so w happens before r. In the case where w and r are on different threads and at least one is non-atomic, race freedom implies and the fact that happens-before is a subset of tot implies that w happens before r. In the case where w and r are on different threads and both are atomic, they synchronise and w happens before r.

If there were a happens-before intervening write to the same location, the fact that happens-before is a subset of tot would make this write a tot-intervening write, violating the premise that w is a tot-visible-side-effect. Consequently, w is a happens-before visible side effect of r. □

THEOREM 7. *The consistent execution with the pre-execution Xo, the projected execution witness Xw_p, and the HB-model's calculated relations is a consistent execution in the HB-model.*

Proof. [This describes `total_consistent_fault_free_then_partial_consistent` in the formal proofs [2]] Consider the conjuncts of the consistency predicate of the HB-model. Several of the conjuncts are trivially satisfied because neither the relations they consider nor the conjuncts themselves have changed. This includes the *well_formed_threads* and *well_formed_rf* conjuncts. The *assumptions* conjunct is trivially satisfied because each of the relations it checks for finite prefixes is a subset of tot, that has finite prefixes by the *tot_consistent_tot* conjunct of the SC-model. The *locks_only_consistent_locks*, *locks_only_consistent_lo*, *consistent_mo*, *sc_accesses_consistent_sc* and *consistent_hb* conjuncts of the HB-model's consistency predicate all follow from the *tot_consistent_tot* conjunct of the SC-model and the construction of X': each of the relations that the predicates restrict is a subset of tot.

The *det_read* conjunct is defined as follows:

$$
\begin{aligned}
&\mathsf{let}\ \mathit{det\_read}\ (\mathit{Xo},\ \mathit{Xw},\ \_ :: (\text{“}\mathit{vse}\text{”},\ \mathit{vse}) :: \_)\ = \\
&\quad \forall\ r \in \mathit{Xo.actions}. \\
&\qquad \mathrm{is\_load}\ r \longrightarrow \\
&\qquad (\exists\ w \in \mathit{Xo.actions}.\ (w,\ r) \in \mathit{vse}) = \\
&\qquad (\exists\ w' \in \mathit{Xo.actions}.\ (w',\ r) \in \mathit{Xw.rf})
\end{aligned}
$$

There are two directions to establish. First assume there is a reads from edge, we must show that there is an Xw_p-happens-before visible side effect edge to the read. Lemma 6 implies that there is a visible side effect edge coincident with the reads-from edge, so it remains to show that if there is a tot visible side effect edge, then there is a reads from edge to the read. In this case, the fact that Xw_p happens before is a subset of tot implies that there is a write to the same location tot-before the read. We can then use the fact that tot has finite prefixes to find a tot-maximal write to the same location before the read, a tot visible side effect. Then we appeal to *tot_det_read* to identifies a reads from edge, as required.

The *consistent_non_atomic_rf* and *consistent_atomic_rf* conjuncts follows directly from Lemma 6 and the already established *consistent_hb* conjunct of the HB-model.

The *coherent_memory_use* conjunct is defined as follows:

```
let coherent_memory_use (Xo, Xw, ("hb", hb) :: _) =
  (* CoRR *)
  ( ¬ ( ∃ (a, b) ∈ Xw.rf (c, d) ∈ Xw.rf.
        (b, d) ∈ hb ∧ (c, a) ∈ Xw.mo ) ) ∧
  (* CoWR *)
  ( ¬ ( ∃ (a, b) ∈ Xw.rf c ∈ Xo.actions.
        (c, b) ∈ hb ∧ (a, c) ∈ Xw.mo ) ) ∧
  (* CoRW *)
  ( ¬ ( ∃ (a, b) ∈ Xw.rf c ∈ Xo.actions.
        (b, c) ∈ hb ∧ (c, a) ∈ Xw.mo ) ) ∧
  (* CoWW *)
  ( ¬ (∃ (a, b) ∈ hb. (b, a) ∈ Xw.mo) )
```

All of the relations that make up the four coherence shapes are subsets of tot. The tot-consistent-tot conjunct of the SC-model's consistency predicate provides that tot is transitive and irreflexive, so the CoRW and CoWW shapes are clearly forbidden, and the CoRR and CoWR shapes would violate the tot-consistent-rf conjunct if they existed, so those shapes are absent as well.

The *rmw_atomicity* conjunct is defined as follows:

```
let rmw_atomicity (Xo, Xw, _) =
  ∀ b ∈ Xo.actions a ∈ Xo.actions.
    is_RMW b ⟶ (adjacent_less_than Xw.mo Xo.actions a b = ((a, b) ∈ Xw.rf))
```

Given a read-modify-write action, r, there are two directions to establish. In the first, there is a reads from edge to r, and we must show that r reads the immediately preceding write in modification order. This follows from the tot-consistent-rf conjunct of the SC-model's consistency predicate together with the definition of modification order from Xw_p.

In the other direction, if there is an immediately preceding write in modification order, then we must show that r reads from that write. This follows from the tot-consistent-rf and tot-det-read conjuncts of the SC-model's consistency predicate together with the definition of modification order from Xw_p

The *sc_accesses_sc_reads_restricted* conjunct is defined as follows:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{sc\_accesses\_sc\_reads\_restricted}\ (\mathit{Xo},\ \mathit{Xw},\ (\text{“}\mathit{hb}\text{”},\ \mathit{hb}) :: \_)\ = \\
\quad \forall\ (w,\ r) \in \mathit{Xw.rf}.\ \text{is\_seq\_cst}\ r \longrightarrow \\
\quad\ (\ \text{is\_seq\_cst}\ w \wedge (w,\ r) \in \mathit{Xw.sc} \wedge \\
\quad\quad \neg\ (\exists\ w' \in \mathit{Xo.actions}. \\
\quad\quad\quad \text{is\_write}\ w' \wedge (\text{loc\_of}\ w = \text{loc\_of}\ w') \wedge \\
\quad\quad\quad (w,\ w') \in \mathit{Xw.sc} \wedge (w',\ r) \in \mathit{Xw.sc}\ )\ ) \vee \\
\quad\ (\ \neg\ (\text{is\_seq\_cst}\ w) \wedge \\
\quad\quad \neg\ (\exists\ w' \in \mathit{Xo.actions}. \\
\quad\quad\quad \text{is\_write}\ w' \wedge (\text{loc\_of}\ w = \text{loc\_of}\ w') \wedge \\
\quad\quad\quad (w,\ w') \in \mathit{hb} \wedge (w',\ r) \in \mathit{Xw.sc}\ )\ )
\end{array}
$$

This conjunct restricts which writes an SC read may read from. There are two cases: one for reading from an SC write and the other for reading from a non-SC write. Both cases forbid intervening writes in some combination of happens before and SC order, two relations that are subsets of tot. To see that the conjunct is satisfied, observe that the tot-consistent-rf conjunct of the SC-model's consistency predicate forbids such tot-intervening writes.

All of the conjuncts of the HB-model's consistency predicate have been shown to hold, so the lemma holds.

□

A Racy Program in the HB-model is Racy in the SC-model

In this case, we start with a consistent execution in the HB-model, $X = (Xo, Xw, rl)$, with a fault that leads to undefined behaviour. We need to construct a (possibly quite different) consistent execution in the SC-model, X', that has fault that leads to undefined behaviour. The proof will proceed as an induction over execution prefixes, and this induction requires the additional restriction that the executions of the program are bounded in size. This is guaranteed by the *bounded_executions* conjunct of *tot_condition*.

Define a *single fault* to be an indeterminate read, the second lock in a double lock or an unlock with no lock sequenced before it, and a *double fault* to be an unsequenced race, a data race or two adjacent unlocks in lock order.

In the proof below, it is necessary to add actions to a consistent execution prefix in the SC-model. On adding an action, we would like to identify a consistent execution that

incorporates the new action. This causes a problem: having received the underlying pre-execution from the HB-model or in the inductive step, in the presence of races the values attached to reads and the outcome of locks may not be consistent in the SC-model. We use our assumption of receptiveness over the thread-local semantics to choose a consistent value or lock outcome. Recall receptiveness:

```
let
  receptiveness opsem =
∀ p Xo A a.
  is_prefix opsem p Xo A ∧
  a ∈ fringe_set Xo A ∧
  (is_read a ∨ is_successful_lock a)
  ⟶
  ∀ v.
    let a′ = replace_read_value a v in
    ∃ Xo′.
      is_prefix opsem p Xo′ A ∧
      a′ ∈ fringe_set Xo′ A ∧
      same_prefix Xo′ (pre_execution_plug Xo a a′) A
```

In particular, we need to choose a value or lock outcome that will be consistent. Given a consistent prefix and an action in its fringe set, in order to add the action, we first append the new action to the end of tot. In the case of reads, if there is a preceding write to the same location, we set the value to that of the immediately preceding write, and choose an arbitrary value otherwise. In the lock case, we set the lock outcome to blocked. Then we produce an execution witness consisting of the original execution witness over the prefix, with the new action appended to tot, and a new reads-from edge if the new action was a read with a preceding write action. Receptiveness provides that this new extended pre-execution is a prefix of some pre-execution accepted by the thread-local semantics.

Lemma 8. *For a program that observes tot_condition, given a consistent execution X in the HB-model with a fault that leads to undefined behaviour, there exists a fault-free prefix containing all of the happens-before predecessors of a single or double fault.*

Proof. [This describes `exists_first_fault_hb_prefix` and `fault_then_exists_hb_minimal_fault` in the formal proofs [2])]] The execution X has at least one fault. Identify actions f and g, where either f is a single fault, or f does not happen before g, f and g together participate in a double fault, and if that fault is an unlock fault, g is ordered before f in lock order.

Define a subset of the actions, B, whose members are the happens-before-predecessors of f if f is a single fault, or the happens-before-predecessors of f and g if together they form a double fault.

Now define another set, *Fps*, the set of subsets of B whose elements are precisely the happens-before predecessors of a single fault or of the two actions that participate in a double fault. *Fps* is non-empty: B is a member. It is finite: the action set is finite, B is a subset of that, and *Fps* is a subset of the power set of B. The subset relation is acyclic over the elements of *Fps*, so we can find a minimal element in the set, *minFp*. There are no single or double faults in *minFp*, otherwise, it would not be minimal. *minFp* is a prefix over the execution: it is a subset of the actions, it is happens-before and *sbasw* downclosed and it is finite. We are done: *minFp* is the prefix we required. □

Lemma 9. *Adding an action to a prefix from its fringe set produces a new prefix.*

Proof. [This describes `add_from_fringe_set_prefix` in the formal proofs [2])]] Recall the definition of the fringe set: it is the minimal actions in happens-before that are not part of the prefix. Adding one of these actions keeps the set of actions downclosed, it remains finite, and the new action is from the action set of a pre-execution that the thread-local-semantics accepts, as required. □

Lemma 10. *Given a consistent prefix of some program in the HB-model, XA', adding an action to the prefix from its fringe set, with an adjusted value as specified above, produces a new prefix with a consistent execution, $XA'1$.*

Proof. [This describes `add_then_tot_consistent` in the formal proofs [2])]] Construct the new prefix execution as specified above. The `tot`-assumptions well-formed-threads, well-formed-rf, consistent-`tot`, `tot`-consistent-rf, `tot`-det-read and `tot`-consistent-locks conjuncts of the SC-model hold by construction. □

Theorem 11. *For a program that observes tot_condition, given a consistent execution X in the HB-model with a fault that leads to undefined behaviour, there exists an execution X' in the SC-model with a fault that leads to undefined behaviour.*

Proof. [This describes `exists_tot_consistent_with_fault`, `exists_tot_faulty_prefix` and `exists_min_fault_tot_prefix` in the formal proofs [2])]] We use Lemma 8 to identify a fault-free prefix of a single fault f or a double fault between f and g. We then restrict the components of the execution X to the prefix actions to produce an execution XA. XA is consistent by inspection of each of the conjuncts in the consistency predicate in the HB-model. We appeal to Lemma 5 to identify an execution, XA', of the same prefix restricted pre-execution in the SC-model, with the same reads from relation and a total order that contains both happens-before and the SC-order of XA.

Note that in the single fault case, there is a pre-execution where f is in the fringe set of the prefix covered by XA', and in the double fault case, there is a pre-execution with both f and g in its fringe set.

Next, use Lemma 10 to add in the action f if f is a single fault, or g and then f if not, giving a consistent prefix, $XA'1$, that contains the value-adjusted actions f' or f' and g' whose precursors exhibited a fault in the HB-model.

Now for each sort of fault, we show that either f' or g' and f' still exhibit a fault in the SC-model. If f was an indeterminate read in X then the original happens-before downclosed prefix of f contained no writes to the same location, so when $XA'1$ was constructed, f' would remain indeterminate.

For a lock fault f, the sequenced-before preceding lock, al, is in the happens-before downclosed prefix of f, and is therefore in $XA'1$. If there were a tot-intervening unlock, au, in $XA'1$, then au would certainly be lock-ordered before f in X. Lemma 2 implies that au must be lock-ordered after al, contradicting the fact that f is a lock-fault, so we have that f' is still a lock fault in the SC-model.

For an unlock fault f, there is no sequenced-before preceding lock at the same location. Construction of the extended execution $XA'1$ implies that there is no sequenced before preceding lock before f' either, so the fault remains.

For an unsequenced race between f and g, construction of the extended execution $XA'1$ implies that there is no sequenced before edge between f' and g' and the unsequenced race remains.

For a data race between f and g, construction of the extended execution $XA'1$ implies that the two actions are adjacent in tot, so a data race remains in the SC-model between f' and g'.

These four cases establish that the SC-model prefix $XA'1$ contains a fault. We must now extend the prefix execution to show the existence of an execution of the original program that has a race.

Lemma 10 tells us that if there are actions in the fringe set of a prefix then we can add them and get a larger prefix execution. This larger execution leaves the relations over the actions of the original prefix unchanged, so any faults within the prefix are preserved.

We shall show by induction that for any n, there is either a racy consistent execution of the program in the SC-model with fewer than n actions, or there is a racy prefix of at least n actions. We have already established the base case with $XA'1$, and we can freely add actions from the fringe set in the inductive step, if they exist. If they do not exist, then the prefix covers all of the actions, and we have racy consistent execution of the program, as required.

The tot-model condition requires all executions of a particular program to be bounded by some number N, so we chose an n greater than N, then we have witnessed a racy consistent execution of the program in the SC-model, as required. □

A Racy Program in the SC-model is Racy in the HB-model

In this case, we start with a consistent execution in the SC-model, $X = (Xo, Xw, rl)$, with a fault that leads to undefined behaviour. We need to construct a (possibly quite different) consistent execution in the HB-model, X', that has fault that leads to undefined behaviour. Again, the proof will proceed as an induction over execution prefixes, and

this induction requires the additional restriction that the executions of the program are bounded in size. This is guaranteed by the *bounded_executions* conjunct of *tot_condition*.

In the proof below, it is necessary to add actions to a consistent execution prefix in the HB-model. On adding an action, we would like to identify a consistent execution that incorporates the new action. As in the previous direction, we use our assumption of receptiveness over the thread-local semantics to choose a consistent value or lock outcome for reads and locks.

Given a consistent prefix and an action in its fringe set, we must add the action to the prefix to produce a new consistent prefix. How we do this depends on what sort of action it is. An unlock action is added to the end of lock order. The outcome of a lock action is first set to blocked, then the action is added to the end of lock order. If it has a visible side effect, the value of a non-atomic load action is set to the value of one of the visible side effects and a reads-from edge is added from the write to the read. If there is no visible side effect, then the load is added with no change to the execution witness. A store at a non-atomic location or a blocked read-modify-write action is added with no change to the execution witness. For atomic reads, we first change the value of the read to the value of the maximal write at the same location, and add an edge from the write to the read to the reads-from relation. Then, for fences or atomic accesses, if the access has memory order SEQ_CST, we add the action to the end of SC order. If the access is a write, then we add the action to the end of modification order.

Receptiveness provides that this new extended pre-execution is a prefix of some pre-execution accepted by the thread-local semantics.

LEMMA 12. *Adding an action to a prefix from its fringe set produces a new prefix.*

Proof. [This describes `add_from_fringe_set_prefix` in the formal proofs [2])]] Recall the definition of the fringe set: it is the minimal actions in happens-before that are not part of the prefix. Adding one of these actions keeps the set of actions downclosed, it remains finite, and the new action is from the action set of a pre-execution that the thread-local-semantics accepts, as required. □

LEMMA 13. *Given a consistent prefix of some program in the HB-model, XA', adding an action to the prefix from its fringe set, with an adjusted value as specified above, produces a new prefix with a consistent execution, $XA'1$.*

Proof. [This describes `add_then_hb_consistent` in the formal proofs [2])]] Construct the new prefix execution as specified above. The assumptions, well-formed-threads, well-formed-rf, locks-only-consistent-locks, locks-only-consistent-lo, consistent-mo, sc-accesses-consistent-sc, consistent-hb, det-read, consistent-non-atomic-rf, consistent-atomic-rf, coherent-memory-use, rmw-atomicity and sc-accesses-sc-reads-restricted conjuncts of the HB-model all hold by construction. □

THEOREM 14. *For a program that observes tot_condition, given a consistent execution X in the SC-model with a fault that leads to undefined behaviour, there exists an execution X' in the HB-model with a fault that leads to undefined behaviour.*

Proof. [This describes *tot_exists_minimal_translated_race* and *no_tot_hb_translated_race* in the formal proofs [2])]] The execution X has at least one fault. Recall that the consistency predicate gives that tot is acyclic and has finite prefixes. Find the tot-minimal single or double fault where either f is the tot-minimal single fault, or g is tot-before f, f and g together participate in a tot-minimal double fault.

Define A as the set of tot-predecessors of f. The set A forms a prefix over X: tot has finite prefixes and both sequenced-before and additional-synchronises-with are subsets of tot.

Restrict the components of the execution X to the prefix actions to produce an execution XA. XA is consistent by inspection of each of the conjuncts in the consistency predicate in the SC-model. Use Lemma 7 to identify an execution, XA', of the same prefix restricted pre-execution in the HB-model, with the same reads from relation and execution-witness relations that are projected from the total order of XA. Note that in the single fault case, there is a pre-execution where f is in the fringe set of the prefix covered by XA', and in the double fault case, there is a pre-execution with both f and g in its fringe set.

Next, use Lemma 13 to add in the action f if f is a single fault, or g and then f if not, giving a consistent prefix, $XA'1$, that contains the value-adjusted actions f' or f' and g' whose precursors exhibited a fault in the tot-model.

Now for each sort of fault, we show that either f' or g' and f' still exhibit a fault in the HB-model. If f was an indeterminate read in X then the original tot downclosed prefix of f contained no writes to the same location, so when $XA'1$ was constructed, f' would remain indeterminate.

For a lock fault f, the sequenced-before preceding lock, al, is in the tot downclosed prefix of f, and is therefore in $XA'1$. If there were a lock-order-intervening unlock, au, in $XA'1$, then au would be tot-intervening between al and f in X, by construction of the execution witness. This would contradict the fact that f is a lock-fault, so we have that f' is still a lock fault in the HB-model. For an unlock fault f, there is no sequenced-before preceding lock at the same location. Construction of the extended execution $XA'1$ implies that there is no sequenced before preceding lock before f' either, so the fault remains. For an unsequenced race between f and g, construction of the extended execution $XA'1$ implies that there is no sequenced before edge between f' and g' and the unsequenced race remains. For a data race between f and g, the construction of the extended execution $XA'1$ added g just before f. One of the two actions is non-atomic, so the two actions cannot synchronise, and they remain unordered by happens before in $XA'1$, forming a data race in the HB-model.

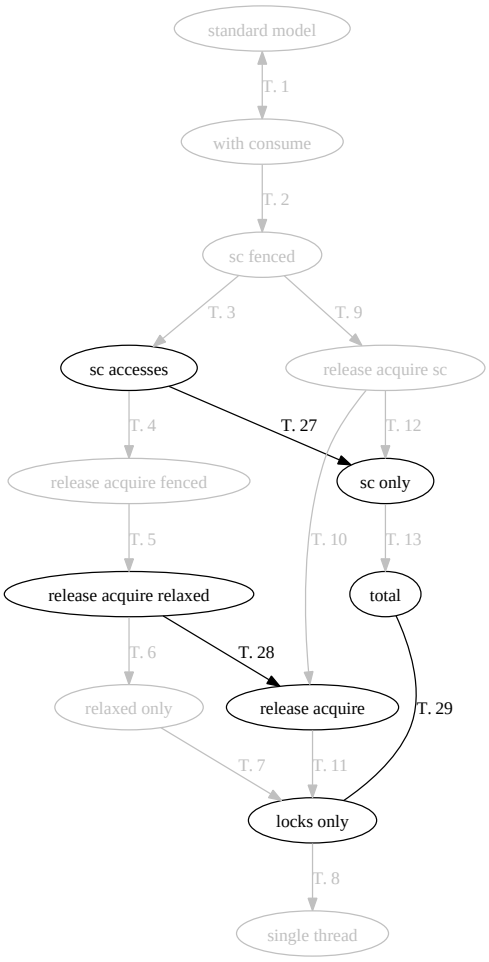
These four cases establish that the HB-model prefix $XA'1$ contains a fault. We must now extend the prefix execution to show the existence of an execution of the original program that has a race.

Lemma 13 tells us that if there are actions in the fringe set of a prefix then we can add them and get a larger prefix execution. This larger execution leaves the relations over the actions of the original prefix unchanged, so any faults within the prefix are preserved.

The remaining steps in the proof are precisely the same as those of the proof of Lemma 11. □

6.6 Linking the three strands of equivalence

This section presents three results that link the three strands of equivalences. These results complete the overview graph:

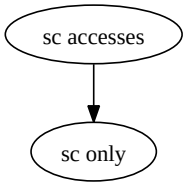


The first two results rely on the following observation: where the arrows are directed in the graph, the model condition of the destination of the edge is strictly stronger than the origin. We can use this property to chain equivalences back through directed edges in the graph to show, for instance, that for programs meeting the sc-only model condition, the sc-only memory model is equivalent to the sc-fenced memory model. With this observation,

we need only show that the sc-only model condition is stronger than the sc-accesses model condition to establish the first result:

THEOREM 14.

$$
\begin{aligned}
&(\forall\ opsem\ p.\\
&\quad statically\_satisfied\ SC\_condition\ opsem\ p \longrightarrow\\
&\quad (behaviour\ SC\_memory\_model\ SC\_condition\ opsem\ p =\\
&\quad behaviour\ sc\_accesses\_memory\_model\ sc\_accesses\_condition\ opsem\ p))
\end{aligned}
$$

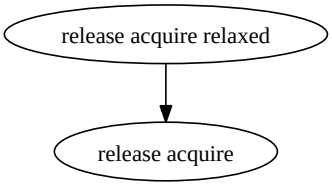


The sc-accesses model condition admits programs that use the release, acquire, relaxed and SC atomics, whereas the sc-only model condition allows only SC atomics, and is therefore strictly stronger. Appealing to Theorems 3, 9 and 12 completes the equivalence proof.

The second result is similar: the stronger protocol implies equivalence through the graph.

THEOREM 15.

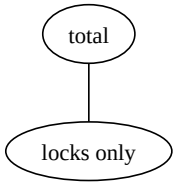
$$
\begin{aligned}
&(\forall\ opsem\ p.\\
&\quad statically\_satisfied\ release\_acquire\_condition\ opsem\ p \longrightarrow\\
&\quad (behaviour\ release\_acquire\_memory\_model\ release\_acquire\_condition\ opsem\ p =\\
&\quad behaviour\ release\_acquire\_relaxed\_memory\_model\ release\_acquire\_relaxed\_condition\ opsem\ p))
\end{aligned}
$$



In the final result, neither the total model condition nor the locks-only model condition is stronger than the other, so we adopt their conjunction as the condition on programs:

THEOREM 16.

$$
\begin{aligned}
&(\forall\ opsem\ p.\\
&\quad opsem\_assumptions\ opsem \wedge\\
&\quad statically\_satisfied\ tot\_condition\ opsem\ p \wedge\\
&\quad statically\_satisfied\ locks\_only\_condition\ opsem\ p\\
&\quad \longrightarrow\\
&\quad (rf\_behaviour\ locks\_only\_memory\_model\ locks\_only\_condition\ opsem\ p =\\
&\quad\ rf\_behaviour\ tot\_memory\_model\ tot\_condition\ opsem\ p))
\end{aligned}
$$



Together the two model conditions are stronger than the model conditions of their predecessors in the overview graph, and the two models are equivalent for programs that satisfy the combined condition.

Chapter 7

Compilation strategies

This chapter presents joint work with Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell, presented in POPL 2011 [28], POPL 2012 [26], and PLDI 2012 [96].

In Chapter 2, we introduced compilation mappings from C/C++11 atomic library calls to machine code. These mappings correspond to the transformations that occur in compilation. The mappings are simplistic: compilers do not simply map from the source program to machine instructions — they perform many optimisations that can affect the memory behaviour of the program. A sound mapping shows that it is at least possible to compile programs correctly to a given hardware architecture.

Early iterations of these mappings [107, 78, 106] formed an intrinsic part of the design process of the language. For each accessor function in the atomic library and each choice of memory order parameter, there is a different fragment of machine code that implements the function on each hardware architecture. The production of these tables requires one to settle on a design for the C/C++11 atomics that is both efficiently implementable across the diverse target architectures, and that provides usable abstractions for programmers.

These mappings *explain* the design of the language: they make it clear why the more intricate features exist. The C/C++11 design would be needlessly complex if it targeted only x86 because the most complicated memory accesses have implementations that behave in a constrained and simple way, and provide only questionable benefit to performance (perhaps through more agressive optimisation) over the simpler features; programmers would be wise to simply ignore consume and relaxed atomics. The Power and ARM mappings have different machine-code implementations for relaxed and consume atomics that makes their purpose clear: on these architectures, their implementations do without barriers (and the associated performance detriment) that are necessary in the implementation of acquire and release atomics. By linking the atomic functions to the barriers that are necessary to implement them, the mappings also support the implicit language-level assumption about performance: relaxed and consume accesses are cheap, release-acquire less so, SC atomics are expensive, and locks more so.

This chapter presents theorems that establish the soundness of the mappings for x86 and Power. These results show that it is possible for a compiler to correctly implement the language above the x86 and Power architectures. They do not apply to compilers that optimise the atomics or fences, but should be applicable to compilers that optimise non-atomic blocks of code (if an optimisation affects the program's semantics, then there was a race). In addition to implementability, we establish that the mappings are locally optimal in the following sense: if they were weakened in any way, they would be unsound. This establishes that the C/C++11 design is not overly complicated — for instance on Power, relaxed, release-acquire and SC atomic accesses each map to different snippets of machine code, the relative cost of which is as expected.

We include sketch proofs of the part of the mapping that covers loads, stores and fences. For the complete proofs and the part of the mappings that include locks and atomic compare-and-swap-like features, see the papers [28, 26, 96].

7.1 x86 mapping correctness

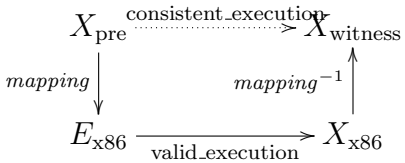
The mapping from C++11 atomics to x86 machine code is presented below:

C/C++11	x86
`load` RELAXED	`MOV` (from memory)
`load` CONSUME	`MOV` (from memory)
`load` ACQUIRE	`MOV` (from memory)
`load` SEQ_CST	`MOV` (from memory)
`store` RELAXED	`MOV` (into memory)
`store` RELEASE	`MOV` (into memory)
`store` SEQ_CST	`MOV` (into memory); `MFENCE`
`fence` ACQUIRE	⟨ignore⟩
`fence` RELEASE	⟨ignore⟩
`fence` ACQ_REL	⟨ignore⟩
`fence` SEQ_CST	`MFENCE`

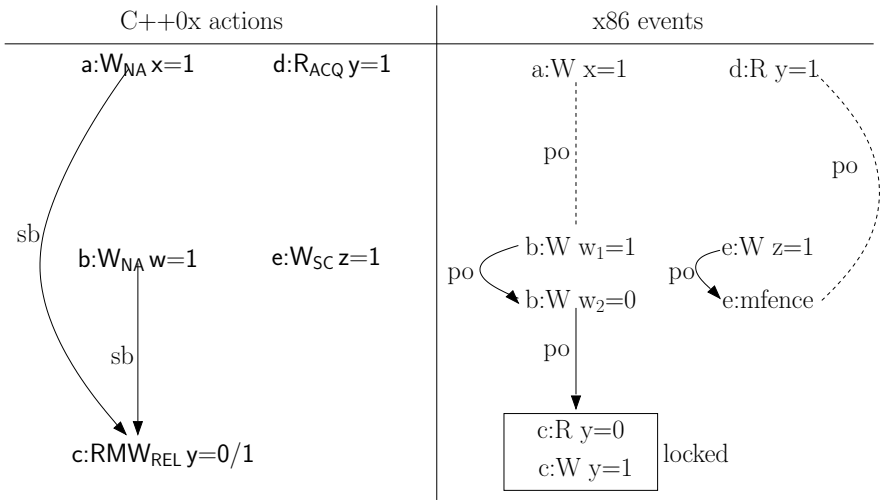
Note that most of the C/C++11 features can be implemented with plain loads and stores on x86 (`MOV`s to and from memory). This is because the x86 memory model provides stronger ordering guarantees than the C/C++11 memory model: the x86 memory model admits only store-buffering relaxed behaviour (see Chapter 2 for details), but only programs that use SC atomics or SC fences forbid store buffering at the language level, so only these features require the addition of hardware fences. The mapping above is only one possible implementation of the language, and it embodies a design decision. We need to emit an `MFENCE` either after each SC store or before each SC load. The mapping chooses to apply the fence to the store for performance, on the assumption that many programming idioms feature infrequent stores and frequent loads.

This section describes a theorem that is stated in terms of the axiomatic x86-TSO model of Owens et al. [99, 91, 104]. The model takes a program and produces an *x86 event structure*, E_{x86}, a set of events ordered by program order — the analogue of a C/C++11 pre-execution. This event structure is then combined with a reads-from relation and *x86 memory order*, a relation that totally orders writes, to form an x86 execution witness, X_{x86}. The model constrains the memory order and reads from relations that can be observed for a given event structure.

We would like to show that, for any program, the mapping above preserves its semantics in C/C++11 over the x86 architecture. In order to do this, we start with the set of pre-executions of the program in the C/C++11 model, then for any pre-execution, X_{pre}, we show that if we (non-deterministically) translate the pre-execution to an x86 event structure, E_{x86}, execute that E_{x86} according to the x86 axiomatic model to get an execution witness, X_{x86}, and then translate the x86 execution witness back to a C/C++11 execution, X_{witness}, then X_{witness} is a consistent execution of the pre-execution in the C/C++11 memory model. The graph below represents this theorem, with the dotted arrow representing the implied relationship:



The translation of a C/C++11 pre-execution to an x86 event structure is not direct: the sequenced-before relation of a C/C++11 pre-execution is partial, whereas x86 event structures have a total program order relation over the events of each thread. In translating from C/C++11 to x86, we must arbitrarily linearise the translated events to produce a valid x86 program order. The following example highlights a choice of program order with dotted lines for a given pre-execution, and exercises some of the cases of the mapping:



Note the translation of the single non-atomic write of w to a pair of writes to addresses w_1 and w_2 in the x86 event structure. We have to record the choices of data layout that occur in translation. To this end, we define a finite map from sets of x86 addresses (each set corresponding to a C/C++11 location), to x86 values. The mapping must be injective, each C/C++11 location must have an entry in the map, the addresses of any two entries must be disjoint, and C/C++11 atomic locations must have singleton address sets in their entries in the map. Now we can state the theorem that shows correctness of the mapping — see [39] for the precise definitions, statement, and proof:

THEOREM 15. *Let p be a C++ program that has no undefined behaviour. Suppose also that p contains no SC fences, forks, joins, locks, or unlocks. Then, if actions, sequenced-before, and location-kinds are members of the X_{pre} part of a candidate execution resulting from the thread-local semantics of p, then the following holds:*

For all compilers, C, finite location-address maps, and x86 executions X_{x86}, if C produces only event structures that correspond to the application of the mapping and finite map to X_{pre}, and X_{x86} is a valid execution of such an event structure in the x86 axiomatic model, then there exists a consistent execution of X_{pre}, X_{witness}, in the C/C++11 model.

Proof outline. We construct an X_{witness} from X_{x86} by reversing the mapping, lifting the reads-from and memory-order relations to relations over X_{pre} and projecting the second to a per-location relation over the writes. This gives us the reads-from and modification order relations of X_{witness}. We construct SC order by lifting x86 memory order to a relation over X_{pre} and then projecting it to a relation over the SC atomics. This relation will not necessarily be total over the SC atomics, so we linearise it using a proof technique from [89]. It remains to show that X_{witness} is a consistent execution.

The rest of the proof relies on two insights: first, x86 program order and memory order lifted to the C/C++11 actions cover the happens-before relation, and second, either reads

are consistent when translated, or there is a racy execution of the program. The first has been established by Owens with the HOL4 theorem prover, and the second by hand.

The proof of consistency involves proving each of the conjuncts of the consistency predicate. Consider the consistent-non-atomic-rf conjunct as an example. We must show that for a reads-from edge at an atomic location from w to r, w is a visible side effect of r. There is no happens-before intervening write to the same location: if there were, then there would be an intervening write in some combination of x86 program-order and memory-order, and the reads-from edge would not be valid in the x86 model, a contradiction. We must also show the existence of a happens-before edge from w to r. Valid executions in the x86 model are all allowed to read from writes in a way that is not consistent in the C/C++11 model. In particular, reads from edges might be created from writes that would be disallowed in C/C++11 or non-atomic reads-from edges might not be well formed for multi-address accesses. Call either behaviour a fault. We shall show that if there is a fault, then there was a race in the original program, forming a contradiction. Start by identifying a minimum fault, following [89]. Create a prefix of the fault. This prefix is consistent in the C/C++11 memory model. Add the actions that led to the fault, showing that these actions constitute a race in the C/C++11 model. Now complete the execution to produce a consistent execution with a race. This contradicts the premise of the theorem. □

7.2 Power mapping correctness

The mapping from C++11 atomics to Power machine code is presented below:

C/C++11	Power
`load` RELAXED	`ld`
`load` CONSUME	`ld` + keep dependencies
`load` ACQUIRE	`ld; cmp; bc; isync`
`load` SEQ_CST	`hwsync; ld; cmp; bc; isync`
`store` RELAXED	`st`
`store` RELEASE	`lwsync; st`
`store` SEQ_CST	`hwsync; st`
`fence` ACQUIRE	`lwsync`
`fence` RELEASE	`lwsync`
`fence` ACQ_REL	`lwsync`
`fence` SEQ_CST	`hwsync`

On the Power architecture, plain loads and stores are performed with the `ld` and `st` instructions. Contrast the mapping with that of x86: here, we have to insert additional synchronisation in the implementation of acquire and release atomics and fences,

and consume atomics require the compiler to preserve any following data or control dependencies. The architecture admits a variety of relaxed behaviour, and the hardware synchronisation in each row of the mapping is chosen to forbid relaxed behaviour as required by the language memory model. Again, this is not the only mapping that would preserve the semantics of the language, but it is locally-optimal: weakening the hardware synchronisation in any row of the mapping would leave it unsound.

7.2.1 Informal correctness of the mapping

This section presents several examples that exercise the compilation mapping, arguing that in each the semantics of the C/C++11 program is preserved.

The happens-before relation is central to the C/C++11 memory model. In particular, release-acquire synchronisation enables one to program using the message-passing idiom by reasoning about dynamically-created happens-before edges in the executions of the program. The first example is a C++11 variant of the message-passing test; we use it to explore how the mapping preserves the semantics of release-acquire synchronisation:

```
int x;
atomic<int> y(0);
// sender thread T0
x=1;
y.store(1,memory_order_release);
// receiver thread T1
while (0==y.load(memory_order_acquire)) {}
int r = x;
```

Applying the mapping to this program yields the following Power assembly program:

y=0			
T0		T1	
r1=1; r2=&x; r3=&y		r2=&x; r3=&y	
a: `stw r1,0(r2)`	write `x=1`	`loop:`	
b: `lwsync`	from write-rel	d: `lwz r4,0(r3)`	read `y`
c: `stw r1,0(r3)`	write `y=1`	e: `cmpwi r4,0`	
		f: `beq loop`	
		g: `isync`	from read-acq
		h: `lwz r5,0(r2)`	read `x`

The program is presented in Power assembly syntax: `stw` is a store, `lwz` is a load, `cmpwi` is a compare-immediate, and `beq` is a conditional branch. The loads and stores correspond to those present in the original C/C++11 program, and the additional instructions are included by the mapping. On Thread 0, the `lwsync` is added in the translation of the release write, and on Thread 1, the compare-branch-isync trio, called a *control-isync*,

arises from the read acquire.

In the naming convention established by Maranget et al. [71], this Power test is called MP+lwsync+ctrlisync: the test shape is message-passing (MP), there is an `lwsync` between the left-hand pair of actions and there is a control dependency followed by an `isync` between the right-hand pair. See Maranget et al. [71] for other variants of this test, other tests, and experimental data on observation of relaxed behaviour on various hardware.

In C/C++11, Thread 1 may not break out of the while loop and then fail to see Thread 0's write of `x`. In the translated Power program, the synchronisation instructions added by the mapping must prevent this behaviour. Chapter 2 described the three ways that the Power architecture might allow the relaxed behaviour: the writes of Thread 0 might be committed out of order, they might be propagated out of order, or the second read on Thread 1 might be speculated. The additional instructions included by the mapping forbid all three behaviours: the `lwsync` prevents commit and propagation reordering of the writes in Thread 0, and the control-isync prevents read speculation on Thread 1. Note that if either the `lwsync` or the control-isync were removed or replaced with weaker synchronisation, then the relaxed behaviour would be allowed.

Now we explore the implementation of release-consume synchronisation with the following example C/C++11 program:

```
int x;
atomic<int *> p(0);
// sender thread
x=1;
p.store(&x,memory_order_release);
// receiver thread
int* xp = p.load(memory_order_consume);
int  r = *xp;
```

Once again, the behaviour that the language forbids is the outcome where Thread 1 first reads the address of `x`, yet fails to see Thread 0's write of. The corresponding Power program following the application of the mapping is:

p=0			
T0		T1	
r1=1; r2=&x; r3=&p		r3=&p	
a: `stw r1,0(r2)`	write `x=1`	d: `lwz r4,0(r3)`	read `p`
b: `lwsync`	from write-rel	e: `lwz r5,0(r4)`	read `*xp`
c: `stw r2,0(r3)`	write `p=&x`		

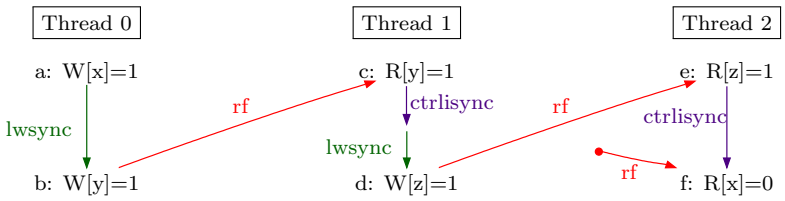
This Power test is called MP+lwsync+addr: the test is similar to the MP+lwsync+isync test above, but the control-isync has been replaced by an address dependency.

Thread 0 has an `lwsync` as in the release-acquire example, and forbids both commit and propagation reordering as a result, but Thread 1 no longer has a control-isync to prevent speculation of the second read. In this example, in the cases where Thread 1 sees the write of `p` by Thread 0, the address dependency from that read to the second read prevents speculation, and the relaxed behaviour is forbidden.

The two cases above show that the mapping effectively implements synchronisation across two threads, but much of the synchronisation in the C/C++11 memory model is transitive through other happens-before edges, and so far we have not argued that this transitivity is correctly implemented. The following program, a C++11 variant of ISA2, explores whether the mapping correctly implements the transitivity of happens-before through a chain of release-acquire synchronisation:

```
int x;  atomic<int> y(0);  atomic<int> z(0);
T0  x=1;
    y.store(1,memory_order_release);
T1  while (0==y.load(memory_order_acquire)) {}
    z.store(1,memory_order_release);
T2  while (0==z.load(memory_order_acquire)) {}
    r = x;
```

In this program, it would be unsound to optimise the program by reordering any of the accesses, so we consider the Power execution of a direct mapping of the program. The mapping introduces `lwsync`s and control-isyncs as before. The following Power execution shows the execution that should be forbidden:



Thread 1's control-isync is subsumed by its `lwsync`, and is not required in the following reasoning. The control-isync of Thread 2 prevents speculation, so we need only show that the writes of `x` and `z` are propagated to Thread 2 in order. Because of the `lwsync`, we know that Thread 0 propagates its writes to Thread 1 in order. We now use cumulativity to show that the write of `x` is propagated to Thread 2 before the write of `z`. In terms of the `lwsync` on Thread 1, the write of `x` is in its Group A, the set of writes that have already been propagated to the thread. Cumulativity implies that the write of `x` must have been propagated to Thread 2 already, before the write of `z`, as required.

Now consider a C/C++11 program that, according to the memory model, can give rise to IRIW behaviour:

```
atomic<int> x(0);  atomic<int> y(0);
T0  x.store(1,memory_order_release);
T1  r1=x.load(memory_order_acquire);
    r2=y.load(memory_order_acquire);
T2  y.store(1,memory_order_release);
T3  r3=y.load(memory_order_acquire);
    r4=x.load(memory_order_acquire);
```

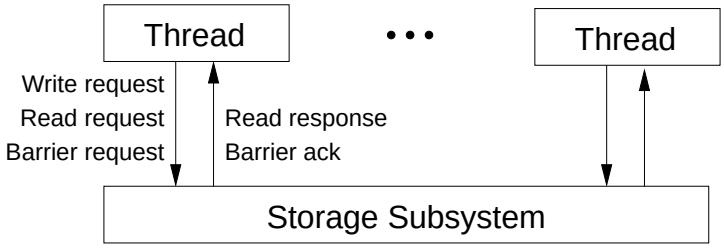
The Power analogue of this program allows IRIW behaviour. Full `sync` instructions would be required between the loads in order to forbid this. The `sync` is stronger than an `lwsync`: it requires all Group A and program-order preceding writes to be propagated to all threads before the thread continues. This is enough to forbid the IRIW relaxed behaviour. If we replace the memory orders in the program above with the `seq_cst` memory order, then the mapping would provide `sync` instructions between the reads, and the IRIW outcome would be forbidden in the compiled Power program, as required.

The examples presented here help to explain how the synchronisation instructions in the mapping forbid executions that the language does not allow. We can also show that if the mapping were weakened in any way, it would fail to correctly implement C/C++11 [26]. The proof of this involves considering each entry in the mapping, weakening it, and observing some new relaxed behaviour that should be forbidden.

For example, consider two of the cases in the mapping in the context of the examples above. First, if we remove the dependency from the implementation of consume atomics, then we enable read-side speculation in the message-passing example, and we allow the relaxed behaviour. Second, if we swap the `sync` for an `lwsync` in the implementation of SC loads, then we would be able to see IRIW behaviour in the example above.

7.2.2 Overview of the formal proof

The proof of correctness of the Power mapping (primarily the work of Sarkar and Memarian) has a similar form to the x86 result at the highest level, but the details are rather different. The Power model is an abstract machine, comprising a set of threads that communicate with a storage subsystem. In Chapter 2 the role of each component was explained: the threads enable speculation of read values, and the storage subsystem models the propagation of values throughout the processor.



The thread and storage subsystems are labeled transition systems (LTSs). Their composition uses the following set of transitions:

label ::= FETCH *tid ii*
— COMMIT_WRITE_INSTRUCTION *tid ii w*
— COMMIT_BARRIER_INSTRUCTION *tid ii barrier*
— COMMIT_READ_INSTRUCTION *tid ii rr*
— COMMIT_REG_OR_BRANCH_INSTRUCTION *tid ii*
— WRITE_PROPAGATE_TO_THREAD *w tid*
— BARRIER_PROPAGATE_TO_THREAD *barrier tid*
— SATISFY_READ_FROM_STORAGE_SUBSYSTEM *tid ii w*
— SATISFY_READ_BY_WRITE_FORWARDING *tid* ii_1 *w* ii_2
— ACKNOWLEDGE_SYNC *barrier*
— PARTIAL_COHERENCE_COMMIT w_1 w_2
— REGISTER_READ_PREV *tid* ii_1 *reg* ii_2
— REGISTER_READ_INITIAL *tid ii reg*
— PARTIAL_EVALUATE *tid ii*

Here *tid* ranges over thread ids, *ii* over instruction instances, *w* over write events, *rr* over read-response events, *barrier* over `sync` or `lwsync` events, and *reg* over Power register names. A Power execution t is simply a trace of abstract-machine states and these labels.

The proof will involve witnessing a C/C++11 execution that is observationally equivalent to a Power execution, and involves mapping between the two sorts of execution. The mapping between traces and pre-executions is relatively straightforward: the Power model has notions of program order, data dependency and address dependency, all of which trivially map to the C/C++11 counterparts: sequenced before and data dependence. Observational equivalence is slightly more subtle because in the Power model, reads and writes are not modeled by a single transition in an execution. We identify the COMMIT_WRITE_INSTRUCTION and COMMIT_READ_INSTRUCTION transitions with C/C++11 writes and reads. For observational equivalence, we require the C/C++11 reads-from map to precisely correspond to the Power reads from map over Power events.

We will restrict the compiler in the statement of the proof of correctness of the mapping, requiring every Power trace of the compiled program to have a corresponding C/C++11 pre-execution and reads-from map such that the pre-execution is admitted by the thread local semantics, the Power trace corresponds to the application of the mapping to the pre-execution, and the reads-from relation is observationally equivalent to that of the Power trace.

Now we can state the theorem:

THEOREM 16. *Let p be a C++ program that has no undefined behaviour. Suppose comp*

is a compiler that satisfies the constraints imposed above. Let p' be the Power program produced by compiling p with comp. Then for any Power execution of p', t, there exists a pre-execution, X_{pre}, and execution witness, X_{witness}, such that X_{pre} is a pre-execution of p according to the thread-local semantics, X_{pre} and X_{witness} form a consistent execution, and X_{pre} and X_{witness} are observably equivalent to t as defined above.

Proof. The proof of correctness involves using the assumptions on the compiler to find a pre-execution X_{pre} and reads-from map that agree with the thread-local semantics and are observationally equivalent to t. Modification order can be calculated from Power coherence order: simply take the final set of coherence constraints in t, linearise them over the actions at each location, and then project out the part of the relation that covers atomic locations. The calculation of SC order is an arbitrary linearisation of the following relation:

$$(po_t^{sc} \cup co_t^{sc} \cup fr_t^{sc} \cup erf_t^{sc})^*$$

Here, po_t^{sc} is the projection of program order to events arising from SC actions, and co_t^{sc}, fr_t^{sc} and erf_t^{sc} are similar restrictions of coherence order, from-reads, and reads-from across two different threads respectively, where from-reads relates reads to the coherence successors of the write that they read.

Now take the C/C++11 execution comprising X_{pre} and an execution witness X_{witness}, made up of the reads-from, modification order and SC orders identified above. We need to show that this pair form a consistent execution, or there is some execution of the program that has a race.

The proof proceeds by considering each of the conjuncts of the consistency predicate in turn. Consistency is dependent on the happens-before relation, and linking patterns of instructions in the Power trace to happens-before edges in the C/C++11 execution will be essential.

To that end, we define a new relation that identifies inter-thread edges in the Power trace that induce happens-before edges in the C/C++11 execution. In C/C++11, a happens-before edge is created from the head of a release sequence to any acquire or consume read that reads from the sequence. Referring to the mapping, a release write is preceded in the Power trace by either a `sync` or `lwsync` and (ignoring for now read-modify-writes) the unbroken chain of coherence-order successors on the same thread form the release sequence. The read must be either an acquire or a consume, so there is either a dependency or a control-isync following the read. Writing this relation down, using semicolon for forward composition of relations, gives $machine\text{-}ithb_t$:

$$\left((sync_t \cup lwsync_t)^{\text{refl}}; coi_t{}^*; rfe_t; (ctrlisync_t \cup dd_t{}^*)^{\text{refl}}\right)^+$$

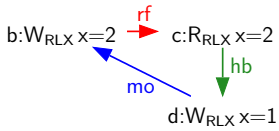
Here, coi_t is the set of coherence edges restricted to pairs on the same thread, and rfe_t is the reads from relation restricted to events on different threads. With this definition,

we know that any inter-thread happens-before edge across two different threads implies the existence of a *machine-ithb*$_t$ edge between the corresponding Power events.

In the Power trace, the values read and the construction of the coherence order depend on propagation order. We define a relation *prop-before* that captures propagation order, and prove (see [26] for the details) that a *machine-ithb*$_t$ edge implies ordering by prop-before.

Each of the conjuncts of the consistency predicate can now be considered in turn. Most of the conjuncts are elided here, but we describe the proofs of two representative cases: the CoRW case of coherent-memory-use and consistent-non-atomic-rf.

CoRW For the CoRW case of coherent-memory-use, we must prove the absence of the following shape in the C/C++11 execution:



First suppose the shape exists, seeking a contradiction. Note that between the corresponding events of the Power trace, there is a Power reads-from edge matching the C/C++11 reads from edge and there is either a Power program-order or *machine-ithb*$_t$ edge corresponding to the happens-before edge. Given these edges, we know that the write on the left-hand thread is propagated to the right-hand thread before the read is committed. We also know that the program-order or *machine-ithb*$_t$ edge implies that the read is propagation-before the write on the right-hand thread, so we have that the left-hand write is coherence ordered before the right-hand write. Then appealing to the construction of the execution witness, modification order contradicts this choice of coherence order.

Consistent-non-atomic-rf This conjunct of the consistency predicate requires reads at non-atomic locations to read from one of their visible side effects. Seeking a contradiction, assume the contrary. The cases where the read reads a happens-before later or happens-before hidden write are elided. We consider the case where it reads from a happens-before-unrelated write.

Such an execution is not consistent according to the C/C++11 memory model, but we can show that in this case, there is a consistent execution of the original program that results in undefined behaviour.

First, identify the earliest read in Power trace order that reads from a happens-before unrelated write. Identify all trace-order predecessors of the read as a prefix trace. Now add in the read, altering its reads-from edge to read from a visible side effect, if one exists, or any write if not. In either case, the read now reads from a write that is consistent according to the C++11 memory model. In the first case, the new reads from

edge relates a write that races with the original write, and in the second case we have introduced an indeterminate read — both result in undefined behaviour.

The speculation of reads in the Power model means that the trace-order prefix may leave out some program-order preceding events. In order for speculation to take place, these events cannot correspond to SC or acquire actions, so we are free to add them back in to the prefix. Finally, complete this prefix in such a way that it forms a consistent C/C++11 execution, and we have the faulting execution of the program we required.

□

Chapter 8

Library abstraction

This chapter presents joint work with Mike Dodds and Alexey Gotsman.

The C/C++11 memory model is intricate, and it is difficult to reason about even small programs. If programmers are to use C/C++11, it must be possible to encapsulate and abstract parts of the program. This chapter explores how this might be accomplished, introducing tools for concurrent compositional reasoning. We use these tools to specify a concurrent data structure, the Treiber stack, and prove a implementation correct.

Java's `java.util.concurrent` library provides concurrent data structures whose interfaces are simple, yet whose implementations provide high performance. If such a library were constructed over C/C++11, there would be several choices of how to specify its interface. The simplest approach might be to ask the programmer to adhere to a set of rules (using a subset of the language, avoiding data races, and so on), provide a high level specification, and promise sequential consistency. This approach comes with a performance penalty: for many concurrent programming idioms, SC behaviour is not essential, but the cost of achieving it is high. A concurrent flag can be compiled very cheaply to x86, for instance, but if it must have an SC interface, then one must add an expensive `MFENCE` barrier. In many cases, the relaxed behaviour may be tolerable, and the performance degradation unacceptable.

Suppose then, that we would like to provide a relaxed library specification to the programmer. This specification has to capture more information than an SC specification: we need to know not just what data will be returned for a given input, but also the relaxed-memory behaviour of the library: what synchronisation it relies on and what it provides.

There are many ways to write such a specification. One might define a sequential specification over calls to the library and describe synchronisation between calls separately. We could describe the specification as an augmentation of the memory model that adds new rules and relations that govern library calls. In this work we choose to represent the library specification as a program, albeit one that executes in an extended version of the C/C++11 memory model. The atomics allow us to conveniently express relaxed memory

behaviour in the specification.

Consider a program composed of a client that makes calls to a library, and suppose we have two versions of the library code: a specification and an implementation. The behaviour of the client composed with either version of the library is defined by the set of executions allowed by the memory model. In each execution, some of the memory accesses will arise from the client, and some from the library. We define an interface between the two by inserting new actions at the calls and returns of library functions. We consider only libraries and clients that access disjoint sets of locations, so there are no modification-order or reads-from edges between the library and client actions, except for those that correspond to parameter passing and returning a value.

Now we can define observational refinement of libraries from the perspective of a client. First we mask the library actions of each execution, preserving just the client and interface actions. The set of all masked executions allowed by the memory model for a given client is the client behaviour. Our specifications are programs, so we can execute clients that call them. If a every client behaviour when composed with the implementation is admitted by the client composed with the specification, then the implementation refines the specification.

In this chapter, we provide an abstraction relation that holds between libraries only when one refines another. To do this, we precisely capture the interaction of a library with its client context in a *history*, we define an abstraction relation over sets of histories. We then prove that if a library implementation's histories are abstracted by a specification's histories then the behaviour of the implementation in an arbitrary client context is a subset of the behaviour of the specification in the same context. We use this abstraction theorem to establish the correctness of a Treiber stack.

The formulation of the theorem is less than ideal, in that it fails to support free composition of programs. The reasons for this weakness lie in the lack of a restriction on out-of-thin-air values, and in particular self-satisfying conditionals. These limitations are discussed together with a sketch of the proof of soundness of the abstraction relation.

The work in this chapter covers C/C++11 programs that do not use consume atomics. The work uses a version of the memory model that is expressed rather differently to the models of Chapter 3. The model can be found in the paper [25], and is not included here. Instead, in this chapter we refer to the *sc_fenced_memory_model* that covers the same subset of the language features and is intended to match, although we have not formally established equivalence.

8.1 A motivating example — the Treiber stack

This chapter is guided by an example data structure, the Treiber stack [108]: a non-blocking concurrent stack with `push` and `pop` methods. Our C/C++11 implementation

```
struct Node {
  int data;
  Node *next;
};
atomic Node *T;

void push(int v) {
  Node *x, *t;
  x = new Node();
  x->data = v;
  do {
    t = load_RLX(&T);
    x->next = t;
  } while
    (!CAS_RLX,REL(&T,t,x));
}
```

```
void init() {
  store_REL(&T,NULL);
}

int pop() {
  Node *t, *x;
  do {
    t = load_ACQ(&T);
    if (t == NULL)
      return EMPTY;
    x = t->next;
  } while
    (!CAS_RLX,RLX(&T,t,x));
  return t->data;
}
```

Figure 8.1: The Treiber stack implementation. For simplicity, we let `pop` leak memory.

of the Treiber stack, which ignores deallocation, is presented in Figure 8.1.

The stack is represented in the implementation as a linked list of nodes, accessed through a top pointer, `T`. Note that the variables in the nodes are not atomic, but `T` is. This reflects the fact that calls that access the data structure contend on the pointer rather than the node data. The initialisation call stores a null value to `T` with release memory order. The `push` call creates a new node, writes data to it, and then tries to insert the node at the top of the stack. Insertion involves loading `T`, writing its value to the next pointer in the node, and then performing a compare-and-swap that will atomically release-write `T` with the address of the new node if `T` still has the same value as was previously loaded. If the compare-and-swap fails, then there is no write of `T`, and the load-CAS loop repeats. The `pop` call features a similar loop, but here `T` is loaded initially with the acquire memory order, and the CAS has relaxed memory order, even on success. In the loop, `pop` checks whether `T` is null, returning empty if it is. Otherwise it reads the node pointed to by `T`, and attempts to CAS that node's next pointer into `T`. If the CAS succeeds, `pop` returns the data of the node, and if not it repeats the loop.

To help abstract the load-CAS-loop programming idiom, used in the push and pop calls of the implementation of the Treiber stack, we introduce a new construct: the *atomic section*. Atomic sections ensure that other actions in the execution cannot be ordered between those in the atomic section. The rules that support atomic sections restrict happens-before, modification order and SC order, and are given the paper [25].

The specification of the Treiber stack is presented in Figure 8.2. It abstracts several details of the implementation. Firstly, the layout of data in memory is no longer explicit; rather than a linked list of nodes, the specification represents the data as an abstract

```
atomic Seq S;

void push(int v) {
  Seq s, s2;
  if (nondet()) while(1);
  atom_sec {
    s = load_RLX(&S);
    s2 = append(s,v);
    CAS_RLX,REL(&S,s,s2);
  }
}

void init() {
  store_REL(&S,empty);
}

int pop() {
  Seq s;
  if (nondet()) while(1);
  atom_sec {
    s = load_ACQ(&S);
    if (s == empty)
      return EMPTY;
    CAS_RLX,RLX(&S,s,tail(s));
    return head(s);
  }
}
```

Figure 8.2: The Treiber stack specification.

sequence, `S`, of values. Three mathematical functions take a sequence as an argument: `append` returns a new sequence with the value passed as an argument appended to it, `head` returns the head of the sequence, and `tail` returns the tail. Each of the library calls produces a locally modified version of the sequence and then atomically overwrites the sequence `S`. These atomic accesses of the sequence encode the synchronisation that is guaranteed by the specification with the choices of their memory-order arguments (this is discussed in detail in the next section). Note the release memory order of the CAS in the push call and the acquire-load in pop. The atomic section is present to help to abstract the load-CAS-loop idiom. The atomic section ensures that there can be no write that intervenes between the load at the start and the CAS at the end of each section. This means that the CAS in each section will always succeed. We cannot replace it with a release store in the specification, because we are using the additional synchronisation through the release sequence. In the implementation, the load-CAS loop in either push or pop may be starved indefinitely, leading to divergence. This is modeled in the specification with nondeterministic divergence.

Note that the implementation does not behave in a straightforward sequentially consistent manner, even when we consider only the values returned by `pop`. Consider the following program that uses two instances of the Treiber stack, `A` and `B`:

Store Buffering (MP):

`A.push(1);`	`B.push(1);`
`r1 = B.pop();`	`r2 = A.pop();`

Both `A` and `B` are initially empty. Each thread pushes an item on to one of the two stacks and then pops from the other, storing the result into a thread-local variable. If

we took a naive specification of the stack in an interleaved memory model, this program would never produce outcomes where both the pop of `A` and the pop of `B` return empty, but the relaxed implementation allows this behaviour. This is the analogue of store-buffering relaxed behaviour, observable at the interface to the Treiber stack. To forbid this non-SC behaviour, additional synchronisation would be required in the implementation, reducing performance. It is therefore desirable to be able to express relaxed specifications of the data structure. In doing so, we admit implementations with higher performance.

We will show that the implementation does meet the specification by introducing an abstraction relation, proving it sound, and applying it to the Treiber stack. We first motivate and define some of the concepts that the abstraction theorem relies on.

8.2 Defining library abstraction

This section introduces an abstraction relation that relates one piece of code to another. Stating the definition of the abstraction relation requires first presenting a series of other definitions on which it is based. The first captures the behaviour of a piece of code at its interface.

8.2.1 Motivating the definition of a history

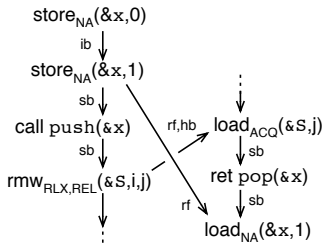
We would like to precisely capture the interaction of a library with its client context. To motivate our definition, we explore the behaviour of the implementation of the Treiber stack. We first augment the thread-local semantics so that it recognises each thread's entry and exit from library functions. We have it mark the boundaries between the library and the client in executions with new actions, `call` and `return`. Each carries the values that are passed to and returned from the function, respectively. With these new actions, we will explore several examples in order to understand how the execution of the Treiber stack impacts an arbitrary client context.

Data consistency To be useful, the specification will have to restrict the values that can be returned by `pop`, so that the data structure behaves as a stack. As noted in the previous section, specifications may admit values that result from relaxed executions.

Synchronisation The memory orders in the implementation presented above are carefully chosen to create synchronisation from code that precedes a call to `push` to code that follows the call to `pop` that returns the `push`'s data. This relies on release-acquire synchronisation to guarantee that the message-passing programming idiom is supported. Consider the following program that uses the Treiber stack, on the left below:

MESSAGE PASSING (MP):

```
int a, b, x=0;

x=1;          ||  do {a=pop();}
push(&x);     ||  while
              ||    (a==EMPTY);
              ||  b=*a;
```



On the left-hand thread, the program writes to `x` and then pushes the address of `x` onto the stack. The right-hand thread repeatedly pops the stack, waiting for data. When data arrives, the thread reads from the address returned by the pop. This program relies on the stack's internal synchronisation: there must be a happens-before edge between the store and the load of `x`, or the program has a race.

In the example execution above, the entry to the push call and return from the pop call are identified with the new abstract actions, and all but the read-modify-write action in the push, and the load acquire action in the pop are elided. The right-hand thread pops the data pushed by the left-hand thread. The successful CAS in the push call results in a read-modify-write action with release memory order, and the successful pop loads with acquire memory order. These two actions synchronise, and the transitivity of happens-before through sequenced-before and synchronises-with means that the store of `x` happens before the load of `x`, avoiding a data race.

The specification of relaxed data structures will have to describe the synchronisation that the client can rely on. If synchronisation had not been provided by the implementation in the case above, the client would have had a race, and the program would have undefined behaviour. We define a new *guarantee* relation, G, that captures library-induced synchronisation. More precisely, G is the projection of happens-before created by internal synchronisation of the library to the call and return actions. Returning to the message-passing example above, we see that the internal synchronisation creates a guarantee edge in the execution from the call action to the return action.

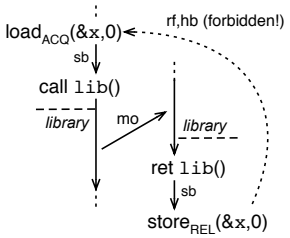
Ordering without synchronisation For simplicity, we consider library code that only interacts with its client through call parameters and return values; i.e. the memory footprint of the library and its client are disjoint. One might expect this assumption to reduce the effect of the library on the client to the return values together with any synchronisation that the library creates between library calls, but this is not the case.

In C/C++11, the library interacts with its client in another more subtle way. In an execution, the actions that result from library calls might be related by reads-from, lock-order, modification-order or SC-order edges. There are rules in the model that restrict how these edges can be combined with happens-before in a consistent execution. For example, recall that the coherence requirements forbid modification-order edges from

opposing happens-before edges. In the program below, the calls to `lib` each contain a write to the same location that must be ordered by modification order. Consider the execution where the modification-order edge points from left to right, depicted on the right below. If the load of `x` in the client context were to read from the store of `x`, then the actions would synchronise, and the transitivity of sequenced-before and synchronises-with would create a happens-before edge opposing the modification order edge. The two edges together violate the CoWW coherence axiom, so this is not a consistent behaviour of the program, and will not be observed.

DENY (DN):

```
      store_REL(&x,1);

load_ACQ(&x); || lib();
lib();        || store_REL(&x,0);
```



We will collect all ordering of this sort into a new relation: the *deny* relation, D, is defined as the set of edges from call to return actions where the addition of a client happens-before edge from return to call would complete a shape that is forbidden by the consistency predicate. In the example above, there is a deny edge from the call to the return: an opposing happens-before edge would violate CoWW. Each rule in the consistency predicate that restricts happens-before with reference to another relation contributes to the deny relation. As a consequence library-internal reads-from, modification-order, lock-order and SC-order edges can all create deny edges.

Deny ordering is weaker than guarantee ordering: the guarantee assures us of the existence of a happens-before edge, there may not be a client happens-before edge that opposes a deny, but there may be one coincident with it.

History Having defined the interface actions, guarantee edges and deny edges, we can now define the *history* of an execution, that identifies the interface between the client part of an execution and the library part:

DEFINITION 17. *The **history** of an execution X is a triple $H = (A_i, G, D)$, where A_i is a set of call and return actions, G is the guarantee relation, D is the deny relation and $G, D \subseteq A_i \times A_i$.*

8.2.2 The most general client

If two library implementations produce the same set of histories, then there is no difference in how they affect the client, and the client behaviour of their composition will be identical. If one implementation produces a subset of the histories of the other then its behaviours

in composition with a client will be a subset of that of the other. From this observation, we define a sound abstraction relation over histories that we can then lift to the sets of histories generated by library code.

With this formulation of abstraction, a specification is simply a collection of histories. Here, our implementation and specification are both programs, and we will enumerate the histories of each by executing them in an arbitrary client context. This motivates the definition of the *most general client*: rather than enumerate the behaviour of the library in an arbitrary client context, we would like a constrained set of client contexts that are sufficient to generate all possible histories. The most general client must enumerate all possible combinations of library calls, on any number of threads, with all possible values of arguments. The definition of the most general client is:

DEFINITION 18. *The* ***most general client*** *is defined as follows: Take* $n \geq 1$ *and let* $\{m_1, \ldots, m_l\}$ *be the methods implemented by a library* $\mathcal{L}$*. We let*

$$\mathsf{MGC}_n(\mathcal{L}) = (\mathsf{let}\ \mathcal{L}\ \mathsf{in}\ C_1^{\mathsf{mgc}} \parallel \ldots \parallel C_n^{\mathsf{mgc}}),$$

where C_t^{mgc} *is*

$$\mathsf{while}(\mathsf{nondet}())\ \{\ \mathsf{if}(\mathsf{nondet}())\ \{m_1\}\mathsf{else}\ \mathsf{if}(\mathsf{nondet}())\ \{m_2\}\ \ldots\ \mathsf{else}\ \{m_l\}\ \}$$

Here, we let the parameters of library methods be chosen arbitrarily.

We are considering a restricted set of programs where all library locations are initialised with writes that happen before all other memory accesses. We write $[\![\mathcal{L}]\!]I$ for the set of executions of the library $\mathcal{L}$ under the most general client starting from an ***initial state*** I.

To cover the set of all possible consistent executions, we must also enumerate all possible client happens-before edges: the presence of a client happens-before edge does not simply restrict the set of consistent executions; it can introduce new ones, by creating a new visible-side-effect for a non-atomic read, allowing a new value to be read, for example. We define the **extension** of an execution X with the relation R as an execution with identical components whose happens-before relation is transitively extended with R. Now we can extend the execution of the most general client with an arbitrary set of client happens-before edges so that we capture all possible behaviours of the library. We write $[\![\mathcal{L}, R]\!]I$ for the set of consistent executions of $\mathcal{L}$ from I extended with R.

In Lemma 22 we establish the following: all library projections of an execution of a client and a library are contained in the execution of the library under the most general client, extended with client happens-before edges. This shows that the MGC can reproduce the behaviour of the library under any client, and that it is, in fact, most general.

8.2.3 The abstraction relation

The history of a library, as identified by the extended most general client, identifies all of the possible behaviours of a piece of library code in an arbitrary client context. We define abstraction in terms of these histories.

First note that one history can impose a stronger restriction on its client than another. We have noted that adding or removing happens-before edges can add or remove behaviours to the execution of the history in a context, but adding deny edges only ever removes behaviours. As a consequence, a history that contains more deny edges permits fewer executions in a particular context. We define abstraction over histories in these terms:

DEFINITION 19. *For histories* (A_1, G_1, D_1) *and* (A_2, G_2, D_2)*, we let* $(A_1, G_1, D_1) \sqsubseteq (A_2, G_2, D_2)$ *if* $A_1 = A_2$*,* $G_1 = G_2$ *and* $D_2 \subseteq D_1$*.*

Now we raise the abstraction relation to a relation over pieces of code using the most general client. Recall that a piece of code can exhibit a consistent execution with a data race, and in that case the whole program has undefined behaviour. We would like to show the soundness of the abstraction relation, and this will not hold in the presence of undefined behaviour. We define a library and set of initial states, $(\mathcal{L}, \mathcal{I})$, as safe if it does not access locations internal to the client, and it does not have any executions under the most general client with faults like data races. We can now raise the definition of abstraction to the level of library code:

DEFINITION 20. *For safe* $(\mathcal{L}_1, \mathcal{I}_1)$ *and* $(\mathcal{L}_2, \mathcal{I}_2)$*,* $(\mathcal{L}_1, \mathcal{I}_1)$ ***is abstracted by*** $(\mathcal{L}_2, \mathcal{I}_2)$*, written* $(\mathcal{L}_1, \mathcal{I}_1) \sqsubseteq (\mathcal{L}_2, \mathcal{I}_2)$*, if for any relation* R *containing only edges from return actions to call actions, we have*

$$\forall I_1 \in \mathcal{I}_1, H_1 \in \mathsf{history}([\![\mathcal{L}_1, R]\!] I_1). \quad \exists I_2 \in \mathcal{I}_2, H_2 \in \mathsf{history}([\![\mathcal{L}_2, R]\!] I_2). \quad H_1 \sqsubseteq H_2.$$

The formulation of the abstraction relation is quantified over all possible happens-before extensions to the most general-client. This enumeration makes part of the history redundant. Deny edges that result from library-internal modification order, lock order and reads-from edges no longer need to be tracked. This is because, in safe programs, the enumeration over happens-before extensions will remove histories where client happens-before edges together with internal mo, lo or rf edges violate the consistency predicate. The only relation that the deny needs to track is SC order, which spans both client and library parts of the execution and must be acyclic.

8.2.4 The abstraction theorem

Suppose we have a library specification that abstracts a library implementation according to the definition above, then we would like to know that the behaviour of that implemen-

tation in an arbitrary client context is a subset of the behaviour of the specification in the same context. This is the guarantee that the abstraction theorem provides, with some caveats.

In order to apply the theorem, we need to establish some properties of the library and the client. We need to know that there are no program faults that lead to undefined behaviour in either, but more problematically, we need to know that the library and the client only communicate through calls and returns to library functions, and that they do not write to each others internal memory locations. Collectively, we call these properties *safety*. Ideally, we would establish safety of the library in an arbitrary client context, and prove that this implies safety of any implementation that the specification abstracts, but this is not the case.

Consider the following specification, $\mathcal{L}_2$, and implementation, $\mathcal{L}_1$, of a library that contains a single method `m`. Internally, the library uses a location, `x`, that it never accesses in the specification, but is written to and read from in the implementation.

$\mathcal{L}_1$:
```
atomic int x;
int m() {
  store_RLX(&x,42);
  return load_RLX(&x);
}
```

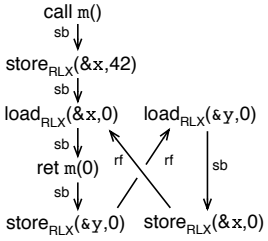
$\mathcal{L}_2$:
```
atomic int x;
int m() {
  return 42;
}
```

For client contexts that do not access `x`, the specification and implementation behave in precisely the same way, and we have $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$. An unsafe client can of course distinguish between the two. Take for example the following:

```
print m(); || store_RLX(&x,0);
```

Any library will behave in unexpected ways if its client context corrupts its internal data structures, so we restrict our abstraction theorem to clients that do not do this. The program above violates this restriction when composed with the specification of the library: it accesses `x`. Now consider the following program, where the location `y` is initially non-zero:

```
a=m()                  || b=load_RLX(&y)
if (a==0)              || if (b==0)
 store_RLX(&y,0)       ||  store_RLX(&x,0)
```



Every execution of this program, when composed with the library specification is safe: the client never executes a write to `x`. That is because the library always returns the value 42, so the conditional in the left-hand thread always fails, there is never a write

to y, the conditional in the right-hand thread is never satisfied, and the write it guards never executes.

Now consider the execution of the client composed with the implementation drawn on the right above. This execution features load-buffering style relaxed behaviour where each load reads from a future write on the other thread. In this execution, the library-internal load of x reads from the client's write of x, violating safety. This example shows that for a given client, we cannot establish safety of the implementation as a consequence of safety of the specification.

It is easy to recognise the client as a potentially unsafe one in the example above because syntactically it contains an access of the library-internal variable x. The same sort of behaviour can be observed with a client that acts on an address that is decided dynamically, so that the potential safety violation would not be so easily identified.

In the formulation of the abstraction theorem, this example forces us to require safety of the client composed with the specification as well as non-interference of the client composed with the implementation:

THEOREM 21 (Abstraction). *Assume that* $(\mathcal{L}_1, \mathcal{I}_1)$, $(\mathcal{L}_2, \mathcal{I}_2)$, $(\mathcal{C}(\mathcal{L}_2), \mathcal{I} \uplus \mathcal{I}_2)$ *are safe,* $(\mathcal{C}(\mathcal{L}_1), \mathcal{I} \uplus \mathcal{I}_1)$ *is non-interfering and* $(\mathcal{L}_1, \mathcal{I}_1) \sqsubseteq (\mathcal{L}_2, \mathcal{I}_2)$. *Then* $(\mathcal{C}(\mathcal{L}_1), \mathcal{I} \uplus \mathcal{I}_1)$ *is safe and*

$$\mathsf{client}([\![\mathcal{C}(\mathcal{L}_1)]\!](\mathcal{I} \uplus \mathcal{I}_1)) \subseteq \mathsf{client}([\![\mathcal{C}(\mathcal{L}_2)]\!](\mathcal{I} \uplus \mathcal{I}_2)).$$

The unfortunate requirement for non-interference of the implementation in the abstraction theorem is an important limitation of the C/C++11 memory model design. It means that we cannot alleviate the complexity of the relaxed memory model by encapsulating part of the program in a library and then providing a simple specification. The programmer will always have to establish that their program is non-interfering according to the C/C++11 memory model with the library implementation.

The problematic example was a consequence of the presence of self-satisfying conditional executions: we placed a safety violation in the guarded block on each thread so that the violations only occurred when self satisfying behaviour was witnessed. If the memory model forbade self satisfying conditionals (this is difficult to ensure, see Chapter 5 for details), then the client presented above would be safe. Moreover, we would not need to check that $(\mathcal{C}(\mathcal{L}_1), \mathcal{I} \uplus \mathcal{I}_1)$ is non-interfering because, after restricting dependency cycles, any incidence of interference that remained allowed would be a safety violation of either $(\mathcal{L}_1, \mathcal{I}_1)$ or $(\mathcal{C}(\mathcal{L}_2), \mathcal{I} \uplus \mathcal{I}_2)$. This new restriction in the model would make the language compositional.

8.3 Abstraction in the Treiber stack

This section presents an overview of the proof of safety, non-interference and abstraction for the specification and implementation of the Treiber stack. The full proof can be found in the paper [25]. The proof relies on a derived total order, $<$, over all of the calls to

push and pop in an execution. The order can be constructed over executions that use the specification or the implementation of the stack.

In constructing the order, note that there are three outcomes for the calls to pop and two to push: a pop can fail and return empty and both can succeed or block. We start building up $<$ by recognising that successful calls feature a read-modify-write action on T. For two calls to the library, I_1 and I_2, $I_1 < I_2$ if for the rmw events $a_1 \in I_1$ and $a_2 \in I_2$, $a_1 \xrightarrow{\mathsf{mo}} a_2$. Failing calls of pop acquire-load from a rmw action. We place the pop call into $<$ immediately following the call that gave rise to the rmw action. We include blocking calls anywhere in $<$.

Now we use the total order $<$ to prove safety, non-interference and abstraction by induction.

Safety and non-interference First we establish that any read from T in an implementation pop happens after every $<$-preceding write to T in push. Note that all push calls contain a release read-modify-write, and all pop calls contain an acquire load. Because all writes are read-modify-writes, each push heads a release sequence that contains all $<$-following writes of T, and the pop synchronises with all prior push calls.

Now we show safety of the implementation by considering each of the possible sources of undefined behaviour: unsequenced races, bad mutex use, indeterminate reads, interference and data races. The first three are trivial: there are no unsequenced actions or locks, and we initialise all locations before any calls to the library. We show non-interference by inducting on $<$: at each step, a call only reads or writes library-local locations. The only non-atomic accesses are of the `next` and `val` fields of nodes. First note that nodes are not reused, so we need only consider write-read races. Writes of either location are followed by a release write to T, and reads of either are preceded by an acquire read of T. We know from the argument above that these actions generate synchronisation that means there is no race.

Safety and non-interference of the specification follow a similar argument, although race-freedom is now straightforward because there are no non-atomic accesses.

Abstraction Proving that the specification abstracts the library amounts to witnessing the history of an arbitrary execution of the implementation in the client combined with the specification. To do this, we take an arbitrary execution of the implementation, $X_{\text{imp}} \in [\![L_1, R]\!]\mathcal{I}_1$, under an arbitrary happens-before extension R, construct an execution of the specification with the same history, and then show that this new execution is consistent.

First project out the pre-execution portion of the candidate execution X_{imp}, and call it C_{imp}. Now replace successful push and pop calls in C_{imp} with the actions corresponding to successful specification calls of push and pop with the same values at interface actions, and do the same for failed pops and blocked calls. We use the $<$ relation to construct the

values returned by the specifications internal function calls over the sequence S: following $<$, we build up the internal state of the sequence at each call starting with an empty sequence, and we set the return values of the append, head and tail calls to match. This gives a new pre-execution C_{spec}. It is straightforward that $C_{\text{spec}} \in \langle L_2 \rangle \mathcal{I}_2$.

Now we construct an execution witness for C_{spec}, by setting reads-from and modification order to match $<$. These two components together with the calculated relations produce the candidate execution X_{spec}. We will now show that X_{spec} is a consistent execution.

First we induct along $<$ to establish that the accesses of `next` and `val` in the implementation correspond to the values appended and returned from head and tail in the specification. This follows from the structure of the induction, the fact that successful push and pop calls execute read-modify-writes, and the fact that modification order contributes to $<$.

Most of the conjuncts of the consistency predicate are trivially satisfied, or follow from the push to pop release-sequence synchronisation. The rules for atomic sections are satisfied by the fact that each location is written only once in the atomic block, and both modification order and reads-from match the total order $<$.

History Inclusion It remains to show that the implementation history is abstracted by the specification history:

$$\mathsf{history}(X_{\text{imp}}) \sqsubseteq \mathsf{history}(X_{\text{spec}})$$

There are no SC accesses in either the library or the implementation, so we need not consider the deny portion of the history. To show that the guarantee portion of the history is the same, note that we do not need to consider blocking calls, because they never return and do not add to the guarantee. For successful calls, the same release-sequence reasoning that identifies the implementation synchronisation applies to the specification calls. Failed pop calls that read the initialisation do not create synchronisation in the implementation or the specification. Calls that read from a push do synchronise with $<$-earlier push calls, but the specification also synchronises in this case.

This completes the proof that the Treiber stack specification abstracts the implementation.

8.4 Soundness of the abstraction relation

The proof of soundness relies on two lemmas: one that allow us to decompose programs into client and library parts, and another that allows us to compose compatible libraries and clients. In decomposition, we abstract the effect of the other component into a history, and show that the behaviour of the whole, projected to the chosen component matches the

execution of the component extended with a history. In composition, we show that two components extended with compatible histories can be composed into a whole program whose component projections match the original history-extended executions. We use these two lemmas to show that Theorem 21 holds.

8.4.1 Necessary definitions

The following propositions use several new definitions, introduced here. First we define $\mathsf{interf}(X)$, a function that projects the interface actions. Then we define functions that identify the guarantee and deny of a library in an execution: $\mathsf{hbL}(X)$ is the projection of happens-before created by the library from call to return actions, and $\mathsf{hbC}(X)$ is the client equivalent of $\mathsf{hbL}(X)$, the projection of client happens-before from return to call actions.

We simplify the deny relation, removing edges that would be made inconsistent by modification order or reads-from. We can do this because non-interference implies that these edges only relate actions of the same component, and our calculation of the history enumerates all possible extensions of happens-before. Therefore, any choice of extension that makes an execution inconsistent must also make a the corresponding execution inconsistent in any component history that abstracts it. The SC relation, on the other hand, relates actions from both components, and the composite SC order must agree with happens before. The remaining deny edges are captured by $\mathsf{scL}(X)$, the projection of $((\mathsf{hb}(X) \cup \mathsf{sc}(X))^+)^{-1}$ to return to call actions. There is again a client equivalent of $\mathsf{scL}(X)$, $\mathsf{scC}(X)$, the projection of $((\mathsf{hb}(X) \cup \mathsf{sc}(X))^+)^{-1}$ to call and return actions.

We define projection functions $\mathsf{lib}(X)$ and $\mathsf{client}(X)$ that project out from execution X all actions (and the relations over them) in the interface together with those from either the library or the client respectively. For an extended execution X, $\mathsf{core}(X)$ is the same execution with happens-before recalculated without the extension.

8.4.2 Decomposition and composition

Now the decomposition and composition lemmas are stated along with sketches of their proofs (see the paper for the full proofs [25]).

LEMMA 22 (Decomposition). *For any* $X \in [\![\mathcal{C}(\mathcal{L}_1)]\!](I \uplus I_1)$ *satisfying* NONINTERF,

$$\mathsf{client}(X) \in [\![\mathcal{C}, \mathsf{hbL}(\mathsf{core}(\mathsf{lib}(X)))]\!]I; \tag{8.1}$$

$$\mathsf{lib}(X) \in [\![\mathcal{L}_1, \mathsf{hbC}(\mathsf{core}(\mathsf{client}(X)))]\!]I_1. \tag{8.2}$$

Furthermore,

- $\mathsf{client}(X)$ *and* $\mathsf{lib}(X)$ *satisfy* NONINTERF*;*
- *if* X *is unsafe, then so is either* $\mathsf{client}(X)$ *or* $\mathsf{lib}(X)$*; and*

- $\mathsf{scC}(\mathsf{client}(X)) \cup \mathsf{scL}(\mathsf{lib}(X))$ *is acyclic.*

Proof Sketch It is straightforward that $\mathsf{client}(X)$ and $\mathsf{lib}(X)$ satisfy NONINTERF. Furthermore, any cycle in $\mathsf{scC}(\mathsf{client}(X)) \cup \mathsf{scL}(\mathsf{lib}(X))$ is also a cycle the original execution, and this contradicts the consistency of SC.

To check 8.1 (8.2 is similar), we need to show that the client projection of the happens-before edges in X match the happens-before relation created by the most general-client execution of the client extended by the library core guarantee. Note that the MGC happens-before is certainly a subset, so it remains to show that any happens-before edge in $\mathsf{client}(X)$ is also an edge in an execution of the most general client. Consider an arbitrary edge between actions u and v. The happens-before edge is made up of a path in sb and sw. All interface actions on the same thread are related by sequenced before, so we can pick out all interface actions in this path. Synchronisation is only ever created between actions at the same location, so non-interference implies that there is never a synchronises-with edge between the two components. Together that implies that any segment of library actions in the path of edges from u to v starts with a call action and ends with a return action, and is covered by the history, as required.

It is clear that the most general client will generate pre-executions that cover the projections in each case above. Any subset of the actions of an execution together with the relations restricted to that set will satisfy most of the conjuncts of the consistency predicate, with those conjuncts that deal with locks, and read values implied by non-interference.

Because X satisfies non-interference, any safety violation must either be a data race, an indeterminate read or an instance of bad mutex use. In the racy case, again because of non-interference, both racy actions must be in the same component, as required, The mutex case is similar. Indeterminate reads are a single action fault, so they reside in one component or the other.

LEMMA 23 (Composition). *Consider*

$$X \in [\![\mathcal{C}, \mathsf{hbL}(\mathsf{core}(Y))]\!] I;$$
$$Y \in [\![\mathcal{L}_2, \mathsf{hbC}(\mathsf{core}(X))]\!] I_2,$$

such that

- $A(X) \cap A(Y) = \mathsf{interf}(X) = \mathsf{interf}(Y)$ *and otherwise action and atomic section identifiers in* $A(X)$ *and* $A(Y)$ *are different;*
- $\mathsf{interf}(\mathsf{sb}(X)) = \mathsf{interf}(\mathsf{sb}(Y))$*;*
- $\mathsf{scC}(X) \cup \mathsf{scL}(Y)$ *is acyclic; and*
- X *and* Y *satisfy* NONINTERF.

Then for some $Z \in [\![\mathcal{C}(\mathcal{L}_2)]\!](I \uplus I_2)$ *we have* $X = \mathsf{client}(Z)$ *and* $Y = \mathsf{lib}(Z)$. *Furthermore, if* X *is unsafe, then so is* Z.

Proof Sketch We construct a Z that satisfies the conditions. Let the actions and modification order of Z be the union of those of X and Y, let the sequenced before relation be the transitive closure of the union of the sequenced-before relations of X and Y, and let asw extend from all initialisation actions of either to the first action on every thread. From the assumptions, we have that the interface actions and sequenced before relations match over the interface actions, the pre-execution of Z is well-formed, sequenced-before is consistent and the modification order is consistent. Let the reads-from map of Z be the union of the reads-from maps of X and Y extended with edges that correspond to the passing of parameters in call actions and the return of values in return actions.

In order to construct the SC order of Z, we show that the union of the SC relations of X and Y, together with the happens-before relation of Z is acyclic. Assume there is a cycle and (appealing to the argument made in Lemma 22) note that the cycle must be made up of alternating paths of edges in the sequenced-before, synchronises-with and SC relations of either component. The cycle must contain at least one pair of interface actions: otherwise, the path is contained in one component, and violates consistency under the MGC. Then each segment between interface actions corresponds to an $(\mathsf{scC}(X))^{-1}$ or $(\mathsf{scL}(Y))^{-1}$ edge, contradicting the assumptions of the lemma. We arbitrarily linearise the projection to SC actions of the union of the SC relations of X and Y together with the happens-before relation of Z to get the SC order of Z.

We have that $\mathsf{client}(Z) = X$ and $\mathsf{lib}(Z) = Y$ by construction. We have shown several conjuncts of the consistency predicate hold over Z, and the remaining ones follow from the construction of Z and non-interference of X and Y.

X satisfies non-interference, so any safety violation must either be a data race, an indeterminate read or an instance of bad mutex use. Any fault of this sort is also a fault of Z because of non-interference, and because $\mathsf{client}(Z) = X$ and $\mathsf{lib}(Z) = Y$.

8.4.3 Proof of soundness of the abstraction relation: Theorem 21

Consider $X \in [\![\mathcal{C}(\mathcal{L}_1)]\!](I \uplus I_1)$ for $I \in \mathcal{I}$ and $I_1 \in \mathcal{I}_1$. Since X satisfies NONINTERF, we can apply Lemma 22, decomposing X into a client execution $\mathsf{client}(X)$ and a library execution $\mathsf{lib}(X)$:

$$\mathsf{client}(X) \in [\![\mathcal{C},\ \mathsf{hbL}(\mathsf{core}(\mathsf{lib}(X)))]\!]I;$$
$$\mathsf{lib}(X) \in [\![\mathcal{L}_1,\ \mathsf{hbC}(\mathsf{core}(\mathsf{client}(X)))]\!]I_1.$$

Let R be the projection of $\mathsf{hbC}(\mathsf{core}(\mathsf{client}(X)))$ to return and call actions. Then noting that synchronisation is inter-thread and actions from library calls on the same thread are totally ordered by sequenced before, we have that $\mathsf{hbC}(\mathsf{core}(\mathsf{client}(X))) \setminus R$ is a

subset of the sequenced-before relations of either $\mathsf{interf}(\mathsf{client}(X))$ or $\mathsf{interf}(\mathsf{lib}(X))$. Thus, from (8.2), we have $\mathsf{lib}(X) \in [\![\mathcal{L}_1,\, R]\!] I_1$.

Recall that $(\mathcal{L}_1, \mathcal{I}_1) \sqsubseteq (\mathcal{L}_2, \mathcal{I}_2)$. We appeal to Definition 20, with the happens-before extension $\mathsf{hbC}(\mathsf{core}(\mathsf{client}(X)))$ to establish that there exist $I_2 \in \mathcal{I}_2$ and $Y \in [\![\mathcal{L}_2, \mathsf{hbC}(\mathsf{core}(\mathsf{client}(X)))]\!] I_2$ such that $\mathsf{history}(\mathsf{lib}(X)) \sqsubseteq \mathsf{history}(Y)$.

Then from (8.1), the safety conditions and by Lemma 23 we get that for some $Z \in [\![\mathcal{C}(\mathcal{L}_2)]\!](I \uplus I_2)$ we have $\mathsf{client}(Z) = \mathsf{client}(X)$. Furthermore, if X is unsafe, then so is $\mathsf{client}(X)$, and by Lemma 23, so is Z. Thus the safety of $\mathcal{C}(\mathcal{L}_2)$ implies that of $\mathcal{C}(\mathcal{L}_1)$.

Chapter 9

Conclusion

From the IBM 370/158MP of 1972, to the computers and mobile phones of today, relaxed shared-memory concurrency is well established as a common design element. A typical machine now includes a multi-core processor with a relaxed interface to memory, together with a GPU that exhibits massive parallelism, and a hierarchy of memory-address spaces. Over time, advances in mainstream hardware and compilers have continuously outstripped our theoretical understanding of computers.

Typically, the systems software of these machines is programmed with some flavour of the C language. One of the key bottlenecks on these systems is memory, where there is a tradeoff between performance and programmer intuition. Expert programmers, like McKenney [79] and Michael [81], write elaborate algorithms that rely on delicate ordering properties in order to avoid a performance penalty on concurrent memory accesses. These algorithms are at the core of operating systems.

Reasoning about concurrent code written for these systems is difficult for several reasons. These machines exhibit unintuitive, unexpected, and sometimes undocumented behaviour. The specifications of machines and programming languages tend to be poor: these documents are universally written in English prose and are therefore untestable, open to interpretation, and often contain errors and omissions. Moreover, it is not clear what the specification of concurrent algorithms should be: there is not an accepted specification language for relaxed-concurrent algorithms.

Without continued effort, this unhappy situation is sure to get worse: hardware is slated to become more parallel, and memory systems more complex. My goal is to formally understand and improve the programming model of current and future systems in the context of both the underlying hardware, and concurrent programming idioms. At the core of my approach is mechanised formal specification. Formal specifications are unambiguous and can be scrutinised precisely. Mechanised specifications can be executed: one can use them to build tools that test the specification. Mechanised formal specifications also enable one to prove properties of the specification within a theorem prover, adding confidence to results that speak about unwieldy mathematical objects. With these tools,

it is possible to identify errant details of a specification and to propose improvements.

In this thesis, I took a mainstream relaxed memory model and subjected it to a comprehensive technical analysis of the sort described above. The work led to changes in the C and C++ language definitions, and my observations will inform the design of future memory models.

Chapter 3 described a mechanised formal model of C/C++11 concurrency that closely follows the published prose specification. Chapter 4 described CPPMEM, a tool for exhaustively executing very small programs according to the memory model. CPPMEM is used both in teaching, and in industry for understanding the memory model. Chapter 5 described problems found with the standard during the process of formalisation, together with solutions that were adopted by the C and C++ standardisation committees as part of the language. This chapter included an in-depth criticism of the memory-model's treatment of thin-air values, an open problem in memory-model design. Chapter 6 described a mechanised proof that shows the equivalence of progressively simpler versions of the C/C++11 memory model, under successively tighter requirements on programs, culminating in the proof for programs without loops or recursion of one of C++11's stated design goals: race-free programs that use only regular memory accesses, locks and SEQ_CST-annotated atomic accesses behave in a sequentially consistent manner. Chapter 7 presented proofs that the compilation mappings for x86 and Power are sound, establishing that the language is efficiently implementable over those architectures. Chapter 8 described a compositional reasoning principle for C/C++11, and its application to a Treiber stack. Appendix A and B presented a side-by-side comparison of the C++11 standard text [30] and the formal memory model, establishing their correspondence. Finally, Appendix C presented the formal definition of the memory model, automatically typeset by Lem.

This work showcases the power and the practicality of formal semantics for real-world systems, and is in stark contrast to the typical process of solely-prose specification. The formal model created an unambiguous artefact that could be judged without the lens of interpretation, and CPPMEM enabled those without expertise in the formal model to test their expectations of it. The timing of the work, and the uninhibited communication with the standardisation committee meant that some of the errors and omissions of the prose specification were rectified before its ratification. Formalisation of the memory model raised the possibility of establishing desirable properties, like implementability and algorithm correctness, with mathematical proof. The clarity of the formal model enabled lucid analysis of its flaws, in particular its treatment of thin-air values.

Formal model validation In prior work formalising the memory models of hardware and languages, there have been existing processors or compilers that implement the specification (e.g. x86, Power, ARM and Java). A key part of the validation of such a model is extensive testing against existing implementations: it ensures that the memory model

captures all of the behaviour allowed by existing implementations, and is evidence that the model is sound.

The design of the C/C++11 memory model predated implementations (although the design was influenced by the LIBATOMIC_OPS concurrency library written by Boehm et al. [33]). This means that the sorts of validation of the memory model one can perform are different. In the case of C/C++11, this form of validation was not possible at the start. Instead, one must validate that the model is implementable above hardware, and that it is usable by the variety of programmers that it targets. This thesis provides several forms of validation of the memory model.

First, small test programs can be executed according to the mechanised formal model with CPPMEM. The standardisation committee used CPPMEM to test that the model behaved as they expected, and a design goal set out by the standardisation committee was established (for a limited set of programs) with mathematical proof. Together with the close correspondence of the standard argued in Appendix A, and the adjustments to the standard detailed in Chapter 5, these facts are evidence that the formal model matches the *intention* of the standardisation committee.

Second, the soundness proofs of the compilation mappings of Chapter 7 imply that the language is implementable over common processor architectures.

Finally, the reasoning principle described in Chapter 8 shows that it is possible to reason compositionally (with some limitations) about programs written in C++11.

In related work, Morisset et al. used the formal memory model to test the soundness of optimisations, finding mistakes in GCC [84].

These forms of validation are compelling, but not complete. One of the key omissions is the testing and analysis of real-world code. It is not yet clear that the programming idioms that C/C++11 supports match the patterns that programmers of large-scale systems will want to use.

9.1 Future work

There are many avenues for further work presented below.

Push concrete changes from Defect Report 407 to a technical corrigendum Defect report 407 [21] notes that several rules concerning SC fences are absent from the standard, and should be included, and suggests a concrete textual amendment. This suggestion should be incorporated in a technical corrigendum, and in future revisions of the language.

A stronger C/C++11 memory model The models presented in Chapter 3 have been shown to be implementable above hardware, but in many ways, they are weaker than they need to be.

The release-sequence of C/C++11 is quite subtle: it is a subsequence of modification order, one of the dynamic inter-thread relations over memory accesses. There are stronger formulations based on sequenced-before (described in Chapter 5) that would be sound over hardware with the existing compiler mappings.

SC fences correspond to strong memory fences on the underlying hardware. Most target hardware (X86, Power, ARM) allows one to restore SC by adding enough strong fences, but one cannot do the same with SC fences, according to the language memory model. This weakness appears to be an artefact of misunderstanding, and it seems that SC fences could be strengthened.

Solve the thin-air problem Chapter 5 described the thin-air problem: our current relaxed memory models do not handle the interplay between source dependencies and compiler optimisations correctly. As a consequence, the language admits thin-air behaviour that it should not. The specification should be strengthened to forbid this, but there is not a straightforward way to do this within the current specification style. We need new forms of specification that take into account dependencies, and treat them appropriately.

Minimal hardware model for C/C++11 The C/C++11 memory model is very different to a hardware model in that it provides racy programs with undefined behaviour. The style of specification differs too: the language defines relations that have an intuitive connotation like sequenced-before, and happens-before. In simple programs, naive intuitions about program behaviour hold, and it is only when programs use expert features that the model becomes more complicated. Hardware memory models are described quite differently. One could define a hardware model that is only just strong enough to implement the C/C++11 features. This would indicate where existing hardware might be relaxed, and further optimisations could be made.

System-scale relaxed-memory testing The tool presented in this thesis can only run minute litmus tests, and is no use for testing real systems code. Future work should enable scalable testing against the relaxed-memory language specification. Such a testing infrastructure could exercise heuristics that provide the most perverse executions possible, or executions that current hardware does not exhibit, but future hardware may.

Complete formal specifications The formal memory model presented here is only a part of the language specification. In future work, this will be combined with formalisations of other parts of the language specification in order to build up a complete C or C++ formal language model.

Mechanised formal model as specification With a complete mechanised formal model of a programming language, why should the prose specification be authoritative?

There are several barriers preventing mechanised formal specification of mainstream languages. The designers of a language like C are experts in the C language and not in formal specification, so understanding the mathematical representation of the formal model from a direct reading would be a daunting task, and, at first, would provide less confident understanding than the prose specification. There would be a loss of control for the language designers, who would have to describe their wishes to the formal specifiers. A diverse set of users of the language need to understand parts of the specification, and some of those require a prose description of the specification, so a formal model alone will not suffice.

There are ways to overcome these issues. One could engage in an iterative process, where the language designers produce a written specification, the formalisers interpret that as a mechanised formal model, and then provide testing results to tease out corner cases that need further discussion. This process would produce a prose description of the language and a mechanised formal model that match one another. This is the process that emerged informally from my contact with working groups 21 and 14, but it could be improved by starting the collaboration earlier, so that unworkable concepts in the language design (like the C/C++11 thin-air restriction) could be understood before committing to related elements of the design.

With the mechanised formal model, the draft design could be tested, both to expose corner cases, as mentioned above, but also to validate the usability of supported programming idioms. This could be achieved by compiling a body of tests, perhaps automatically, that probe the intricate details of the language. These tests, combined with a tool for exploring the behaviour of the model and capturing new tests, would provide the interface between designers and formalisers. At the end of this process, the mechanised model could be made the authoritative specification, and conformance testing tools could be built directly from the model.

GPGPU memory models The GPU is becoming more important as a computational tool in modern computers; in many systems, the GPU is afforded more resources than the CPU in terms of die area, part cost or power consumption. There has been an effort to enable programmers to write general purpose code above graphics processing hardware, which is tuned for memory throughput, rather than low latency. Until recently, programmers of such systems had been required to avoid writing racy code, but the OpenCL 2.0 specification has introduced atomic accesses similar to those of C11 that expose relaxed memory effects to the programmer. Typical GPUs have several address spaces, each of which identify memory in an increasingly local part of a memory hierarchy. This additional complexity is layered above the existing complexity of the C/C++11 memory model. There are a variety of GPU vendors, each with a different quickly-evolving architecture. Understanding GPU programming is an area for future work.

Future systems: unified address spaces, non-uniform memory hierarchies There are many directions for future hardware designs that would impact the program-

mer's memory model. The CPU and GPU may gain a shared address space, at which point, racy access to this memory will have to be carefully specified, and the explicit memory hierarchy of the GPU will be able to interact with the implicit cache hierarchy of the CPU. Language programming models will have to adapt to these changes.

Relaxed specification languages Few concurrent algorithms or OS calls are formally specified in a manner that describes their relaxed behaviour, but internally, mainstream systems do admit relaxed behaviour. We lack a specification language for describing high-level relaxed memory code. Chapter 8 demonstrated that, for performance, relaxed-memory effects cannot be hidden behind sequential interfaces. This specification language should be human and machine readable and could be included in header files. Specifications could be validated with some combination of proof and testing, both against the language specification.

Systematic programmer-idiom search Current language designs are informed by the programming idioms that the designers intend to support. In practice, programmers may use a different set of idioms. If they use unsupported idioms, the language might not provide the guarantees necessary to ensure correct behaviour, even if testing on current systems reveals the code acts as intended. If no programmer uses a particular idiom that is provided for in the specification, then the potential for optimisation is needlessly curtailed in the compiler and hardware. Future language designs should be informed by systematic analysis of which language guarantees programmers rely on in practice. The language should provide at least this set of guarantees, perhaps with well-motivated additions.

Appendix A

The C++11 standard: side-by-side model comparison

This appendix relates the formal model from Section 3.10 to the text of the C++11 standard [30], showing that there is a tight correspondence between the two. For those familiar with the standard text, this appendix serves as an explanation of the model. C11 adopted the C++11 memory model with little more than editorial adjustments, so the correspondence with the C++ standard argued in this section serves for C as well. We focus on the standard as ratified: errors in drafts that were subsequently fixed and issues that remain in the standard are described in Chapter 5.

The structure of this appendix follows the structure of the standard, interspersing the formal model definition from Section 3.10 with the verbatim text of the standard. The standard describes the memory model in Sections 1, 17, 29 and 30, and parts of these are reproduced here. Note that we suspend the numbering system of the rest of the document for the remainder of this appendix, and adopt the numbering scheme of the standard instead. Comments relating the standard to the formal model are typeset in boxes, and irrelevant parts of the standard are either printed in gray or elided.

This appendix (inevitably) involves the repetition of the definitions of the formal model; no new definitions are introduced.

1 General [intro]

[elided]

1.3 Terms and definitions [intro.defs]

1 For the purposes of this document, the following definitions apply.

2

17.3 defines additional terms that are used only in Clauses 17 through 30 and Annex D.

3 Terms that are used only in a small portion of this International Standard are defined where they are used and italicized where they are defined.

[elided]

1.3.24 [defns.undefined]

Undefined behaviour The standard introduces undefined behaviour below:

undefined behavior behavior for which this International Standard imposes no requirements [*Note:* Undefined behavior may be expected when this International Standard omits any explicit definition of behavior or when a program uses an erroneous construct or erroneous data. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). Many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed. — *end note*]

The note is particularly important for the interpretation of future passages: it implies that the standard simply fails to define the behaviour of some programs, but it may identify faults that lead explicitly to undefined behaviour as well. The majority of the undefined behaviour recognised in the model is explicitly identified by the standard. The model works out instances of undefined behaviour in an execution using a named list of functions that identify either a single faulting action, or a pair of actions engaging in a fault together. The type of the elements in this list is *fault_setgen*:

```
type FAULT_SETGEN =
    ONE of (STRING * (CANDIDATE_EXECUTION → SET (ACTION)))
  | TWO of (STRING * (CANDIDATE_EXECUTION → SET (ACTION * ACTION)))
```

The rest of this section identifies sources of undefined behaviour described by the standard, and for each case, relates the text to a function from executions to actions or pairs of actions in the model. The section concludes by drawing all of these sources together in a record that captures all of the undefined behaviour tracked in the model.

[elided]

1.7 The C++ memory model [intro.memory]

1 The fundamental storage unit in the C++ memory model is the *byte*. A byte is at least large enough to contain any member of the basic execution character set (2.3) and the eight-bit code units of the Unicode UTF-8 encoding form and is composed of a contiguous sequence of bits, the number of which is implementation defined. The least significant bit is called the *low-order bit*; the most significant bit is called the *high-order bit*. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address.

2 [*Note:* The representation of types is described in 3.9. — *end note*]

Locations and location-kinds Many of the disjuncts that make up the action type take a location as a parameter. Section 1.7 introduces memory locations, and their relationship to objects:

3 A *memory location* is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width. [*Note:* Various features of the language, such as references and virtual functions, might involve additional memory locations that are not accessible to programs but are managed by the implementation. — *end note*] Two threads of execution (1.10) can update and access separate memory locations without interfering with each other.

The memory model is not phrased in terms of objects, but instead keeps track of accesses to locations. A map from locations to *location-kinds* captures whether a particular location represents a non-atomic, atomic or mutex object:

```
type LOCATION_KIND =
    MUTEX
  | NON_ATOMIC
  | ATOMIC
```

4 [*Note:* Thus a bit-field and an adjacent non-bit-field are in separate memory locations, and therefore can be concurrently updated by two threads of execution without interference. The same applies to two bit-fields, if one is declared inside a nested struct declaration

and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field declaration. It is not safe to concurrently update two bit-fields in the same struct if all fields between them are also bit-fields of non-zero width. — *end note*]

5 [*Example:* A structure declared as

```
struct {
  char a;
  int b:5,
  c:11,
  :0,
  d:8;
  struct {int ee:8;} e;
}
```

contains four separate memory locations: The field `a` and bit-fields `d` and `e.ee` are each separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields `b` and `c` together constitute the fourth memory location. The bit-fields `b` and `c` cannot be concurrently modified, but `b` and `a`, for example, can be. — *end example*]

1.8 The C++ object model [intro.object]

[elided]

1.9 Program execution [intro.execution]

Top-level judgement At the highest level, the standard claims to describe the behaviour of a program in terms of an abstract machine:

1 The semantic descriptions in this International Standard define a parameterized nondeterministic abstract machine. This International Standard places no requirement on the structure of conforming implementations. In particular, they need not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required

to emulate (only) the observable behavior of the abstract machine as explained below.[5]

> As discussed in Chapter 2, this is not quite true: the standard does not describe a machine that can be incrementally executed; instead, the language is defined in an axiomatic style.

2 Certain aspects and operations of the abstract machine are described in this International Standard as implementation-defined (for example, `sizeof(int)`). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects.[6] Such documentation shall define the instance of the abstract machine that corresponds to that implementation (referred to as the "corresponding instance" below).

Certain other aspects and operations of the abstract machine are described in this International Standard as unspecified (for example, order of evaluation of arguments to a function). Where possible, this International Standard defines a set of allowable behaviors. These define the nondeterministic aspects of the abstract machine. An instance of the abstract machine can thus have more than one possible execution for a given program and a given input.

4 Certain other operations are described in this International Standard as undefined (for example, the effect of attempting to modify a `const` object). [*Note:* This International Standard imposes no requirements on the behavior of programs that contain undefined behavior. — *end note*]

> The passage below describes the behaviour of programs that contain undefined behaviour, and refers again to the model definition in the standard as an abstract machine:

5 A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the

[5] This provision is sometimes called the "as-if" rule, because an implementation is free to disregard any requirement of this International Standard as long as the result is *as if* the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced.

[6] This documentation also includes conditonally-supported constructs and locale-specific behavior. See 1.4.

abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

This passage justifies the top-level judgement of the model: the set of consistent executions of a program is calculated, and if any of these executions contains an instance of undefined behaviour then the behaviour of the program is undefined, otherwise, the behaviour is the set of consistent executions.

The model captures this with a function that takes a memory-model record, a model condition, a thread-local semantics, and a program. The function calculates the consistent executions of the program whose pre-executions are valid according to the thread-local semantics. The function then checks for model condition violations or undefined behaviour and returns either the set of consistent executions, or undefined behaviour:

```
let behaviour M condition opsem (p : PROGRAM) =
  let consistent_executions =
    { (Xo, Xw, rl) |
        opsem p Xo ∧
        apply_tree M.consistent (Xo, Xw, rl) ∧
        rl = M.relation_calculation Xo Xw } in
  if condition consistent_executions ∧
    ∀ X ∈ consistent_executions.
      each_empty M.undefined X
  then Defined (observable_filter consistent_executions)
  else Undefined
```

There are no model condition restrictions on programs that use the standard model, so in presenting the function from programs to their behaviour under the memory model from the standard, the following model condition is used:

```
let true_condition _ = true
```

6 When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects which are neither

- of type `volatile std::sig_atomic_t` nor
- lock-free atomic objects (29.4)

are unspecified during the execution of the signal handler, and the value of any object not in either of these two categories that is modified by the handler becomes undefined.

7 An instance of each object with automatic storage duration (3.7.3) is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).

8 The least requirements on a conforming implementation are:

- Access to volatile objects are evaluated strictly according to the rules of the abstract machine.
- At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.
- The input and output dynamics of interactive devices shall take place in such a fashion that prompting output is actually delivered before a program waits for input. What constitutes an interactive device is implementation-defined.

These collectively are referred to as the *observable behavior* of the program. [*Note:* More stringent correspondences between abstract and actual semantics may be defined by each implementation. — *end note*]

9 [*Note:* Operators can be regrouped according to the usual mathematical rules only where the operators really are associative or commutative.[7] For example, in the following fragment

```
int a, b;
/* ... */
a = a + 32760 + b + 5;
```

the expression statement behaves exactly the same as

```
a = (((a + 32760) + b) + 5);
```

due to the associativity and precedence of these operators. Thus, the result of the sum `(a + 32760)` is next added to `b`, and that result is then added to 5 which results in the value assigned to `a`. On a machine in which overflows produce an exception and in which the range of values representable by an `int` is `[-32768,+32767]`, the implementation cannot rewrite this expression as

[7] Overloaded operators are never assumed to be associative or commutative.

```
a = ((a + b) + 32765);
```

since if the values for `a` and `b` were, respectively, -32754 and -15, the sum `a + b` would produce an exception while the original expression would not; nor can the expression be rewritten either as

```
a = ((a + 32765) + b);
```

or

```
a = (a + (b + 32765));
```

since the values for `a` and `b` might have been, respectively, 4 and -8 or -17 and 12. However on a machine in which overflows do not produce an exception and in which the results of overflows are reversible, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur. — *end note*]

10 A *full-expression* is an expression that is not a subexpression of another expression. If a language construct is defined to produce an implicit call of a function, a use of the language construct is considered to be an expression for the purposes of this definition. A call to a destructor generated at the end of the lifetime of an object other than a temporary object is an implicit full-expression. Conversions applied to the result of an expression in order to satisfy the requirements of the language construct in which the expression appears are also considered to be part of the full-expression. [*Example:*

```
struct S {
  S(int i): I(i) { }
  int& v() { return I; }
private:
  int I;
};

 S s1(1);           // full-expression is call of S::S(int)
 S s2 = 2;          // full-expression is call of S::S(int)

void f() {          // full-expression includes lvalue-to-rvalue and
  if (S(3).v())     // int to bool conversions, performed before
                    // temporary is deleted at end of full-expression
  { }
}
```

— *end example*]

11

[*Note:* The evaluation of a full-expression can include the evaluation of subexpressions that are not lexically part of the full-expression. For example, subexpressions involved in evaluating default arguments (8.3.6) are considered to be created in the expression that calls the function, not the expression that defines the default argument. — *end note*]

All of the atomics and fence calls (defined in Sections 29 and 30 of the standard) are called either on a mutex or an atomic object. The standard describes the behaviour of these calls in terms of the following "operations" on memory. Writes to memory are "modifications" or, more generally "side effects". Reads are described as "value computations", or more generally "evaluations". This language is introduced in the following paragraph:

12 Accessing an object designated by a `volatile` glvalue (3.10), modifying an object, calling a library I/O function, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment. *Evaluation* of an expression (or a sub-expression) in general includes both value computations (including determining the identity of an object for glvalue evaluation and fetching a value previously assigned to an object for prvalue evaluation) and initiation of side effects. When a call to a library I/O function returns or an access to a `volatile` object is evaluated the side effect is considered complete, even though some external actions implied by the call (such as the I/O itself) or by the `volatile` access may not have completed yet.

The action type The formal model describes the behaviour of accesses of memory in terms of memory interactions called *memory actions*, that correspond to "operations" in terms of the standard. Despite mention of the "state of the execution environment" above, the rest of the standard discusses the behaviour of programs in terms of relations over actions where there is no obvious interpretation of an instantaneous state. The formal model defines the action type. It captures the modifications and value computations (or stores and loads), on non-atomic and atomic objects, the locks and unlocks on mutex objects, and memory fences present in an execution. The action type is presented below:

```
type ACTION =
  | LOCK of AID * TID * LOCATION * LOCK_OUTCOME
  | UNLOCK of AID * TID * LOCATION
  | LOAD of AID * TID * MEMORY_ORDER * LOCATION * CVALUE
  | STORE of AID * TID * MEMORY_ORDER * LOCATION * CVALUE
```

```
  | RMW of AID * TID * MEMORY_ORDER * LOCATION * CVALUE * CVALUE
  | FENCE of AID * TID * MEMORY_ORDER
  | BLOCKED_RMW of AID * TID * LOCATION
```

The memory model is stated in terms of candidate executions in which each memory action is identified by a unique identifier and carries the identifier of thread that produced the action. Each memory action above has an *aid* and *tid* that represent this information. The uniqueness of action identifiers is ensured in the *well_formed_threads* predicate by requiring the following action identifier projection function to be injective:

```
let aid_of a =
  match a with
  | Lock aid _ _ _ → aid
  | Unlock aid _ _ → aid
  | Load aid _ _ _ _ → aid
  | Store aid _ _ _ _ → aid
  | RMW aid _ _ _ _ _ → aid
  | Fence aid _ _ → aid
  | Blocked_rmw aid _ _ → aid
  end
```

Sequenced before The pre-execution record contains the sequenced-before relation that records thread-local syntactically-imposed program-order. The standard introduces sequenced before below:

13 *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread (1.10), which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. If A is not sequenced before B and B is not sequenced before A, then A and B are *unsequenced*. [*Note:* The execution of unsequenced evaluations can overlap. — *end note*] Evaluations A and B are *indeterminately sequenced* when either A is sequenced before B or B is sequenced before A, but it is unspecified which. [*Note:* Indeterminately sequenced evaluations cannot overlap, but either could be executed first. — *end note*]

> The memory model deals with memory accesses rather than evaluations, so the thread-local semantics calculates this relation over the memory actions. The standard requires the relation be an asymmetric, transitive partial order, relating only evaluations from the same thread. The *well_formed_threads* predicate ensures that sequenced-before orders the actions accordingly.

14 Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.[8].

15 Except where noted, evaluations of operands of individual operators and of subexpressions of individual expressions are unsequenced. [*Note:* In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations. — *end note*] The value computations of the operands of an operator are sequenced before the value computation of the result of the operator. If a side effect on a scalar object is unsequenced relative to either another side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. [*Example:*

```
void f(int, int);
void g(int i, int* v) {
  i = v[i++];         // the behavior is undefined
  i = 7, i++, i++;    // i becomes 9

  i = i++ + 1;        // the behavior is undefined
  i = i + 1;          // the value of i is incremented

  f(i = -1, i = -1);  // the behavior is undefined
}
```

— *end example*] When calling a function (whether or not the function is inline), every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, is sequenced before execution of every expression or statement in the body of the called function. [*Note:* Value computations and side effects associated with different argument expressions are unsequenced. — *end note*] Every evaluation in the calling function (including other function calls) that is

[8] As specified in 12.2, after a full-expression is evaluated, a sequence of zero or more invocations of destructor functions for temporary objects takes place, usually in reverse order of the construction of each temporary object.

not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.[9] Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit. [*Example:* Evaluation of a `new` expression invokes one or more allocation and constructor functions; see 5.3.4. For another example, invocation of a conversion function (12.3.2) can arise in contexts in which no function call syntax appears. — *end example*] The sequencing constraints on the execution of the called function (as described above) are features of the function calls as evaluated, whatever the syntax of the expression that calls the function might be.

[9] In other words, function executions do not interleave with each other.

indeterminate sequencing Sequenced before can be partial: the paragraph above leaves the ordering of the evaluation of arguments to functions undefined. It also explains that the body of functions on a single thread are ordered by sequenced before, even if they are used as the arguments to a function.

Atomic accesses, fences and mutex calls are all represented in the standard as functions and are therefore always sequenced with respect to other memory accesses. This property must be guaranteed by the thread-local semantics, and is captured by the following predicate:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{indeterminate\_sequencing}\ \mathit{Xo}\ = \\
\quad \forall\ a \in \mathit{Xo.actions}\ b \in \mathit{Xo.actions}. \\
\quad\ (\text{tid\_of}\ a = \text{tid\_of}\ b) \wedge (a \neq b) \wedge \\
\quad\ \neg\ (\text{is\_at\_non\_atomic\_location}\ \mathit{Xo.lk}\ a \wedge \text{is\_at\_non\_atomic\_location}\ \mathit{Xo.lk}\ b) \longrightarrow \\
\quad\ (a,\ b) \in \mathit{Xo.sb} \vee (b,\ a) \in \mathit{Xo.sb}
\end{array}
$$

Unsequenced races The paragraph above identifies insufficient sequenced-before ordering as a source of undefined behaviour. This is an instance where undefined behaviour is the result of a pair of actions, so the model captures the instances of undefined behaviour as a relation. In particular, the relation contains pairs of distinct actions at the same location, on the same thread, where at least one of the actions is a write, and the pair are unrelated by sequenced before:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{unsequenced\_races}\ (\mathit{Xo},\ \_,\ \_)\ = \\
\quad \{\ (a,\ b)\ |\ \forall\ a \in \mathit{Xo.actions}\ b \in \mathit{Xo.actions}\ | \\
\qquad \text{is\_at\_non\_atomic\_location}\ \mathit{Xo.lk}\ a\ \wedge \\
\qquad \neg\ (a = b) \wedge (\text{loc\_of}\ a = \text{loc\_of}\ b) \wedge (\text{is\_write}\ a \vee \text{is\_write}\ b)\ \wedge \\
\qquad (\text{tid\_of}\ a = \text{tid\_of}\ b)\ \wedge \\
\qquad \neg\ ((a,\ b) \in \mathit{Xo.sb} \vee (b,\ a) \in \mathit{Xo.sb})\ \}
\end{array}
$$

1.10 Multi-threaded executions and data races [intro.multithread]

1 A *thread of execution* (also known as a *thread*) is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread. [*Note*: When one thread creates another, the initial call to the top-level function of the new thread is executed by the new thread, not by the creating thread. — *end note*] Every thread in a pro-

gram can potentially access every object and function in a program.[10] Under a hosted implementation, a C++ program can have more than one thread running concurrently. The execution of each thread proceeds as defined by the remainder of this standard. The execution of the entire program consists of an execution of all of its threads. [*Note*: Usually the execution can be viewed as an interleaving of all its threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving, as described below. — *end note*] Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.

2 Implementations should ensure that all unblocked threads eventually make progress. [*Note*: Standard library functions may silently block on I/O or locks. Factors in the execution environment, including externally-imposed thread priorities, may prevent an implementation from making certain guarantees of forward progress. — *end note*]

Reads-from The behaviour of reads from memory is described below:

3 The value of an object visible to a thread T at a particular point is the initial value of the object, a value assigned to the object by T , or a value assigned to the object by another thread, according to the rules below. [*Note*: In some cases, there may instead be undefined behavior. Much of this section is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs. — *end note*]

The paragraph requires a read to read its value from a particular write. Consequently, the model records the particular write that a read reads from, and collects all such relationships together in the reads-from relation. The model checks some basic sanity properties over the relation: that it is an injective relation from write actions to read actions at the same location, and that each read returns the same value as the related write:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{well\_formed\_rf}\ (\mathit{Xo},\ \mathit{Xw},\ \_)\ = \\
\quad \forall\ (a,\ b) \in \mathit{Xw.rf}. \\
\qquad a \in \mathit{Xo.actions} \wedge b \in \mathit{Xo.actions} \wedge \\
\qquad \mathrm{loc\_of}\ a = \mathrm{loc\_of}\ b \wedge \\
\qquad \mathrm{is\_write}\ a \wedge \mathrm{is\_read}\ b \wedge
\end{array}
$$

[10] An object with automatic or thread storage duration (3.7) is associated with one specific thread, and can be accessed by a different thread only indirectly through a pointer or reference (3.9.2).

$$\text{value\_read\_by } b = \text{value\_written\_by } a \wedge$$
$$\forall\, a' \in Xo.actions.\ (a',\ b) \in Xw.rf \longrightarrow a = a'$$

The passage requires reads with a visible side effect to read from some write at the same location. It does not describe the behaviour of the program if there are no writes visible to a given read, but does allude to the possibility of undefined behaviour in the note. In this case, as we shall see in Section B.5.3, the model provides the program with undefined behaviour. For the purposes of calculating the consistent executions, a read that lacks a related write can return any value. The *det_read* predicate forces reads with visible side effects to have a write related to them by reads-from, and leaves all other reads unconstrained:

$$\begin{aligned}&\textsf{let } det\_read\ (Xo,\ Xw,\ \_ :: (\text{“}vse\text{”},\ vse) :: \_)\ = \\ &\quad \forall\, r \in Xo.actions. \\ &\qquad \text{is\_load } r \longrightarrow \\ &\qquad (\exists\, w \in Xo.actions.\ (w,\ r) \in vse) = \\ &\qquad (\exists\, w' \in Xo.actions.\ (w',\ r) \in Xw.rf)\end{aligned}$$

Indeterminate read The standard defines which writes a given read may read from when there is a visible side effect of the read. In the absence of such a side effect, the standard is not precise. The paragraph above defines which values a read may take in the normative text, and then proclaims that some unspecified cases have undefined behaviour in the note. In the model, we provide programs that have reads with no visible side effects with undefined behaviour:

$$\begin{aligned}&\textsf{let } indeterminate\_reads\ (Xo,\ Xw,\ \_)\ = \\ &\quad \{b \mid \forall\ b \in Xo.actions \mid \text{is\_read } b \wedge \neg\ (\exists\ a \in Xo.actions.\ (a,\ b) \in Xw.rf)\}\end{aligned}$$

4 Two expression evaluations *conflict* if one of them modifies a memory location (1.7) and the other one accesses or modifies the same memory location.

5 The library defines a number of atomic operations (Clause 29) and operations on mutexes (Clause 30) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A synchronization operation on one or more memory locations is either a consume operation, an acquire operation, a release operation, or both an acquire and release operation. A synchronization operation without an associated memory location is a fence and can be either an

acquire fence, a release fence, or both an acquire and release fence. In addition, there are relaxed atomic operations, which are not synchronization operations, and atomic read-modify-write operations, which have special characteristics. [*Note:* For example, a call that acquires a mutex will perform an acquire operation on the locations comprising the mutex. Correspondingly, a call that releases the same mutex will perform a release operation on those same locations. Informally, performing a release operation on A forces prior side effects on other memory locations to become visible to other threads that later perform a consume or an acquire operation on A. "Relaxed" atomic operations are not synchronization operations even though, like synchronization operations, they cannot contribute to data races. — *end note*]

Modification order Modification order is introduced below:

6 All modifications to a particular atomic object M occur in some particular total order, called the *modification order* of M . If A and B are modifications of an atomic object M and A happens before (as defined below) B, then A shall precede B in the modification order of M , which is defined below. [*Note:* This states that the modification orders must respect the "happens before" relationship. — *end note*] [*Note:* There is a separate order for each atomic object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads may observe modifications to different objects in inconsistent orders. — *end note*]

The model includes a modification-order relation in the execution witness, that collects together the orders on all atomic locations. The passage requires each modification order to be a total order over the writes at its location. The restriction is imposed by the *consistent_mo* predicate. It requires that mo is transitive and irreflexive, that it agrees with happens-before, and that for every pair of actions, they are related by modification order if and only if they are writes to the same atomic location.

```
let consistent_mo (Xo, Xw, _) =
    relation_over Xo.actions Xw.mo ∧
    isTransitive Xw.mo ∧
    isIrreflexive Xw.mo ∧
    ∀ a ∈ Xo.actions b ∈ Xo.actions.
      ((a, b) ∈ Xw.mo ∨ (b, a) ∈ Xw.mo)
      = ( (¬ (a = b)) ∧
          is_write a ∧ is_write b ∧
```

(loc_of a = loc_of b) ∧
is_at_atomic_location $Xo.lk$ a)

Together these restrictions require mo to be a total order over the writes at each location, as the standard requires. The passage also requires happens-before and modification order to agree, but this is stated again in write-write coherence below. We introduce the model's corresponding restriction there.

Release sequence Release sequences are introduced below:

7 A *release sequence* headed by a release operation A on an atomic object M is a maximal contiguous subsequence of side effects in the modification order of M , where the first operation is A, and every subsequent operation

- is performed by the same thread that performed A, or
- is an atomic read-modify-write operation.

This sequence is captured by the model in two functions. The first takes a head action as an argument, identifies actions on the same thread, and any read-modify-writes, i.e. the actions that may be in the tail of the release sequence:

```
let rs_element head a =
  (tid_of a = tid_of head) ∨ is_RMW a
```

The second function defines a relation, where for every pair (rel, b), rel is the head of a release sequence that contains b. The function first checks that rel is a release. Then the function checks that b is in the release sequence of rel: either b and rel are the same action, or b satisfies $rs\_element$, b follows rel in modification order, and every intervening action in modification order is in the release sequence:

```
let release_sequence_set actions lk mo =
  { (rel, b) | ∀ rel ∈ actions b ∈ actions |
    is_release rel ∧
    ( (b = rel) ∨
      ( (rel, b) ∈ mo ∧
        rs_element rel b ∧
        ∀ c ∈ actions.
          ((rel, c) ∈ mo ∧ (c, b) ∈ mo) ⟶ rs_element rel c ) ) }
```

This relation does not order the members of a particular release sequence, but that information is contained in modification order, and is never used in the judgement of the memory model.

Synchronises with The standard introduces synchronises-with:

8 Certain library calls *synchronize with* other library calls performed by another thread. For example, an atomic store-release synchronizes with a load-acquire that takes its value from the store (29.3). [*Note:* Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation. — *end note*] [*Note:* The specifications of the synchronization operations define when one reads the value written by another. For atomic objects, the definition is clear. All operations on a given mutex occur in a single total order. Each mutex acquisition "reads the value written" by the last mutex release. — *end note*]

The synchronises-with relation captures the majority of inter-thread ordering defined by the standard. The individual cases where library calls do create synchronisation are described throughout the rest of the standard, and not all synchronising features are supported by the thread-local semantics (handler functions, stream objects, thread exit notifications and futures). The sources of synchronisation that are supported are thread creation, mutex accesses, atomic reads and writes, and atomic fences. Each of these will be dealt with in turn.

Carries a dependency to and data dependence The standard introduces weaker inter-thread ordering to express the semantics of consume atomics. Several relations take part in this definition, starting with carries-a-dependency-to:

9 An evaluation A *carries a dependency* to an evaluation B if

- the value of A is used as an operand of B, unless:
  - B is an invocation of any specialization of `std::kill_dependency` (29.3), or
  - A is the left operand of a built-in logical AND (`&&`, see 5.14) or logical OR (——, see 5.15) operator, or
  - A is the left operand of a conditional (`?:`, see 5.16) operator, or

– A is the left operand of the built-in comma (`,`) operator (5.18);

or

- A writes a scalar object or bit-field M , B reads the value written by A from M, and A is sequenced before B, or
- for some evaluation X, A carries a dependency to X , and X carries a dependency to B.

[*Note:* "Carries a dependency to" is a subset of "is sequenced before", and is similarly strictly intra-thread. — end note]

This definition is contingent on a notion of data dependence. The thread-local semantics collects these syntactic dependencies in the data dependence relation, dd, and that is passed to the memory model as part of the pre-execution. The relation is a partial order that identifies a subset of sequenced-before where there is data dependence between memory actions. The restrictions imposed on the relation in *well_formed_threads* are:

strict_partial_order $Xo.actions$ $Xo.dd$ $\wedge$
$Xo.dd$ subset $Xo.sb$

The model has a carries-a-dependency-to relation, cad, that uses the data-dependence relation, dd, previously defined to capture the ordering described by the first bullet in the passage. The second bullet is captured in the standard as the intersection of reads-from and sequenced-before. The final rule includes all transitive closures of the union of the first two rules. The formalisation of the cad relation is simply the transitive closure of the union of data dependence and read-from intersected with sequenced-before:

let $with\_consume\_cad\_set$ $actions$ sb dd rf = transitiveClosure ($(rf \cap sb) \cup dd$)

Dependency ordered before

The next calculated relation defines an analogue of release-acquire synchronisation for release and consume atomics:

10 An evaluation A is *dependency-ordered before* an evaluation B if

- A performs a release operation on an atomic object M, and, in another thread, B

performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or

- for some evaluation X, A is dependency-ordered before X and X carries a dependency to B.

[*Note:* The relation "is dependency-ordered before" is analogous to "synchronizes with", but uses release/consume in place of release/acquire. — *end note*]

The first case in the definition above is similar to the definition of release and acquire synchronisation: a release write is dependency ordered before a read if the release write heads a release sequence that contains a write, and the read reads-from that write. The second case extends dependency-ordered-before edges through all following carries-a-dependency-to edges. The definition of the dependency-ordered-before relation, dob, is formalised directly in the model using the rs and cad calculated relations:

```
let with_consume_dob actions rf rs cad w a =
    tid_of w ≠ tid_of a ∧
    ∃ w' ∈ actions r ∈ actions.
      is_consume r ∧
      (w, w') ∈ rs ∧ (w', r) ∈ rf ∧
      ( (r, a) ∈ cad ∨ (r = a) )
```

Inter-thread happens before

The standard defines inter-thread happens-before next:

11 An evaluation A *inter-thread happens before* an evaluation B if

- A synchronizes with B, or
- A is dependency-ordered before B, or
- for some evaluation X
  - A synchronizes with X and X is sequenced before B, or
  - A is sequenced before X and X inter-thread happens before B, or
  - A inter-thread happens before X and X inter-thread happens before B.

[*Note:* The "inter-thread happens before" relation describes arbitrary concatenations of "sequenced before", "synchronizes with" and "dependency-ordered before" relationships, with two exceptions. The first exception is that a concatenation is not permitted to end with "dependency-ordered before" followed by "sequenced before". The reason for this limitation is that a consume operation participating in a "dependency-ordered before" relationship provides ordering only with respect to operations to which this consume operation actually carries a dependency. The reason that this limitation applies only to the end of such a concatenation is that any subsequent release operation will provide the required ordering for a prior consume operation. The second exception is that a concatenation is not permitted to consist entirely of "sequenced before". The reasons for this limitation are (1) to permit "inter-thread happens before" to be transitively closed and (2) the "happens before" relation, defined below, provides for relationships consisting entirely of "sequenced before". — *end note*]

As the note explains, at each dependency-ordered-before edge, the relation is only extended through dependencies. The definition of inter-thread happens-before, ithb, in the model is:

```
let inter_thread_happens_before actions sb sw dob =
  let r = sw ∪ dob ∪ (compose sw sb) in
    transitiveClosure (r ∪ (compose sb r))
```

This relation has been proven equivalent to a direct formalisation of the standard's recursive definition in the HOL4 theorem prover [2]. The proof relies on the transitivity of sequenced before, and involves an induction over the transitive closure of model happens-before in one direction, and an induction over the rules that define the standard's relation in the other. All cases are trivial except the case where we have an edge in the standard's relation with a sequenced-before edge preceding an inter-thread-happens-before edge. To witness an edge in the model's happens-before relation, we consider two cases: one where the inter-thread-happens-before is a single step in the model's relation, and another where it is a single step followed by an edge in transitively-closed inter-thread-happens-before. In either case, by transitivity of sequenced before the sequenced edge can be composed with the single-step inter-thread-happens-before edge to get an edge in transitively-closed inter-thread-happens-before, and we are done.

Happens-before The standard defines happens-before as follows:

12 An evaluation A *happens before* an evaluation B if:

- A is sequenced before B, or
- A inter-thread happens before B.

The implementation shall ensure that no program execution demonstrates a cycle in the "happens before" relation. [*Note:* This cycle would otherwise be possible only through the use of consume operations. — end note]

The definition of happens-before in the model follows directly from the text. It is simply the union of sequenced-before and inter-thread happens-before:

```
let happens_before actions sb ithb =
  sb ∪ ithb
```

The acyclicity of happens-before is guaranteed by the following predicate over consistent executions:

```
let consistent_hb (Xo, _, ("hb", hb) :: _) =
  isIrreflexive (transitiveClosure hb)
```

Visible side-effects

The standard defines visible side effects below:

13 A *visible side effect* A on a scalar object or bit-field M with respect to a value computation B of M satisfies the conditions:

- A happens before B and
- there is no other side effect X to M such that A happens before X and X happens before B.

The value of a non-atomic scalar object or bit-field M , as determined by evaluation B, shall be the value stored by the visible side effect A. [*Note:* If there is ambiguity about which side effect to a non-atomic object or bit-field is visible, then the behavior is either unspecified or undefined. — *end note*] [*Note:* This states that operations on ordinary objects are not visibly reordered. This is not actually detectable without data races, but it is necessary to ensure that data races, as defined below, and with suitable restrictions on the use of atomics, correspond to data races in a simple interleaved (sequentially consistent) execution. — end note]

The model represents visible side-effects in a relation from each visible side-effect to any reads to which it is visible. Any two related actions must be related by happens-before, they must be a write and a read at the same location, and there must not be a happens-before-intervening write at the same location:

```
let visible_side_effect_set actions hb =
  { (a, b) | ∀ (a, b) ∈ hb |
    is_write a ∧ is_read b ∧ (loc_of a = loc_of b) ∧
    ¬ ( ∃ c ∈ actions. ¬ (c ∈ {a, b}) ∧
        is_write c ∧ (loc_of c = loc_of b) ∧
        (a, c) ∈ hb ∧ (c, b) ∈ hb) }
```

The passage above indicates that visible side effects are used, amongst other things, to determine which writes may be read by non-atomic reads.

It has not been established that the visible side effect of a non-atomic read is unique, and in fact in racy programs, it need not be. The note in the passage above is alluding to this. As a consequence of this possible ambiguity, the model permits non-atomic reads to read from any visible side effect:

```
let consistent_non_atomic_rf (Xo, Xw, _ :: ("vse", vse) :: _) =
  ∀ (w, r) ∈ Xw.rf. is_at_non_atomic_location Xo.lk r ⟶
    (w, r) ∈ vse
```

Visible sequence of side effects The standard includes the notion of a visible sequence of side effects:

14 The *visible sequence of side effects* on an atomic object M, with respect to a value computation B of M, is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first side effect is visible with respect to B, and for every side effect, it is not the case that B happens before it. The value of an atomic object M, as determined by evaluation B, shall be the value stored by some operation in the visible sequence of M with respect to B. [*Note:* It can be shown that the visible sequence of side effects of a value computation is unique given the coherence requirements below. — *end note*]

The model represents these sequences as a relation with edges from the set of writes that makes up each sequence, to its read. The ordering of the sequence is provided by modification order. The definition of the relation is a set comprehension that, for a given read, first finds the modification-order-maximal visible side effect of the read, and then relates this write to the read, as well as any modification-order-following writes that do not happen after the read, or have a modification-order intervening write that does:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{standard\_vsses}\ \mathit{actions}\ \mathit{lk}\ \mathit{mo}\ \mathit{hb}\ \mathit{vse}\ = \\
\quad \{\ (v,\ r)\ |\ \forall\ r \in \mathit{actions}\ v \in \mathit{actions}\ \mathit{head} \in \mathit{actions}\ | \\
\quad\ \mathrm{is\_at\_atomic\_location}\ \mathit{lk}\ r \wedge (\mathit{head},\ r) \in \mathit{vse} \wedge \\
\quad\ \neg\ (\exists\ v' \in \mathit{actions}.\ (v',\ r) \in \mathit{vse} \wedge (\mathit{head},\ v') \in \mathit{mo}) \wedge \\
\quad\ (\ v = \mathit{head} \vee \\
\quad\quad (\ (\mathit{head},\ v) \in \mathit{mo} \wedge \neg\ ((r,\ v) \in \mathit{hb}) \wedge \\
\quad\quad\ \forall\ w \in \mathit{actions}. \\
\quad\quad\quad ((\mathit{head},\ w) \in \mathit{mo} \wedge (w,\ v) \in \mathit{mo}) \longrightarrow \neg\ ((r,\ w) \in \mathit{hb}) \\
\quad\quad\ ) \\
\quad\ ) \\
\quad\ \}
\end{array}
$$

In the passage above, the standard says that the write read by an atomic read is restricted to a write in the visible sequence of side effects of the read.

As we shall see in Chapter 5, this sequence was intended to enumerate the set of all writes that may be read by an atomic read, but it is made redundant by the coherence requirements. Nonetheless the mathematical model that follows the standard includes the restriction. Every read at an atomic location must read from a write in its visible sequence of side effects:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{standard\_consistent\_atomic\_rf}\ (\mathit{Xo},\ \mathit{Xw},\ \_ ::\ \_ ::\ \_ ::\ (\text{“}\mathit{vsses}\text{”},\ \mathit{vsses}) ::\ \_)\ = \\
\quad \forall\ (w,\ r) \in \mathit{Xw.rf}.\ \mathrm{is\_at\_atomic\_location}\ \mathit{Xo.lk}\ r \wedge \mathrm{is\_load}\ r \longrightarrow \\
\quad\quad (w,\ r) \in \mathit{vsses}
\end{array}
$$

Coherence Visible sequences of side effects were originally intended to enumerate the writes that a particular atomic read could consistently read from. This relation was superseded by the four coherence requirements below. Each is a relatively simple subgraph of reads and writes at a single location. The graphs are made up of modification order, happens-before and reads-from edges, and each represents a behaviour

where the three relations disagree in some way.

CoWW Modification order and happens-before must agree:

15 If an operation A that modifies an atomic object M happens before an operation B that modifies M, then A shall be earlier than B in the modification order of M. [*Note:* This requirement is known as write-write coherence. — *end note*]

The model formalises this restriction as follows:

```
let coherent_memory_use (Xo, Xw, ("hb", hb) :: _) =
[...]   (* CoWW *)
   (¬ (∃(a, b)∈hb. (b, a) ∈ Xw.mo))))
```

CoRR

In this behaviour, two reads on a single thread observe writes in an order that opposes modification order:

16 If a value computation A of an atomic object M happens before a value computation B of M, and A takes its value from a side effect X on M, then the value computed by B shall either be the value stored by X or the value stored by a side effect Y on M, where Y follows X in the modification order of M. [*Note:* This requirement is known as read-read coherence. — *end note*]

This restriction is captured by the following part of the consistency predicate:

```
let coherent_memory_use (Xo, Xw, ("hb", hb) :: _) =
   (* CoRR *)
   (¬ (∃(a, b)∈Xw.rf (c, d)∈Xw.rf.
         (b, d) ∈ hb ∧ ((c, a) ∈ Xw.mo))) ∧
[...]
```

The other coherence requirements can be seen as derivatives of this where a read and write that participate in a reads-from edge are replaced with a single write in place of the read.

CoRW In this coherence violation, there is a cycle created by modification order, happens-before and reads-from:

17 If a value computation A of an atomic object M happens before an operation B on M, then A shall take its value from a side effect X on M, where X precedes B in the modification order of M. [*Note:* This requirement is known as read-write coherence. — *end note*]

The model formalises this restriction as follows:

```
let coherent_memory_use (Xo, Xw, ("hb", hb) :: _) =
[...]   (* CoRW *)
   ((¬ (∃(a, b)∈Xw.rf  c∈Xo.actions.
          (b, c)  ∈  hb ∧ ((c, a)  ∈  Xw.mo))) ∧
[...]
```

CoWR Here the read reads from a write that is hidden by a modification-order-later write that happens before the read:

18 If a side effect X on an atomic object M happens before a value computation B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M. [*Note:* This requirement is known as write-read coherence. — *end note*]

The model formalises this restriction as follows:

```
let coherent_memory_use (Xo, Xw, ("hb", hb) :: _) =
[...]   (* CoWR *)
   ((¬ (∃(a, b)∈Xw.rf  c∈Xo.actions.
          (c, b)  ∈  hb ∧ ((a, c)  ∈  Xw.mo))) ∧
[...]
```

The coherence restriction These four restrictions are combined in the *coherent_memory_use* restriction in the memory model:

```
let coherent_memory_use (Xo, Xw, ("hb", hb) :: _) =
  (* CoRR *)
  ( ¬ ( ∃ (a, b) ∈ Xw.rf (c, d) ∈ Xw.rf.
        (b, d) ∈ hb ∧ (c, a) ∈ Xw.mo ) ) ∧
  (* CoWR *)
  ( ¬ ( ∃ (a, b) ∈ Xw.rf c ∈ Xo.actions.
        (c, b) ∈ hb ∧ (a, c) ∈ Xw.mo ) ) ∧
  (* CoRW *)
  ( ¬ ( ∃ (a, b) ∈ Xw.rf c ∈ Xo.actions.
        (b, c) ∈ hb ∧ (c, a) ∈ Xw.mo ) ) ∧
  (* CoWW *)
  ( ¬ (∃ (a, b) ∈ hb. (b, a) ∈ Xw.mo) )
```

19 [*Note:* The four preceding coherence requirements effectively disallow compiler reordering of atomic operations to a single object, even if both operations are relaxed loads. This effectively makes the cache coherence guarantee provided by most hardware available to C++ atomic operations. — *end note*]

20 [*Note:* The visible sequence of side effects depends on the "happens before" relation, which depends on the values observed by loads of atomics, which we are restricting here. The intended reading is that there must exist an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the "happens before" relation derived as described above, satisfy the resulting constraints as imposed here. — *end note*]

Data races Concurrent programs must order their non-atomic accesses to memory sufficiently, or face undefined behaviour. Pairs of accesses with insufficient happens-before ordering are faults, called data races, introduced by the standard below:

21 The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior. [*Note:* It can be shown that programs that correctly use mutexes and `memory_order_seq_cst` operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally

referred to as "sequential consistency". However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result must perform an undefined operation. — *end note*]

The definition above relies on the definition of conflict from 1.10p4. The model unpacks the definition of conflict, and defines the set of data races in a program as the set of pairs of distinct actions at the same location where the actions are on different threads, at least one is a write, at most one is atomic, and where the pair is unrelated by happens-before:

$$
\begin{aligned}
&\textsf{let } \mathit{data\_races}\ (Xo,\ Xw,\ (\text{“}hb\text{”},\ hb) :: \_)\ = \\
&\quad \{\ (a,\ b)\ |\ \forall\ a \in Xo.\mathit{actions}\ b \in Xo.\mathit{actions}\ | \\
&\qquad \neg\ (a = b) \wedge (\text{loc\_of}\ a = \text{loc\_of}\ b) \wedge (\text{is\_write}\ a \vee \text{is\_write}\ b) \wedge \\
&\qquad (\text{tid\_of}\ a \neq \text{tid\_of}\ b) \wedge \\
&\qquad \neg\ (\text{is\_atomic\_action}\ a \wedge \text{is\_atomic\_action}\ b) \wedge \\
&\qquad \neg\ ((a,\ b) \in hb \vee (b,\ a) \in hb)\ \}
\end{aligned}
$$

22 [*Note:* Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this standard, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race. This includes implementations of data member assignment that overwrite adjacent members in separate memory locations. Reordering of atomic loads in cases in which the atomics in question may alias is also generally precluded, since this may violate the "visible sequence" rules. — *end note*]

23 [*Note:* Transformations that introduce a speculative read of a potentially shared memory location may not preserve the semantics of the C++ program as defined in this standard, since they potentially introduce a data race. However, they are typically valid in the context of an optimizing compiler that targets a specific machine with well-defined semantics for data races. They would be invalid for a hypothetical machine that is not tolerant of races or provides hardware race detection. — *end note*]

24 The implementation may assume that any thread will eventually do one of the following:

- terminate,
- make a call to a library I/O function,

- access or modify a volatile object, or
- perform a synchronization operation or an atomic operation.

[*Note:* This is intended to allow compiler transformations such as removal of empty loops, even when termination cannot be proven. — *end note*]

25 An implementation should ensure that the last value (in modification order) assigned by an atomic or synchronization operation will become visible to all other threads in a finite period of time.

17 Library introduction [library]

[elided]

17.6 Library-wide requirements [requirements]

[elided]

17.6.4 Constraints on programs [constraints]

[elided]

17.6.4.11 Requires paragraph [res.on.required]

Undefined behaviour

The standard places preconditions on the use of some features (especially locks and unlocks over mutex locations). If a program contains an execution that does not satisfy these preconditions then the passage below tells us that the program has undefined behaviour:

1 Violation of the preconditions specified in a function's *Requires:* paragraph results in undefined behavior unless the function's *Throws:* paragraph specifies throwing an exception when the precondition is violated.

[elided]

29 Atomic operations library [atomics]

29.1 General [atomics.general]

1 This Clause describes components for fine-grained atomic access. This access is provided via operations on atomic objects.

2 The following subclauses describe atomics requirements and components for types and operations, as summarized below.

Table 144 — Atomics library summary

	Subclause	Header(s)
29.3	Order and Consistency	
29.4	Lock-free Property	
29.5	Atomic Types	¡atomic¿
29.6	Operations on Atomic Types	
29.7	Flag Type and Operations	
29.8	Fences	

29.2 Header `<atomic>` synopsis [atomics.syn]

[elided]

29.3 Order and consistency [atomics.order]

Memory order Loads, stores, read-modify-writes and fences all take a memory-order parameter. The following enumeration specifies the options:

```
namespace std {
  typedef enum memory_order {
    memory_order_relaxed, memory_order_consume, memory_order_acquire,
    memory_order_release, memory_order_acq_rel, memory_order_seq_cst
  } memory_order;
}
```

In the model, the *memory_order* type captures these orders, adding non-atomic ordering, *NA*, to the list to unify atomic and non-atomic actions:

```
type MEMORY_ORDER =
  | NA
  | SEQ_CST
  | RELAXED
  | RELEASE
  | ACQUIRE
  | CONSUME
  | ACQ_REL
```

Not all actions can be given every memory order. The *well_formed_actions* predicate, described in Section B.1.2, will define which memory orders are allowed for each variety of action.

Release and acquire actions Depending on the choice of memory order, some memory accesses may be identified by the standard as releases or acquires. The standard defines which loads, stores and read-modify-writes are releases and acquires below: (whether fences are releases or acquires is defined in Section 29.8 of the standard):

1 The enumeration `memory_order` specifies the detailed regular (non-atomic) memory synchronization order as defined in 1.10 and may provide for operation ordering. Its enumerated values and their meanings are as follows:

- `memory_order_relaxed`: no operation orders memory.
- `memory_order_release`, `memory_order_acq_rel`, and `memory_order_seq_cst`: a store operation performs a release operation on the affected memory location.
- `memory_order_consume`: a load operation performs a consume operation on the affected memory location.
- `memory_order_acquire`, `memory_order_acq_rel`, and `memory_order_seq_cst`: a load operation performs an acquire operation on the affected memory location.

[*Note:* Atomic operations specifying `memory_order_relaxed` are relaxed with respect to memory ordering. Implementations must still guarantee that any given atomic access to a particular atomic object be indivisible with respect to all other atomic accesses to that object. — *end note*]

The model includes functions that take an action and judge whether it is a release or an acquire. The definitions collect together the requirements from the quote above and that in 29.8:

```
let is_release a =
  match a with
  | Store _ _ mo _ _ → mo ∈ {Release, Seq_cst}
  | RMW _ _ mo _ _ _ → mo ∈ {Release, Acq_rel, Seq_cst}
  | Fence _ _ mo → mo ∈ {Release, Acq_rel, Seq_cst}
  | _ → false
  end

let is_acquire a =
  match a with
  | Load _ _ mo _ _ → mo ∈ {Acquire, Seq_cst}
  | RMW _ _ mo _ _ _ → mo ∈ {Acquire, Acq_rel, Seq_cst}
  | Fence _ _ mo → mo ∈ {Acquire, Consume, Acq_rel, Seq_cst}
  | _ → false
  end
```

Release-acquire synchronisation

Release-acquire synchronisation is described below:

2 An atomic operation A that performs a release operation on an atomic object M synchronizes with an atomic operation B that performs an acquire operation on M and takes its value from any side effect in the release sequence headed by A.

The model already has a calculated relation that represents release sequences, so the formalisation of this requirement is straightforward. Synchronisation is created from release actions to acquire actions that read from writes in the release sequence of the release:

```
let release_acquire_fenced_synchronizes_with actions sb asw rf lo rs hrs a b =
  (tid_of a ≠ tid_of b) ∧
([...]   (* rel/acq sync *)
    (is_release a ∧ (is_acquire b ∧
```

$(\exists c \in actions.\ (a,\ c)\ \in\ rs \wedge (c,\ b)\ \in\ rf) \vee$
$[\ldots])$

SC order and SC reads The SC order and the additional restriction it imposes on sequentially consistent reads are introduced below:

3 There shall be a single total order S on all `memory_order_seq_cst` operations, consistent with the "happens before" order and modification orders for all affected locations, such that each `memory_order_seq_cst` operation B that loads a value from an atomic object M observes one of the following values:

- the result of the last modification A of M that precedes B in S, if it exists, or
- if A exists, the result of some modification of M in the visible sequence of side effects with respect to B that is not `memory_order_seq_cst` and that does not happen before A, or
- if A does not exist, the result of some modification of M in the visible sequence of side effects with respect to B that is not `memory_order_seq_cst`.

[*Note:* Although it is not explicitly required that S include locks, it can always be extended to an order that does include lock and unlock operations, since the ordering between those is already included in the "happens before" ordering. — *end note*]

The model captures the S order with the sc relation, and the restrictions required by the standard are imposed by the *sc_accesses_consistent_sc* predicate. The predicate is similar to those of modification order and lock order, but this relation is not restricted to a single location, it is over all SEQ_CST accesses, and it must agree with both happens-before and modification order:

```
let sc_accesses_consistent_sc (Xo, Xw, ("hb", hb) :: _) =
    relation_over Xo.actions Xw.sc ∧
    isTransitive Xw.sc ∧
    isIrreflexive Xw.sc ∧
    ∀ a ∈ Xo.actions b ∈ Xo.actions.
      ((a, b) ∈ Xw.sc ⟶ ¬ ((b, a) ∈ hb ∪ Xw.mo)) ∧
      ( ((a, b) ∈ Xw.sc ∨ (b, a) ∈ Xw.sc) =
        ( (¬ (a = b)) ∧ is_seq_cst a ∧ is_seq_cst b)
      )
```

The predicate requires sc order to be an irreflexive total order over SEQ_CST accesses that agrees with happens-before and modification order, as required by the standard

The passage above describes the behaviour of SC reads in three cases, but by phrasing the restriction differently, the model coalesces the last two. In the case that the sequentially consistent read reads from a sequentially consistent write, the model requires that there is no sc intervening write to the same location. This matches the first case above. To cover the second and third cases, where the write is not sequentially consistent, the model forbids executions where the write has a happens-before successor that is a sequentially consistent write, and that write is sc ordered before the read. This restriction encompasses both the case where there is no sc predecessor of the read, and the case where there is. The restriction in the model is therefore:

$$
\begin{aligned}
&\textsf{let } \mathit{sc\_accesses\_sc\_reads\_restricted}\ (Xo,\ Xw,\ (\text{“}hb\text{”},\ hb) :: \_)\ = \\
&\quad \forall\ (w,\ r) \in Xw.rf.\ \text{is\_seq\_cst}\ r \longrightarrow \\
&\quad\ (\ \text{is\_seq\_cst}\ w \wedge (w,\ r) \in Xw.sc \wedge \\
&\quad\quad \neg\ (\exists\ w' \in Xo.actions. \\
&\quad\quad\quad \text{is\_write}\ w' \wedge (\text{loc\_of}\ w = \text{loc\_of}\ w') \wedge \\
&\quad\quad\quad (w,\ w') \in Xw.sc \wedge (w',\ r) \in Xw.sc\ )\ ) \vee \\
&\quad\ (\ \neg\ (\text{is\_seq\_cst}\ w) \wedge \\
&\quad\quad \neg\ (\exists\ w' \in Xo.actions. \\
&\quad\quad\quad \text{is\_write}\ w' \wedge (\text{loc\_of}\ w = \text{loc\_of}\ w') \wedge \\
&\quad\quad\quad (w,\ w') \in hb \wedge (w',\ r) \in Xw.sc\ )\ )
\end{aligned}
$$

SC fences The semantics of fences with sequentially consistent memory order are given next. There are, broadly speaking, two sorts of restriction imposed: those that impact reads-from edges and those that impact modification order. For each case, there is a canonical restriction that uses two sequentially consistent fences, and then there are some derivative restrictions that govern the interaction between sequentially consistent fences and sequentially consistent atomic accesses. The model enforces stronger restrictions than the standard here — the standard omits the derivative cases for restricting modification order. There is an outstanding proposal to change this, and it has met with informal approval amongst the C++ committee. This proposal is further discussed in Chapter 5.

The canonical reads-from restriction The standard imposes the following restriction on the writes that may be read by an atomic read, in the presence of sequentially consistent fences:

4 For an atomic operation B that reads the value of an atomic object M, if there is a `memory_order_seq_cst` fence X sequenced before B, then B observes either the last `memory_order_seq_cst` modification of M preceding X in the total order S or a later modification of M in its modification order.

Rather than identifying the writes that may be read from, as the standard does above, the model identifies those writes that may not be read. The restriction below forbids executions from reading writes that precede the last modification of the location before the fence:

```
let sc_fenced_sc_fences_heeded (Xo, Xw, ("hb", hb) :: _) =
  ∀f∈Xo.actions f'∈Xo.actions
      r∈Xo.actions
      w∈Xo.actions w'∈Xo.actions. ¬ (is_fence f ∧ (is_fence f' ∧
[...]          (* fence restriction N3291 29.3p6 *)
          (((w, w') ∈ Xw.mo ∧
            ((w', f) ∈ Xo.sb ∧
            ((f, f') ∈ Xw.sc ∧
            ((f', r) ∈ Xo.sb ∧
            ((w, r) ∈ Xw.rf))))) ∨
[...]
```

First read-from derivative In this passage, the standard describes the interaction of a sequentially consistent read with an atomic write followed in the same thread by a sequentially consistent fence:

5 For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there is a `memory_order_seq_cst` fence X such that A is sequenced before X and B follows X in S, then B observes either the effects of A or a later modification of M in its modification order.

Again, the model formalises this by forbidding the read from reading writes earlier in the modification order than the one that precedes the fence.

```
let sc_fenced_sc_fences_heeded (Xo, Xw, ("hb", hb) :: _) =
  ∀f∈Xo.actions f'∈Xo.actions
      r∈Xo.actions
```

```
        w∈Xo.actions w'∈Xo.actions. ¬ (is_fence f ∧ (is_fence f' ∧
[...]       (* fence restriction N3291 29.3p5 *)
          (((w, w') ∈ Xw.mo ∧
            ((w', f) ∈ Xo.sb ∧
            ((f, r) ∈ Xw.sc ∧
            ((w, r) ∈ Xw.rf)))) ∨
[...]
```

Second read-from derivative In this passage, the standard describes the interaction of a sequentially consistent write with a sequentially consistent fence followed in the same thread by an atomic read:

6 For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S, then B observes either the effects of A or a later modification of M in its modification order.

Again, the restriction in the model forbids rather than allows reads-from edges. This time, there is an atomic write in the writing thread, and the restriction forbids the read from reading a modification predecessor of this write:

```
let sc_fenced_sc_fences_heeded (Xo, Xw, ("hb", hb) :: _) =
  ∀f∈Xo.actions f'∈Xo.actions
       r∈Xo.actions
       w∈Xo.actions w'∈Xo.actions. ¬ (is_fence f ∧ (is_fence f' ∧
       ((* fence restriction N3291 29.3p4 *)
         ((w, w') ∈ Xw.mo ∧
           ((w', f) ∈ Xw.sc ∧
           ((f, r) ∈ Xo.sb ∧
           ((w, r) ∈ Xw.rf)))) ∨
[...]
```

Canonical modification order restriction The standard provides one further restriction: sc ordering over fences imposes modification order over writes in the fenced threads:

7

For atomic operations A and B on an atomic object M, if there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S, then B occurs later than A in the modification order of M.

Once more, the restriction is expressed as a negation in the model. Rather than forcing modification order to agree with sc order, the model forbids it from disagreeing, relying on the totality of modification order to ensure there is an edge that agrees.

```
let sc_fenced_sc_fences_heeded (Xo, Xw, ("hb", hb) :: _) =
  ∀f∈Xo.actions f'∈Xo.actions
      r∈Xo.actions
      w∈Xo.actions w'∈Xo.actions. ¬ (is_fence f ∧ (is_fence f' ∧
[...]          (* SC fences impose mo N3291 29.3p7 *)
          (((w', f) ∈ Xo.sb ∧
            (f, f') ∈ Xw.sc ∧
            (f', w) ∈ Xo.sb ∧
            (w, w') ∈ Xw.mo)))) ∨
[...]
```

For the two derivatives, see Chapter 3

8 [*Note:* `memory_order_seq_cst` ensures sequential consistency only for a program that is free of data races and uses exclusively `memory_order_seq_cst` operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In particular, `memory_order_seq_cst` fences ensure a total order only for the fences themselves. Fences cannot, in general, be used to restore sequential consistency for atomic operations with weaker ordering specifications. — *end note*]

9 An atomic store shall only store a value that has been computed from constants and program input values by a finite sequence of program evaluations, such that each evaluation observes the values of variables as computed by the last prior assignment in the sequence. The ordering of evaluations in this sequence shall be such that:

- if an evaluation B observes a value computed by A in a different thread, then B does not happen before A, and
- if an evaluation A is included in the sequence, then every evaluation that assigns to the same variable and happens before A is included.

10

[*Note:* The second requirement disallows "out-of-thin-air" or "speculative" stores of atomics when relaxed atomics are used. Since unordered operations are involved, evaluations may appear in this sequence out of thread order. For example, with x and y initially zero,

```
// Thread 1:
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);

// Thread 2:
r2 = x.load(memory_order_relaxed);
y.store(42, memory_order_relaxed);
```

is allowed to produce `r1 = r2 = 42`. The sequence of evaluations justifying this consists of:

```
y.store(42, memory_order_relaxed);
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);
r2 = x.load(memory_order_relaxed);
```

On the other hand,

```
// Thread 1:
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);

// Thread 2:
r2 = x.load(memory_order_relaxed);
y.store(r2, memory_order_relaxed);
```

may not produce `r1 = r2 = 42`, since there is no sequence of evaluations that results in the computation of 42. In the absence of "relaxed" operations and read-modify-write operations with weaker than `memory_order_acq_rel` ordering, the second requirement has no impact. — *end note*]

11 [*Note:* The requirements do allow r1 == r2 == 42 in the following example, with x and y initially zero:

```
// Thread 1:
r1 = x.load(memory_order_relaxed);
if (r1 == 42) y.store(r1, memory_order_relaxed);

// Thread 2:
r2 = y.load(memory_order_relaxed);
if (r2 == 42) x.store(42, memory_order_relaxed);
```

However, implementations should not allow such behavior. — *end note*]

Modification order and read-modify-writes Modification order ensures the atomicity of read-modify-write actions. The standard restricts the writes that may be read by read-modify-write actions:

12 Atomic read-modify-write operations shall always read the last value (in the modification order) written before the write associated with the read-modify-write operation.

The corresponding predicate in the model checks that reads-from edges to read-modify-write actions come from the immediately preceding write in modification order, establishing the restriction from the standard directly:

$$\begin{array}{l}\textsf{let } rmw\_atomicity\ (Xo,\ Xw,\ \_)\ = \\ \quad \forall\ b \in Xo.actions\ a \in Xo.actions. \\ \quad\ \text{is\_RMW}\ b \longrightarrow (\text{adjacent\_less\_than}\ Xw.mo\ Xo.actions\ a\ b = ((a,\ b) \in Xw.rf))\end{array}$$

13 Implementations should make atomic stores visible to atomic loads within a reasonable amount of time.

```
template <class T>
  T kill_dependency(T y) noexcept;
```

14 *Effects:* The argument does not carry a dependency to the return value (1.10).

15 *Returns:* `y`.

29.4 Lock-free property [atomics.lockfree]

[elided]

29.5 Atomic types [atomics.types.generic]

[elided]

29.6 Operations on atomic types [atomics.types.operations]

[elided]

29.6.5 Requirements for operations on atomic types [atomics.types.operations.req]

> Sections 29 and 30, introduce atomic acesses and mutexes respectively. Each section defines a limited set of accesses that can be made to atomics and mutexes.

1 There are only a few kinds of operations on atomic types, though there are many instances on those kinds. This section specifies each general kind. The specific instances are defined in 29.5, 29.6.1, 29.6.3, and 29.6.4.

> These sections guarantee that pre-executions will not feature a lock at an atomic location, for instance. The memory model captures this guarantee in the *actions_respect_location_kinds* predicate on pre-executions. It requires that lock and unlock actions act on mutex locations, that atomic actions act on atomic locations, and it forbids non-atomic loads on atomic locations — there is no way to non-atomically load from an atomic location in C++11, but in C11 one can use `memcpy` to copy an atomic object. We match C++11 and consider executions without non-atomic reads of atomic locations, but the model is intended to apply to executions that include them.
>
> Writes to atomic objects that are the result of initialisation are non-atomic, so non-atomic stores at atomic locations are allowed. Non-atomic locations may only be accessed by non-atomic loads and stores. The *actions_respect_location_kinds* predicate is given below:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{actions\_respect\_location\_kinds}\ \mathit{actions}\ \mathit{lk}\ = \\
\quad \forall\ a \in \mathit{actions}.\ \mathsf{match}\ a\ \mathsf{with} \\
\qquad \mid \text{Lock}\ \_\ \_\ l\ \_\ \rightarrow\ \mathit{lk}\ l = \text{Mutex} \\
\qquad \mid \text{Unlock}\ \_\ \_\ l\ \rightarrow\ \mathit{lk}\ l = \text{Mutex} \\
\qquad \mid \text{Load}\ \_\ \_\ \mathit{mo}\ l\ \_\ \rightarrow \\
\qquad\quad (\mathit{mo} = \text{NA} \wedge \mathit{lk}\ l = \text{Non\_Atomic}) \vee (\mathit{mo} \neq \text{NA} \wedge \mathit{lk}\ l = \text{Atomic}) \\
\qquad \mid \text{Store}\ \_\ \_\ \mathit{mo}\ l\ \_\ \rightarrow \\
\qquad\quad (\mathit{mo} = \text{NA} \wedge \mathit{lk}\ l = \text{Non\_Atomic}) \vee \mathit{lk}\ l = \text{Atomic}
\end{array}
$$

```
  | RMW _ _ _ l _ _ → lk l = Atomic
  | Fence _ _ _ → true
  | Blocked_rmw _ _ l → lk l = Atomic
  end
```

2 In the following operation definitions:

- an A refers to one of the atomic types.
- a C refers to its corresponding non-atomic type. The `atomic_address` atomic type corresponds to the `void*` non-atomic type.
- an M refers to type of the other argument for arithmetic operations. For integral atomic types, M is C. For atomic address types, M is `std::ptrdiff_t`.
- the free functions not ending in `_explicit` have the semantics of their corresponding `_explicit` with `memory_order` arguments of `memory_order_seq_cst`.

3 [*Note:* Many operations are volatile-qualified. The "volatile as device register" semantics have not changed in the standard. This qualification means that volatility is preserved when applying these operations to volatile objects. It does not mean that operations on non-volatile objects become volatile. Thus, volatile qualified operations on non-volatile objects may be merged under some conditions. — *end note*]

```
A::A() noexcept = default;
```

4 *Effects:* leaves the atomic object in an uninitialized state. [*Note:* These semantics ensure compatibility with C. — *end note*]

```
constexpr A::A(C desired) noexcept;
```

5 *Effects:* Initializes the object with the value `desired`. Initialization is not an atomic operation (1.10). [*Note:* it is possible to have an access to an atomic object A race with its construction, for example by communicating the address of the just-constructed object A to another thread via `memory_order_relaxed` operations on a suitable atomic pointer variable, and then immediately accessing A in the receiving thread. This results in undefined behavior. — *end note*]

Racy initialisation It is important to note that initialisation on atomic locations is performed with a non-atomic write, that will race with happens-before unordered atomic accesses of the location. The standard provides two ways to initialise atomic variables. The first is via a macro:

`#define ATOMIC_VAR_INIT(value)` see below

6 The macro expands to a token sequence suitable for constant initialization of an atomic variable of static storage duration of a type that is initialization-compatible with *value*. [*Note:* This operation may need to initialize locks. — *end note*] Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race. [*Example:* `atomic<int> v = ATOMIC_VAR_INIT(5);` — *end example*]

Here the standard explicitly identifies the possibility of a data race with the initialisation. The introduction of the other syntax for atomic initialisation below includes a similar note.

```
bool atomic_is_lock_free(const volatile A *object) noexcept;
bool atomic_is_lock_free(const A *object) noexcept;
bool A::is_lock_free() const volatile noexcept;
bool A::is_lock_free() const noexcept;
```

7 *Returns:* True if the object's operations are lock-free, false otherwise.

This syntax allows the initialisation of an existing atomic object:

```
void atomic_init(volatile A *object, C desired) noexcept;
void atomic_init(A *object, C desired) noexcept;
```

8 *Effects:* Non-atomically initializes `*object` with value desired. This function shall only be applied to objects that have been default constructed, and then only once. [*Note:* These semantics ensure compatibility with C. — *end note*] [*Note:* Concurrent access from another thread, even via an atomic operation, constitutes a data race. — *end note*]

The model captures these potential races in the thread-local semantics by expressing initialisation writes at atomic locations as non-atomic writes. In early drafts of the standard, atomics could be re-initialised, permitting multiple non-atomic writes to atomic location. The text above explicitly forbids this.

Well-formed actions The following requirement restricts the memory order that may be given to stores. There are similar requirements for loads in Paragraph 29.6.5p13 and for CAS-like accesses in Passage 29.6.5p20.

```
void atomic_store(volatile A* object, C desired) noexcept;
void atomic_store(A* object, C desired) noexcept;
void atomic_store_explicit(volatile A *object, C desired, memory_order order) noexcept;
void atomic_store_explicit(A* object, C desired, memory_order order) noexcept;
void A::store(C desired, memory_order order = memory_order_seq_cst) volatile noexcept;
void A::store(C desired, memory_order order = memory_order_seq_cst) noexcept;
```

9 *Requires:* The order argument shall not be `memory_order_consume`, `memory_order_acquire`, nor `memory_order_acq_rel`.

The model collects these requirements in the *well_formed_action* predicate. In particular, we require stores and loads to take the correct memory orders, we restrict the memory order of a read resulting from a failed CAS, and we restrict atomic accesses from taking the non-atomic memory order:

```
let well_formed_action a =
  match a with
    | Load _ _ mo _ _ → mo ∈ {NA, Relaxed, Acquire, Seq_cst, Consume}
    | Store _ _ mo _ _ → mo ∈ {NA, Relaxed, Release, Seq_cst}
    | RMW _ _ mo _ _ _ → mo ∈ {Relaxed, Release, Acquire, Acq_rel, Seq_cst}
   | Fence _ _ mo → mo ∈ {Relaxed, Release, Acquire, Acq_rel, Consume, Seq_cst}
    | _ → true
    end
```

10 *Effects:* Atomically replaces the value pointed to by `object` or by `this` with the value of `desired`. Memory is affected according to the value of order.

```
C A::operator=(C desired) volatile noexcept;
C A::operator=(C desired) noexcept;
```

11 *Effects:* store(desired)

12 *Returns:* desired

```
C atomic_load(const volatile A* object) noexcept;
C atomic_load(const A* object) noexcept;
C atomic_load_explicit(const volatile A* object, memory_order) noexcept;
C atomic_load_explicit(const A* object, memory_order) noexcept;
C A::load(memory_order order = memory_order_seq_cst) const volatile noexcept;
C A::load(memory_order order = memory_order_seq_cst) const noexcept;
```

13 *Requires:* The order argument shall not be `memory_order_release` nor `memory_order_acq_rel`.

14 *Effects:* Memory is affected according to the value of order.

15 *Returns:* Atomically returns the value pointed to by object or by this.

```
A::operator C () const volatile noexcept;
A::operator C () const noexcept;
```

16 *Effects:* load()

17 *Returns:* The result of load().

```
C atomic_exchange(volatile A* object, C desired) noexcept;
C atomic_exchange(A* object, C desired) noexcept;
C atomic_exchange_explicit(volatile A* object, C desired, memory_order) noexcept;
C atomic_exchange_explicit(A* object, C desired, memory_order) noexcept;
C A::exchange(C desired, memory_order order = memory_order_seq_cst) volatile noexcept;
C A::exchange(C desired, memory_order order = memory_order_seq_cst) noexcept;
```

18 *Effects:* Atomically replaces the value pointed to by `object` or by `this` with `desired`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (1.10).

19 *Returns:* Atomically returns the value pointed to by `object` or by `this` immediately before the effects.

284

```
bool atomic_compare_exchange_weak(volatile A* object, C * expected, C desired) noexcept;
bool atomic_compare_exchange_weak(A* object, C * expected, C desired) noexcept;
bool atomic_compare_exchange_strong(volatile A* object, C * expected, C desired) noexcept;
bool atomic_compare_exchange_strong(A* object, C * expected, C desired) noexcept;
bool atomic_compare_exchange_weak_explicit(volatile A* object, C * expected, C desired,
  memory_order success, memory_order failure) noexcept;
bool atomic_compare_exchange_weak_explicit(A* object, C * expected, C desired,
  memory_order success, memory_order failure) noexcept;
bool atomic_compare_exchange_strong_explicit(volatile A* object, C * expected, C desired,
  memory_order success, memory_order failure) noexcept;
bool atomic_compare_exchange_strong_explicit(A* object, C * expected, C desired,
  memory_order success, memory_order failure) noexcept;
bool A::compare_exchange_weak(C & expected, C desired,
  memory_order success, memory_order failure) volatile noexcept;
bool A::compare_exchange_weak(C & expected, C desired,
  memory_order success, memory_order failure) noexcept;
bool A::compare_exchange_strong(C & expected, C desired,
  memory_order success, memory_order failure) volatile noexcept;
bool A::compare_exchange_strong(C & expected, C desired,
  memory_order success, memory_order failure) noexcept;
bool A::compare_exchange_weak(C & expected, C desired,
  memory_order order = memory_order_seq_cst) volatile noexcept;
bool A::compare_exchange_weak(C & expected, C desired,
  memory_order order = memory_order_seq_cst) noexcept;
bool A::compare_exchange_strong(C & expected, C desired,
  memory_order order = memory_order_seq_cst) volatile noexcept;
bool A::compare_exchange_strong(C & expected, C desired,
  memory_order order = memory_order_seq_cst) noexcept;
```

20 *Requires:* The failure argument shall not be `memory_order_release` nor `memory_order_acq_rel`. The failure argument shall be no stronger than the success argument.

CAS and read-modify-writes C11 and C++11 provide compare-and-swap (CAS) calls that can be used to atomically read and write atomic objects. The C++11 syntax for a CAS on an atomic object `a`, that expects to see a value pointed to by `expected`, and will attempt to write a value `desired` is either `a.compare_exchange_weak(expected, desired, order_succ, order_fail)` or `a.compare_exchange_strong(expected, desired, order_succ, order_fail)`. The standard describes the behaviour of these calls in Section 29.6.5:

21

Effects: Atomically, compares the contents of the memory pointed to by `object` or by `this` for equality with that in `expected`, and if true, replaces the contents of the memory pointed to by `object` or by `this` with that in `desired`, and if false, updates the contents of the memory in `expected` with the contents of the memory pointed to by `object` or by `this`. Further, if the comparison is true, memory is affected according to the value of `success`, and if the comparison is false, memory is affected according to the value of `failure`. When only one `memory_order` argument is supplied, the value of `success` is order, and the value of `failure` is `order` except that a value of `memory_order_acq_rel` shall be replaced by the value `memory_order_acquire` and a value of `memory_order_release` shall be replaced by the value `memory_order_relaxed`. If the operation returns true, these operations are atomic read-modify-write operations (1.10). Otherwise, these operations are atomic load operations.

CAS calls can therefore give rise to two possible sequences of actions. On success they result in a non-atomic read of `expected` followed by an atomic read-modify-write of the atomic location. On failure they result in a non atomic read of `expected`, then an atomic read of the atomic location, followed by a non-atomic write of the value read to `expected`.

22 *Returns:* The result of the comparison.

Blocking CAS The thread-local semantics captures the behaviour of CAS calls by enumerating executions in which the CAS succeeds and manifests as a read-modify-write action, and those where it fails, and generates an atomic load action.

There is one other detail of CAS that is not mentioned in the standard, but impacts on the action type in the memory model. The strong version of CAS does not spuriously fail, so it can be thought of as described in the note in below:

23 [*Note:* For example, the effect of `atomic_compare_exchange_strong` is

```
if (memcmp(object, expected, sizeof(*object)) == 0)
  memcpy(object, \&desired, sizeof(*object));
else
  memcpy(expected, object, sizeof(*object));
```

— *end note*] [*Example:* the expected use of the compare-and-exchange operations is as follows. The compare-and-exchange operations will update expected when another iteration of the loop is needed.

```
expected = current.load();
do {
  desired = function(expected);
} while (!current.compare_exchange_weak(expected, desired));
```

— *end example*]

On the Power and ARM architectures, it is necessary to build a loop for the implementation of the strong C/C++11 CAS (see Chapter 7). This loop uses load-linked store-conditional accesses in order to make sure no other access of the same location occurred between the read and write. The store conditional is sensitive to cache traffic, and may spuriously fail because of accesses to other locations that, for example, share a cache line with the location of the CAS. Consequently, even in the absence of other writes to the location of the CAS, on Power, a strong CAS might block inside the loop. The semantics of C/C++11 must take this into account. The standard does not mention this behaviour, but it is included in the memory model in order to provide guarantees about implementability. The model introduces a new action that represents the blocking outcome for a strong CAS, the blocked read-modify-write:

| BLOCKED_RMW of AID * TID * LOCATION

The thread-local semantics must enumerate this additional outcome for calls to strong-CAS.

24 Implementations should ensure that weak compare-and-exchange operations do not consistently return false unless either the atomic object has value different from expected or there are concurrent modifications to the atomic object.

The weak CAS permits spurious failure that the thread-local semantics must enumerate as well:

25 Remark: A weak compare-and-exchange operation may fail spuriously. That is, even when the contents of memory referred to by `expected` and `object` are equal, it may return false and store back to `expected` the same memory contents that were originally there. [*Note:* This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g., load-locked store-conditional machines. A consequence of spurious failure is that nearly all uses of weak compareand-exchange will be in a loop.

When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable. — *end note*]

26 [*Note:* The `memcpy` and `memcmp` semantics of the compare-and-exchange operations may result in failed comparisons for values that compare equal with operator `==` if the underlying type has padding bits, trap bits, or alternate representations of the same value. Thus, `compare_exchange_strong` should be used with extreme care. On the other hand, `compare_exchange_weak` should converge rapidly. — *end note*]

27 The following operations perform arithmetic computations. The key, operator, and computation correspondence is:

Table 147 — Atomic arithmetic computations

Key	Op	Computation	Key	Op	Computation
`add`	`+`	addition	`sub`	`-`	subtraction
`or`	`|`	bitwise inclusive or	`xor`	`^`	bitwise exclusive or
`and`	`&`	bitwise and			

```
C atomic_fetch_key(volatile A *object, M operand) noexcept;
C atomic_fetch_key(A* object, M operand) noexcept;
C atomic_fetch_key_explicit(volatile A *object, M operand, memory_order order) noexcept;
C atomic_fetch_key_explicit(A* object, M operand, memory_order order) noexcept;
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst) volatile noexcept;
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst) noexcept;
```

28 *Effects:* Atomically replaces the value pointed to by `object` or by `this` with the result of the computation applied to the value pointed to by `object` or by `this` and the given operand. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (1.10).

29 *Returns:* Atomically, the value pointed to by `object` or by `this` immediately before the effects.

30 Remark: For signed integer types, arithmetic is defined to use two's complement representation. There are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.

```
C A::operator op=(M operand) volatile noexcept;
C A::operator op=(M operand) noexcept;
```

31 *Effects:* `fetch_key(operand)`

32 *Returns:* `fetch_key(operand) op operand`

```
C A::operator++(int) volatile noexcept;
C A::operator++(int) noexcept;
```

33 *Returns:* `fetch_add(1)`

```
C A::operator--(int) volatile noexcept;
C A::operator--(int) noexcept;
```

34 *Returns:* `fetch_sub(1)`

```
C A::operator++() volatile noexcept;
C A::operator++() noexcept;
```

35 *Effects:* `fetch_add(1)`

36 *Returns:* `fetch_add(1) + 1`

```
C A::operator--() volatile noexcept;
C A::operator--() noexcept;
```

37 *Effects:* `fetch_sub(1)`

38 *Returns:* `fetch_sub(1) - 1`

29.7 Flag type and operations [atomics.flag]

[elided]

29.8 Fences [atomics.fences]

1 This section introduces synchronization primitives called *fences*. Fences can have acquire semantics, release semantics, or both. A fence with acquire semantics is called an *acquire fence*. A fence with release semantics is called a *release fence*.

Fence synchronisation Section 29.8 of the standard details the synchronisation that is created by release and acquire fences:

2 A release fence A synchronizes with an acquire fence B if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

This passage is relatively intricate. See Chapter 3 for a motivation for the existence of fences and an example of a program that relies on this requirement.

The passage relies on a notion of hypothetical release sequences. Although not an explicit definition, this relation decides the synchronisation behaviour of release fences, so the model does include the relation. In the passage above and the other mention of hypothetical release sequences in the standard, the head is an atomic write. Consequently, the definition is the same as that of release sequences, with the requirement that the head be a release relaxed, so that it only need be an atomic write:

```
let hypothetical_release_sequence_set actions lk mo =
  { (a, b) | ∀ a ∈ actions b ∈ actions |
    is_atomic_action a ∧
    is_write a ∧
    ( (b = a) ∨
      ( (a, b) ∈ mo ∧
        rs_element a b ∧
        ∀ c ∈ actions.
          ((a, c) ∈ mo ∧ (c, b) ∈ mo) ⟶ rs_element a c ) ) }
```

In the model, this synchronisation is formalised directly using the calculated relation that describes the hypothetical release sequence:

```
let release_acquire_fenced_synchronizes_with actions sb asw rf lo rs hrs a b =
  (tid_of a ≠ tid_of b) ∧
([...]
(* fence synchronisation *)
   (is_fence a ∧ is_release a ∧ is_fence b ∧ is_acquire b ∧
    ∃x∈actions y∈actions z∈actions.
```

$(a,\ x)\ \in\ sb \wedge (x,\ y)\ \in\ hrs \wedge (y,\ z)\ \in\ rf \wedge (y,\ b)\ \in\ sb) \vee$
$[\ldots]$

Partially-fenced synchronisation

The standard defines the interaction of release and acquire fences with release and acquire atomics in two passages in Section 29.8. The first describes the case where the acquire side of the usual release-acquire graph is an atomic acquire read, and the release side is a release fence followed by an atomic write:

3 A release fence A synchronizes with an atomic operation B that performs an acquire operation on an atomic object M if there exists an atomic operation X such that A is sequenced before X, X modifies M, and B reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

The second passage describes the other case, where the release thread contains an atomic write-release, and the acquire thread atomically reads, and then has an acquire fence:

4 An atomic operation A that is a release operation on an atomic object M synchronizes with an acquire fence B if there exists some atomic operation X on M such that X is sequenced before B and reads the value written by A or a value written by any side effect in the release sequence headed by A.

The corresponding rules in the model express the execution fragments described in the standard directly using the calculated relations release sequence and hypothetical release sequence:

let *release_acquire_fenced_synchronizes_with actions sb asw rf lo rs hrs a b* =
  (tid_of $a \neq$ tid_of b) $\wedge$
([$\ldots$] (is_fence $a \wedge$ is_release $a \wedge$ is_acquire $b \wedge$
    $\exists x \in actions\ y \in actions.$
      $(a,\ x)\ \in\ sb \wedge (x,\ y)\ \in\ hrs \wedge (y,\ b)\ \in\ rf) \vee$
   (is_release $a \wedge$ is_fence $b \wedge$ is_acquire $b \wedge$
    $\exists x \in actions\ y \in actions.$
      $(a,\ x)\ \in\ rs \wedge (x,\ y)\ \in\ rf \wedge (y,\ b)\ \in\ sb)$

Release and acquire fences Depending on the choice of memory order, fences may be identified by the standard as releases or acquires:

```
extern "C" void atomic_thread_fence(memory_order order) noexcept;
```

5 *Effects:* depending on the value of order, this operation:

- has no effects, if `order == memory_order_relaxed`;
- is an acquire fence, if `order == memory_order_acquire || order == memory_order_consume`;
- is a release fence, if `order == memory_order_release`;
- is both an acquire fence and a release fence, if `order == memory_order_acq_rel`;
- is a sequentially consistent acquire and release fence, if `order == memory_order_seq_cst`.

The model's is-release and is-acquire functions capture this requirement:

```
let is_release a =
  match a with
  | Store _ _ mo _ _ → mo ∈ {Release, Seq_cst}
  | RMW _ _ mo _ _ _ → mo ∈ {Release, Acq_rel, Seq_cst}
  | Fence _ _ mo → mo ∈ {Release, Acq_rel, Seq_cst}
  | _ → false
  end

let is_acquire a =
  match a with
  | Load _ _ mo _ _ → mo ∈ {Acquire, Seq_cst}
  | RMW _ _ mo _ _ _ → mo ∈ {Acquire, Acq_rel, Seq_cst}
  | Fence _ _ mo → mo ∈ {Acquire, Consume, Acq_rel, Seq_cst}
  | _ → false
  end
```

```
extern "C" void atomic_signal_fence(memory_order order) noexcept;
```

6 *Effects:* equivalent to `atomic_thread_fence(order)`, except that the resulting ordering constraints are established only between a thread and a signal handler executed in the same thread.

7 *Note:* `atomic_signal_fence` can be used to specify the order in which actions performed by the thread become visible to the signal handler.

8 *Note:* compiler optimizations and reorderings of loads and stores are inhibited in the same way as with `atomic_thread_fence`, but the hardware fence instructions that `atomic_thread_fence` would have inserted are not emitted.

30 Thread support library [thread]

30.1 General [thread.general]

1 The following subclauses describe components to create and manage threads (1.10), perform mutual exclusion, and communicate conditions and values between threads, as summarized in Table 148.

Table 148 — Thread support library summary

	Subclause	**Header(s)**
30.2	Requirements	
30.3	Threads	`<thread>`
30.4	Mutual exclusion	`<mutex>`
30.5	Condition variables	`<condition_variable>`
30.6	Futures	`<future>`

30.2 Requirements [thread.req]

[elided]

30.2.5 Requirements for Lockable types [thread.req.lockable]

[elided]

30.2.5.2 `BasicLockable` requirements [thread.req.lockable.basic]

1 A type `L` meets the `BasicLockable` requirements if the following expressions are well-formed and have the specified semantics (`m` denotes a value of type `L`).

`m.lock()`

2 *Effects:* Blocks until a lock can be acquired for the current execution agent. If an exception is thrown then a lock shall not have been acquired for the current execution agent.

`m.unlock()`

3 *Requires:* The current execution agent shall hold a lock on `m`.

4 *Effects:* Releases a lock on `m` held by the current execution agent.

Throws: Nothing.

[elided]

30.3 Threads [thread.threads]

1 30.3 describes components that can be used to create and manage threads. [*Note:* These threads are intended to map one-to-one with operating system threads. — *end note*]

Header `<thread>` synopsis

[elided]

30.3.1 Class thread [thread.thread.class]

1 The class `thread` provides a mechanism to create a new thread of execution, to join with a thread (i.e., wait for a thread to complete), and to perform other operations that manage and query the state of a thread. A `thread` object uniquely represents a particular thread of execution. That representation may be transferred to other `thread` objects in such a way that no two `thread` objects simultaneously represent the same thread of execution. A thread of execution is detached when no `thread` object represents that thread. Objects of class thread can be in a state that does not represent a thread of execution. [*Note:* A `thread` object does not represent a thread of execution after default construction, after being moved from, or after a successful call to `detach` or `join`. — *end note*]

[elided]

30.3.1.2 thread constructors [thread.thread.constr]

```
thread() noexcept;
```

1 *Effects:* Constructs a `thread` object that does not represent a thread of execution.

2 *Postcondition:* `get_id() == id()`

> **Additional synchronises with** The examples throughout this thesis, and the language presented in the thread-local semantics, provide a simple parallel composition syntax that differs from that of the standard. Nonetheless, the impact of thread construction on the order of accesses to memory must be recorded, regardless of syntax. The standard provides its thread construction syntax here:

```
template <class F, class ...Args> explicit thread(F&& f, Args&&...
args);
```

3 *Requires:* `F` and each `Ti` in `Args` shall satisfy the `MoveConstructible` requirements. `INVOKE (DECAY_COPY ( std::forward<F>(f)), DECAY_COPY (std::forward<Args>(args))...)` (20.8.2) shall be a valid expression.

4 *Effects:* Constructs an object of type `thread`. The new thread of execution executes `INVOKE (DECAY_COPY ( std::forward<F>(f)), DECAY_COPY (std::forward<Args>(args))...)` with the calls to `DECAY_COPY` being evaluated in the constructing thread. Any return value from this invocation is ignored. [*Note:* This implies that any exceptions not thrown from the invocation of the copy of `f` will be thrown in the constructing thread, not the new thread. — *end note*] If the invocation of `INVOKE (DECAY_COPY ( std::forward<F>(f)), DECAY_COPY (std::forward<Args>(args))...)` terminates with an uncaught exception, `std::terminate` shall be called.

5 *Synchronization:* The completion of the invocation of the constructor synchronizes with the beginning of the invocation of the copy of `f`.

> The thread call executes the expressions provided in `f` as a new thread. The standard dictates that there is synchronises-with ordering from the call of the thread constructor in the parent, to the start of the child thread. In the model, this ordering is collected in the additional synchronises-with relation, asw. The thread-local semantics creates this ordering from the actions preceding a parallel composition, to the sequenced-before minima in each child thread. The *well_formed_threads* predicate

does not impose much restriction on additional synchronises-with, just that it is a relation over the actions that relates actions from different threads:

relation_over *Xo.actions Xo.asw* ∧ interthread *Xo.actions Xo.asw*

The calculation of the synchronises-with relation includes all of additional-synchronises-with:

let *release_acquire_fenced_synchronizes_with actions sb asw rf lo rs hrs a b* =
(tid_of $a \neq$ tid_of b) ∧
(`(* thread sync *)`
(a, b) ∈ *asw* ∨
[...])

6 *Postconditions:* `get_id() != id()`. `*this` represents the newly started thread.

7 *Throws:* `system_error` if unable to start the new thread.

8 *Error conditions:* — `resource_unavailable_try_again` — the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

```
thread(thread&& x) noexcept;
```

9 *Effects:* Constructs an object of type `thread` from `x`, and sets `x` to a default constructed state.

10 *Postconditions:* `x.get_id() == id()` and `get_id()` returns the value of `x.get_id()` prior to the start of construction.

30.3.1.3 thread destructor [thread.thread.destr]

[elided]

30.4 Mutual exclusion [thread.mutex]

1 This section provides mechanisms for mutual exclusion: mutexes, locks, and call once. These mechanisms ease the production of race-free programs (1.10).

Header `<mutex>` synopsis

[elided]

30.4.1 Mutex requirements [thread.mutex.requirements]

30.4.1.1 In general [thread.mutex.requirements.general]

1 A mutex object facilitates protection against data races and allows safe synchronization of data between execution agents (30.2.5). An execution agent *owns* a mutex from the time it successfully calls one of the lock functions until it calls unlock. Mutexes can be either recursive or non-recursive, and can grant simultaneous ownership to one or many execution agents. The mutex types supplied by the standard library provide exclusive ownership semantics: only one thread may own the mutex at a time. Both recursive and non-recursive mutexes are supplied.

30.4.1.2 Mutex types [thread.mutex.requirements.mutex]

1 The *mutex types* are the standard library types `std::mutex`, `std::recursive_mutex`, `std::timed_mutex`, and `std::recursive_timed_mutex`. They shall meet the requirements set out in this section. In this description, `m` denotes an object of a mutex type.

2 The mutex types shall meet the `Lockable` requirements (30.2.5.3).

3 The mutex types shall be `DefaultConstructible` and `Destructible`. If initialization of an object of a mutex type fails, an exception of type `system_error` shall be thrown. The mutex types shall not be copyable or movable.

4 The error conditions for error codes, if any, reported by member functions of the mutex types shall be:

- `resource_unavailable_try_again` — if any native handle type manipulated is not available.
- `operation_not_permitted` — if the thread does not have the privilege to perform the operation.
- `device_or_resource_busy` — if any native handle type manipulated is already locked.
- `invalid_argument` — if any native handle type manipulated as part of mutex construction is incorrect.

Lock order Total orders over all of the lock and unlock actions on mutex locations are introduced below:

5 The implementation shall provide lock and unlock operations, as described below. For purposes of determining the existence of a data race, these behave as atomic operations (1.10). The lock and unlock operations on a single mutex shall appear to occur in a single total order. [*Note:* this can be viewed as the modification order (1.10) of the mutex. — *end note*] [*Note:* Construction and destruction of an object of a mutex type need not be thread-safe; other synchronization should be used to ensure that mutex objects are initialized and visible to other threads. — *end note*]

The model captures these per-location orders as a single relation that is simply the union over all locations in the execution. The *locks_only_consistent_lo* predicate checks that the lock order relation behaves accordingly. Given the note in the passage above, the predicate is similar to the restriction on modification order, except that in this case, the relation is over the set of all lock and unlock actions at each mutex location, and the predicate checks that lock order and happens-before agree:

```
let locks_only_consistent_lo (Xo, Xw, ("hb", hb) :: _) =
    relation_over Xo.actions Xw.lo ∧
    isTransitive Xw.lo ∧
    isIrreflexive Xw.lo ∧
    ∀ a ∈ Xo.actions b ∈ Xo.actions.
      ((a, b) ∈ Xw.lo ⟶ ¬ ((b, a) ∈ hb)) ∧
      ( ((a, b) ∈ Xw.lo ∨ (b, a) ∈ Xw.lo)
        =
        ( (¬ (a = b)) ∧
          (is_lock a ∨ is_unlock a) ∧
          (is_lock b ∨ is_unlock b) ∧
          (loc_of a = loc_of b) ∧
          is_at_mutex_location Xo.lk a
        )
      )
```

Together, these restrictions require lock order to be an irreflexive total order over each mutex location. The restriction that makes lock order agree with happens-before, although not explicitly stated, is suggested by the note that draws a similarity between lock order and modification order.

The behaviour of locks and unlocks is governed by the lock order. The consistency predicate defines how locks may behave in an execution with its *locks_only_consistent_locks* conjunct. To justify this predicate, first consider the semantics of locks as defined in the requirements, effects and postconditions of the lock and unlock calls given below:

6 The expression `m.lock()` shall be well-formed and have the following semantics:

The requirement over calls to `m.lock()` is provided below:

7 *Requires:* If `m` is of type `std::mutex` or `std::timed_mutex`, the calling thread does not own the mutex.

If the programmer violates this requirement, their program has undefined behaviour. The ownership of the mutex is embedded in the lock ordering over the mutex actions. Consequently, this source of undefined behaviour are identified in the model using the lock order. The model defines what it is to be a good mutex:

```
let locks_only_good_mutex_use actions lk sb lo a =
     (* violated requirement: The calling thread shall own the mutex.
*)
   ( is_unlock a
     ⟶
     ( ∃ al ∈ actions.
        is_successful_lock al ∧ (al, a) ∈ sb ∧ (al, a) ∈ lo ∧
        ∀ au ∈ actions.
          is_unlock au ⟶ ¬ ((al, au) ∈ lo ∧ (au, a) ∈ lo)
      )
   ) ∧
         (* violated requirement: The calling thread does not own the
mutex. *)
   ( is_lock a
     ⟶
     ∀ al ∈ actions.
       is_successful_lock al ∧ (al, a) ∈ sb ∧ (al, a) ∈ lo
       ⟶
```

$\exists\ au \in actions.$
is_unlock $au \wedge (al,\ au) \in lo \wedge (au,\ a) \in lo$
)

The second conjunct requires every pair of locks where the first is successful to have an intervening unlock. This implies that the program should not lock a mutex twice, as required above.

Lock outcome This section defines the syntax of lock and unlock calls as `m.lock()` and `m.unlock()` respectively. The calls take no arguments, so the lock and unlock actions do not take any value or memory order parameters. Locks are called on an object, so lock actions must take a location parameter. In addition, the standard says that lock calls can block:

8 *Effects:* Blocks the calling thread until ownership of the mutex can be obtained for the calling thread.

A lock can block forever, and as the memory model deals with whole executions, this behaviour is modeled as a special blocked-lock action. The lock-action type takes a parameter that identifies whether, in the execution, the lock remains blocked for ever, or successfully locks the mutex. The parameter is called the lock outcome:

```
type LOCK_OUTCOME =
    LOCKED
  | BLOCKED
```

The semantics of blocked locks and blocked read-modify-write actions is defined in terms of sequenced before. A consistent execution cannot have a blocked action with further actions sequenced after it — that is, the thread must block, and not allow program-order-later code to be part of the execution. This restriction is captured by the *blocking_observed* predicate:

```
let blocking_observed actions sb =
    (∀ a ∈ actions.
      (is_blocked_rmw a ∨ is_blocked_lock a)
      ⟶
      ¬ (∃ b ∈ actions. (a, b) ∈ sb))
```

9 *Postcondition:* The calling thread owns the mutex.

10 *Return type:* `void`

Mutex synchronisation Synchronisation resulting from locks is described in terms of the lock order relation. The standard describes the specifics of mutex synchronisation, first on locks:

11 *Synchronization:* Prior `unlock()` operations on the same object shall *synchronize with* (1.10) this operation.

12 *Throws:* `system_error` when an exception is required (30.2.2).

13 *Error conditions:*

- `operation_not_permitted` — if the thread does not have the privilege to perform the operation.
- `resource_deadlock_would_occur` — if the implementation detects that a deadlock would occur.
- `device_or_resource_busy` — if the mutex is already locked and blocking is not possible.

14 The expression `m.try_lock()` shall be well-formed and have the following semantics:

15 *Requires:* If `m` is of type `std::mutex` or `std::timed_mutex`, the calling thread does not own the mutex.

16 *Effects:* Attempts to obtain ownership of the mutex for the calling thread without blocking. If ownership is not obtained, there is no effect and `try_lock()` immediately returns. An implementation may fail to obtain the lock even if it is not held by any other thread. [*Note:* This spurious failure is normally uncommon, but allows interesting implementations based on a simple compare and exchange (Clause 29). — *end note*] An implementation should ensure that `try_lock()` does not consistently return `false` in the absence of contending mutex acquisitions.

17 *Return type:* `bool`

18 *Returns:* `true` if ownership of the mutex was obtained for the calling thread, otherwise `false`.

19 *Synchronization:* If `try_lock()` returns `true`, prior `unlock()` operations on the same object *synchronize with* (1.10) this operation. [*Note:* Since `lock()` does not synchronize with a failed subsequent `try_lock()`, the visibility rules are weak enough that

little would be known about the state after a failure, even in the absence of spurious failures. — *end note*]

20 *Throws:* Nothing.

21 The expression `m.unlock()` shall be well-formed and have the following semantics:

A requirement on the use of unlock is given below:

22 *Requires:* The calling thread shall own the mutex.

Again, violating this requirement leads to undefined behaviour, and violations are identified in the model using the lock order by defining what it is to be a good mutex:

```
let locks_only_good_mutex_use actions lk sb lo a =
     (* violated requirement: The calling thread shall own the mutex.
*)
   ( is_unlock a
     ⟶
     ( ∃ al ∈ actions.
        is_successful_lock al ∧ (al, a) ∈ sb ∧ (al, a) ∈ lo ∧
        ∀ au ∈ actions.
          is_unlock au ⟶ ¬ ((al, au) ∈ lo ∧ (au, a) ∈ lo)
      )
   ) ∧
         (* violated requirement: The calling thread does not own the
mutex. *)
   ( is_lock a
     ⟶
     ∀ al ∈ actions.
       is_successful_lock al ∧ (al, a) ∈ sb ∧ (al, a) ∈ lo
       ⟶
       ∃ au ∈ actions.
         is_unlock au ∧ (al, au) ∈ lo ∧ (au, a) ∈ lo
   )
```

The first conjunct requires every unlock to be immediately preceded in lock order by a lock that is sequenced before the unlock. This implies that the program should not unlock a mutex twice, as required above. Faulty mutex accesses are then identified as the set of accesses that fail the *good_mutex_use* requirement:

$$\begin{array}{l}\mathsf{let}\ \mathit{locks\_only\_bad\_mutexes}\ (\mathit{Xo},\ \mathit{Xw},\ \_)\ = \\ \quad \{\ a \mid \forall\ a \in \mathit{Xo.actions} \mid \\ \quad\quad \neg\ (\text{locks\_only\_good\_mutex\_use}\ \mathit{Xo.actions}\ \mathit{Xo.lk}\ \mathit{Xo.sb}\ \mathit{Xw.lo}\ a)\}\end{array}$$

23 *Effects:* Releases the calling thread's ownership of the mutex.

The formal model does not model ownership of mutexes, but instead identifies orders of mutex actions that correctly use ownership. The requirements, effects and postconditions of the lock and unlock calls above forbid executions with a lock order where two successful locks on the same mutex are adjacent in lo, and the *locks_only_consistent_locks* predicate in the model captures that:

$$\begin{array}{l}\mathsf{let}\ \mathit{locks\_only\_consistent\_locks}\ (\mathit{Xo},\ \mathit{Xw},\ \_)\ = \\ \quad (\forall\ (a,\ c) \in \mathit{Xw.lo}. \\ \quad\quad \text{is\_successful\_lock}\ a \wedge \text{is\_successful\_lock}\ c \\ \quad\quad \longrightarrow \\ \quad\quad (\exists\ b \in \mathit{Xo.actions}.\ \text{is\_unlock}\ b \wedge (a,\ b) \in \mathit{Xw.lo} \wedge (b,\ c) \in \mathit{Xw.lo}))\end{array}$$

According to the standard, the first lock acquires ownership of the mutex, and the second violates the specification by acquiring ownership that was not relinquished with an intervening unlock. Recall that the thread-local semantics generates blocked lock actions at each lock because locks can arbitrarily block on some targets like Power and ARM processors. These block actions cover the cases where locks would block because of deadlock.

24 *Return type:* `void`

And then on unlocks:

25 *Synchronization:* This operation *synchronizes with* (1.10) subsequent lock operations that obtain ownership on the same object.

In the formal model, blocked locks do not engage in synchronisation. The disjunct that captures mutex synchronisation therefore turns every edge in lock order from an unlock to a successful lock into a synchronises-with edge:

```
let release_acquire_fenced_synchronizes_with actions sb asw rf lo rs hrs a b =
  (tid_of a ≠ tid_of b) ∧
([...]   (* mutex sync *)
    (is_unlock a ∧ is_successful_lock b ∧ (a, b) ∈ lo) ∨
[...])
```

26 *Throws:* Nothing.

30.4.1.2.1 Class mutex [thread.mutex.class]

```
namespace std {
  class mutex {
  public:
    constexpr mutex() noexcept;
    ~mutex();

    mutex(const mutex&) = delete;
    mutex& operator=(const mutex&) = delete;

    void lock();
    bool try_lock();
    void unlock();

    typedef implementation-defined native_handle_type; // See 30.2.3
    native_handle_type native_handle();                // See 30.2.3
  };
}
```

1 The class `mutex` provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a mutex object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`) until the owning thread has released ownership with a call to `unlock()`.

2 [*Note:* After a thread `A` has called `unlock()`, releasing a mutex, it is possible for another thread `B` to lock the same mutex, observe that it is no longer in use, unlock it, and destroy it, before thread `A` appears to have returned from its unlock call. Implementations are required to handle such scenarios correctly, as long as thread `A`x doesn't access the mutex

after the unlock call returns. These cases typically occur when a reference-counted object contains a mutex that is used to protect the reference count. — *end note*]

3 The class `mutex` shall satisfy all the `Mutex` requirements (30.4.1). (Clause 9).

4 [*Note:* A program may deadlock if the thread that owns a `mutex` object calls `lock()` on that object. If the implementation can detect the deadlock, a `resource_deadlock_would_occur` error condition may be observed. — *end note*]

It shall be a standard-layout class

5 The behavior of a program is undefined if it destroys a `mutex` object owned by any thread or a thread terminates while owning a `mutex` object.

30.4.1.2.2 Class recursive_mutex [thread.mutex.recursive]

[elided]

Appendix B

The C++11 standard: side-by-side comparison, following the model

This appendix relates the formal model from Section 3.10 to the text of the C++11 standard [30], showing that there is a tight correspondence between the two. The memory model used in both C11 and C++11 was defined by the C++11 standardisation committee, and C11 has adopted it with little more than editorial adjustments, so the correspondence with C++ argued in this section serves for C as well.

This appendix focusses on the standard as ratified (including defect reports): errors in drafts that were subsequently fixed, and issues that remain in the standard are described in Chapter 5.

The structure of this appendix roughly follows the structure of Chapter 3: there is (inevitably) repetition, and there are no new model definitions. Parts of the consistency predicate will be discussed throughout, and then collected together later in the appendix, together with the explanation of the remaining parts of the predicate. Finally the top-level treatment of undefined behaviour will be discussed.

Each part of the mathematical formalism is justified by quotes from the standard, which are referenced by a section and paragraph number separated by the letter "p", so paragraph two in section 1.10 is referred to as **1.10p2**. Parts of the standard that are not formalised directly, but are relavant to the formal model are also discussed.

B.1 The pre-execution type

Sets of pre-executions are generated by the thread-local semantics. They correspond to the execution of each path of control flow, with no restriction on the values read. The pre-execution type in the memory model is:

```
type PRE_EXECUTION =
  ⟨| actions : SET (ACTION);
```

$threads$: SET (TID);
lk : LOCATION $\rightarrow$ LOCATION_KIND;
sb : SET (ACTION $*$ ACTION) ;
asw : SET (ACTION $*$ ACTION) ;
dd : SET (ACTION $*$ ACTION) ;
$\rangle$

This section will take each of the fields in the pre-execution record and justify it with text from the standard. The set of restrictions that the model imposes on pre-executions will be identified and collected in the *well_formed_threads* predicate at the end of the section.

B.1.1 The action type

The atomics and mutexes are defined in Sections 29 and 30 of the standard. Calls to atomic objects, mutex objects, and fences allow the programmer to induce synchronisation in their program and avoid data races. The syntax of these calls is provided below:

30.4.1.2.1, 29.6.5, 29.8p5

```
class mutex {
void lock();
void unlock();
}


C A::load(memory_order order = memory_order_seq_cst) const noexcept;
void A::store(C desired, memory_order order = memory_order_seq_cst) noexcept;
bool A::compare_exchange_weak(C& expected, C desired, memory_order success,
memory_order failure) noexcept;
bool A::compare_exchange_strong(C& expected, C desired, memory_order success,
memory_order failure) noexcept;


extern "C" void atomic_thread_fence(memory_order order) noexcept;
```

All of the functions except fence calls are called either on a mutex or an atomic object. The parameters and behaviour of each call will be described in due course. The standard describes the behaviour of these calls in terms of the following "operations" on memory. Writes to memory are "modifications" or, more generally "side effects". Reads are described as "value computations", or more generally "evaluations". This language is introduced in **1.9p12**:

1.9p12

> Accessing an object designated by a volatile glvalue (3.10), modifying an object, calling a library I/O function, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment. Evaluation of an expression (or a subexpression) in general includes both value computations (including determining the identity of an object for glvalue evaluation and fetching a value previously assigned to an object for prvalue evaluation) and initiation of side effects. [...]

The formal model describes the behaviour of accesses of memory in terms of memory interactions called *memory actions*, that correspond to "operations" in terms of the standard. Despite mention of the "state of the execution environment" the rest of the standard discusses the behaviour of programs in terms of relations over actions where there is no obvious interpretation of an instantaneous state. The formal model defines the action type. It captures the modifications and value computations (or stores and loads), on non-atomic and atomic objects, the locks and unlocks on mutex objects, and memory fences present in an execution. The action type is presented below:

```
type ACTION =
  | LOCK of AID * TID * LOCATION * LOCK_OUTCOME
  | UNLOCK of AID * TID * LOCATION
  | LOAD of AID * TID * MEMORY_ORDER * LOCATION * CVALUE
  | STORE of AID * TID * MEMORY_ORDER * LOCATION * CVALUE
  | RMW of AID * TID * MEMORY_ORDER * LOCATION * CVALUE * CVALUE
  | FENCE of AID * TID * MEMORY_ORDER
  | BLOCKED_RMW of AID * TID * LOCATION
```

The memory model is stated in terms of candidate executions in which each memory action is identified by a unique identifier and carries the identifier of thread that produced the action. Each memory action above has an *aid* and *tid* that represent this information. The uniqueness of action identifiers is ensured in the *well_formed_threads* predicate by requiring the following action identifier projection function to be injective:

```
let aid_of a =
  match a with
  | Lock aid _ _ _ → aid
  | Unlock aid _ _ → aid
  | Load aid _ _ _ _ → aid
  | Store aid _ _ _ _ → aid
  | RMW aid _ _ _ _ _ → aid
  | Fence aid _ _ → aid
  | Blocked_rmw aid _ _ → aid
  end
```

Locations and location-kinds

Many of the disjuncts that make up the action type take a location as a parameter. Section **1.7** introduces memory locations, and their relationship to objects:

1.7p3

> A *memory location* is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width. [...]Two threads of execution (1.10) can update and access separate memory locations without interfering with each other.

The memory model does not deal with objects, but instead keeps track of accesses to locations. A map from locations to *location-kinds* captures whether a particular location represents a non-atomic, atomic or mutex object:

```
type LOCATION_KIND =
    MUTEX
  | NON_ATOMIC
  | ATOMIC
```

In Sections **29** and **30**, the standard introduces atomic acesses and mutexes respectively. **29.6.5p2** requires that we call the atomic accessor functions only on objects of atomic type:

29.6.5p2

> In the following operation definitions [...] an A refers to one of the atomic types.

Similarly, **30.4.1.2p1** requires that mutex accessor function act only on objects of mutex type:

30.4.1.2p1

> [...] In this description, m denotes an object of a mutex type.

These requirements guarantee that pre-executions will not feature a lock at an atomic location, for instance. The memory model captures this guarantee in the *actions_respect_location_kinds* predicate on pre-executions. It requires that lock and unlock actions act on mutex locations, that atomic actions act on atomic locations, and it forbids non-atomic loads on atomic locations — there does not appear to be a way to non-atomically load from an atomic location (see Section 5 for further discussion). Writes to atomic objects that are the result of initialisation are non-atomic, so non-atomic stores at atomic locations are allowed. Non-atomic locations may only be accessed by non-atomic loads and stores. The *actions_respect_location_kinds* predicate is given below:

```
let actions_respect_location_kinds actions lk =
 ∀ a ∈ actions. match a with
    | Lock _ _ l _ → lk l = Mutex
    | Unlock _ _ l → lk l = Mutex
    | Load _ _ mo l _ →
        (mo = NA ∧ lk l = Non_Atomic) ∨ (mo ≠ NA ∧ lk l = Atomic)
    | Store _ _ mo l _ →
        (mo = NA ∧ lk l = Non_Atomic) ∨ lk l = Atomic
    | RMW _ _ _ l _ _ → lk l = Atomic
    | Fence _ _ _ → true
    | Blocked_rmw _ _ l → lk l = Atomic
    end
```

Locks

Section **30.4.1.2** defines the syntax of lock and unlock calls as `m.lock()` and `m.unlock()` respectively. The calls take no arguments, so the lock and unlock actions do not take any value or memory order parameters. Locks are called on an object, so lock actions must take a location parameter. In addition, the standard says that lock calls can block:

30.4.1.2p8

> Effects: Blocks the calling thread until ownership of the mutex can be obtained for the calling thread.

The memory model does not explicitly model ownership of mutex locations; instead it maintains a relation over the locks on a particular mutex, discussed in Section B.2. A lock can block forever, and as the memory model deals with whole executions, this behaviour is modeled as a special blocked-lock action. The lock-action type takes a parameter that identifies whether, in the execution, the lock remains blocked for ever, or successfully locks the mutex. The parameter is called the lock outcome:

```
type LOCK_OUTCOME =
    LOCKED
  | BLOCKED
```

Memory order

Loads, stores, read-modify-writes and fences all take a memory-order parameter. Section **29.3** specifies the options with the following enumeration:

```
namespace std {
       typedef enum memory_order {
         memory_order_relaxed, memory_order_consume, memory_order_acquire,
         memory_order_release, memory_order_acq_rel, memory_order_seq_cst
       } memory_order;
}
```

In the model, the *memory_order* type captures these orders, adding non-atomic ordering, *NA*, to the list to unify atomic and non-atomic actions:

```
type MEMORY_ORDER =
  | NA
  | SEQ_CST
  | RELAXED
  | RELEASE
  | ACQUIRE
  | CONSUME
  | ACQ_REL
```

Not all actions can be given every memory order. The *well_formed_actions* predicate,

described in Section B.1.2, will define which memory orders are allowed for each variety of action.

Loads and stores

Atomic loads and stores are introduced in **29.6.5**, where the standard specifies the syntax `a.load(order)` for a load of an atomic object `a` with a memory order `order`, and `a.store(1,order)` for a store of `a` of value `1` with order `order`. The object that loads and stores access is represented in the model as a location, so the corresponding disjuncts in the action type contain a location parameter. The calls to load and store are also parameterised by a memory order, and the disjuncts of the type are parameterised similarly. Store calls specify the value that is written to memory, and loads return a value. In the formal model, both are parameterised by a value: stores by the value written, and loads by the value read.

Rather than have separate disjuncts for non-atomic loads and stores, the formal model represents non-atomic accesses with the same disjuncts as atomics, adding a non-atomic disjunct to the memory order type.

CAS and read-modify-writes

C11 and C++11 provide compare-and-swap (CAS) calls that can be used to atomically read and write atomic objects. The C++11 syntax for a CAS on an atomic object `a`, that expects to see a value pointed to by `expected`, and will attempt to write a value `desired` is either `a.compare_exchange_weak(expected, desired, order_succ, order_fail)` or `a.compare_exchange_strong(expected, desired, order_succ, order_fail)`. The standard describes the behaviour of these calls in section **29.6.5**:

29.6.5p21

> Effects: Atomically, compares the contents of the memory pointed to by `object` or by `this` for equality with that in `expected`, and if true, replaces the contents of the memory pointed to by `object` or by `this` with that in `desired`, and if false, updates the contents of the memory in `expected` with the contents of the memory pointed to by `object` or by `this`. Further, if the comparison is true, memory is affected according to the value of `success`, and if the comparison is false, memory is affected according to the value of `failure`. [...]If the operation returns `true`, these operations are atomic read-modify-write operations (1.10). Otherwise, these operations are atomic load operations.

CAS calls can therefore give rise to two possible sequences of actions. On success they result in a non-atomic read of `expected` followed by an atomic read-modify-write of the

atomic location. On failure they result in a non atomic read of `expected`, then an atomic read of the atomic location, followed by a non-atomic write of the value read to `expected`. The read-modify-write action is parameterised by a memory order and two values, the expected value and the desired one.

The thread-local semantics captures the behaviour of CAS calls by enumerating executions in which the CAS succeeds and manifests as a read-modify-write action, and those where it fails, and generates an atomic load action. The weak CAS permits spurious failure, that the thread-local semantics must enumerate as well:

29.6.5p25

> Remark: A weak compare-and-exchange operation may fail spuriously. That is, even when the contents of memory referred to by `expected` and `object` are equal, it may return false and store back to `expected` the same memory contents that were originally there. [*Note:* This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g., load-locked store-conditional machines. A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will be in a loop. [...]— *end note*]

There is one other detail of CAS that is not mentioned in the standard, but impacts on the action type in the memory model. The strong version of CAS does not spuriously fail, so it can be thought of as described in the note in Section **29.6.5**:

29.6.5p23

> [*Note:* For example, the effect of `atomic_compare_exchange_strong` is
>
> ```
> if (memcmp(object, expected, sizeof(*object)) == 0)
> memcpy(object, &desired, sizeof(*object));
> else
> memcpy(expected, object, sizeof(*object));
> ```
>
> — *end note*] [...]

On the Power and ARM architectures, it is necessary to build a loop for the implementation of the strong C/C++11 CAS (see Chapter 7). This loop uses load-linked store-conditional accesses in order to make sure no other access of the same location occurred between the read and write. The store conditional is sensitive to cache traffic, and may spuriously fail because of accesses to other locations that, for example, share a cache line with the location of the CAS. Consequently, even in the absence of other

writes to the location of the CAS, on Power, a strong CAS might block inside the loop. The semantics of C/C++11 must take this into account. The standard does not mention this behaviour, but it is included in the memory model in order to provide guarantees about implementability. The model introduces a new action that represents the blocking outcome for a strong CAS, the blocked read-modify-write:

| BLOCKED_RMW of AID * TID * LOCATION

The thread-local semantics must enumerate this additional outcome for calls to strong-CAS.

Fences

A fence call can be made with the syntax: `atomic_thread_fence(order)`. They take only one parameter, a memory order, and are not associated with an object or a location. As a consequence, the disjunct of the action type for fences does not include a location, only a memory order.

B.1.2 Well-formed actions

The *well_formed_action* predicate restricts the memory orders that can be given to an atomic action, and corresponds to requirements expressed in the standard. Section **29.6.5** states the following of loads and stores:

29.6.5p9, 29.6.5p13

> `void A::store(C desired, memory_order order = memory_order_seq_cst) noexcept;`
> Requires: The order argument shall not be `memory_order_consume`, `memory_order_acquire`, nor `memory_order_acq_rel`.
>
> `C A::load(memory_order order = memory_order_seq_cst) const noexcept;`
> Requires: The order argument shall not be `memory_order_release` nor `memory_order_acq_rel`.

The standard makes no restriction on the memory orders of fences or read-modify-write accesses resulting from a CAS. A failing CAS gives rise to a load action, and the standard does restrict the memory order that applies to this read. Recall that the memory order of the load is given in the `failure` argument:

29.6.5p20

> *Requires*: The `failure` argument shall not be `memory_order_release` nor `memory_order_acq_rel`. [...]

The model differentiates between the non-atomic and atomic loads by means of the non-atomic memory order. As a consequence, neither read-modify-writes nor fences are allowed to take the non-atomic memory order. These restrictions are enforced by the *well_formed_action* predicate:

```
let well_formed_action a =
  match a with
    | Load _ _ mo _ _ → mo ∈ {NA, Relaxed, Acquire, Seq_cst, Consume}
    | Store _ _ mo _ _ → mo ∈ {NA, Relaxed, Release, Seq_cst}
    | RMW _ _ mo _ _ _ → mo ∈ {Relaxed, Release, Acquire, Acq_rel, Seq_cst}
    | Fence _ _ mo → mo ∈ {Relaxed, Release, Acquire, Acq_rel, Consume, Seq_cst}
    | _ → true
    end
```

is_release, *is_acquire* Depending on the choice of memory order, some memory accesses may be identified by the standard as releases or acquires. Now that loads, stores, read-modify-writes and fences have been introduced, these definitions can be given. The standard defines which loads, stores and read-modify-writes are releases and acquires in Section **29.3**:

29.3p1

> The enumeration `memory_order` specifies the detailed regular (non-atomic) memory synchronization order as defined in 1.10 and may provide for operation ordering. Its enumerated values and their meanings are as follows:
>
> - `memory_order_relaxed`: no operation orders memory.
> - `memory_order_release`, `memory_order_acq_rel`, and `memory_order_seq_cst`: a store operation performs a release operation on the affected memory location.
> - `memory_order_consume`: a load operation performs a consume operation on the affected memory location.
> - `memory_order_acquire`, `memory_order_acq_rel`, and `memory_order_seq_cst`: a load operation performs an acquire operation on the affected memory location.
>
> [...]

Whether fences are releases or acquires is defined in Section **29.8** of the standard:

29.8p5

> Effects: depending on the value of `order`, this operation:
>
> - has no effects, if `order == memory_order_relaxed`;
> - is an acquire fence, if `order == memory_order_acquire || order == memory_order_consume`;
> - is a release fence, if `order == memory_order_release`;
> - is both an acquire fence and a release fence, if `order == memory_order_acq_rel`;
> - is a sequentially consistent acquire and release fence, if `order == memory_order_seq_cst`.

The model includes functions that take an action and judge whether it is a release or an acquire. The definitions collect together the requirements from the quotes above:

```
let is_release a =
  match a with
  | Store _ _ mo _ _ → mo ∈ {Release, Seq_cst}
  | RMW _ _ mo _ _ _ → mo ∈ {Release, Acq_rel, Seq_cst}
```

```
    | Fence _ _ mo  →  mo ∈ {Release, Acq_rel, Seq_cst}
    | _  →  false
    end

let is_acquire a  =
    match a with
    | Load _ _ mo _ _  →  mo ∈ {Acquire, Seq_cst}
    | RMW _ _ mo _ _ _  →  mo ∈ {Acquire, Acq_rel, Seq_cst}
    | Fence _ _ mo  →  mo ∈ {Acquire, Consume, Acq_rel, Seq_cst}
    | _  →  false
    end
```

B.1.3 Sequenced before

The pre-execution record contains the sequenced-before relation that records thread-local syntactically-imposed program-order. The standard introduces sequenced before in Section **1.9**:

1,9p13

> *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread (1.10), which induces a partial order among those evaluations. [...]

The memory model deals with memory accesses rather than evaluations, so the thread-local semantics calculates this relation over the memory actions. The standard requires the relation be an asymmetric, transitive partial order, relating only evaluations from the same thread. The *well_formed_threads* predicate ensures that sequenced-before orders the actions accordingly with the following conjuncts:

```
relation_over Xo.actions Xo.sb ∧
threadwise Xo.actions Xo.sb ∧
strict_partial_order Xo.actions Xo.sb
```

Sequenced before can be partial: paragraph **1.9p15** leaves the ordering of the evaluation of arguments to functions undefined. It also explains that the body of functions on a single thread are ordered by sequenced before, even if they are used as the arguments to a function:

1,9p15

> [. . .][*Note:* Value computations and side effects associated with different argument expressions are unsequenced. — *end note*] Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.[. . .]

Atomic accesses, fences and mutex calls are all represented in the standard as functions and are therefore always sequenced with respect to other memory accesses. This property must be guaranteed by the thread-local semantics, and is captured by the following predicate:

```
let indeterminate_sequencing Xo =
  ∀ a ∈ Xo.actions b ∈ Xo.actions.
    (tid_of a = tid_of b) ∧ (a ≠ b) ∧
    ¬ (is_at_non_atomic_location Xo.lk a ∧ is_at_non_atomic_location Xo.lk b) ⟶
    (a, b) ∈ Xo.sb ∨ (b, a) ∈ Xo.sb
```

The model presents the behaviour of a program as a set of candidate executions, and as we have seen there are some actions that block permanently. The semantics of the blocking actions is defined in terms of sequenced before. A consistent execution cannot have a blocked action with further actions sequenced after it — that is, the thread must block, and not allow program-order-later code to be part of the execution. This restriction is captured by the *blocking_observed* predicate:

```
let blocking_observed actions sb =
  (∀ a ∈ actions.
    (is_blocked_rmw a ∨ is_blocked_lock a)
    ⟶
    ¬ (∃ b ∈ actions. (a, b) ∈ sb))
```

B.1.4 Additional synchronises with

The examples throughout this thesis, and the language presented in the thread-local semantics, provide a simple parallel composition syntax that differs from that of the standard. Nonetheless, the impact of thread construction on the order of accesses to memory must be recorded, regardless of syntax. The standard provides intricate thread construction syntax in Section **30.3.1.2**, and describes its synchronisation behaviour:

30.3.1.2p5

```
template <class F, class ...Args> explicit thread(F&& f, Args&&...
args);
```

Synchronization: The completion of the invocation of the constructor synchronizes with the beginning of the invocation of the copy of `f`.

This call executes the expressions provided in `f` as a new thread. The standard dictates that there is synchronises-with ordering from the call of the thread constructor in the parent, to the start of the child thread. In the model, this ordering is collected in the additional synchronises-with relation, asw. The thread-local semantics creates this ordering from the actions preceding a parallel composition, to the sequenced-before minima in each child thread. The *well_formed_threads* predicate does not impose much restriction on additional synchronises-with, just that it is a relation over the actions that relates actions from different threads:

relation_over $Xo.actions$ $Xo.asw$ $\wedge$
interthread $Xo.actions$ $Xo.asw$

B.1.5 Data dependence

In Section **1.10**, the definition of happens-before is contingent on a notion of data dependence that is described in the definition of the carries-a-dependency-to relation:

1.10p

An evaluation A *carries a dependency to* an evaluation B if

- the value of A is used as an operand of B, unless:
  - B is an invocation of any specialization of `std::kill_dependency` (29.3), or
  - A is the left operand of a built-in logical AND (`&&`, see 5.14) or logical OR (`||`, see 5.15) operator, or
  - A is the left operand of a conditional (`?:`, see 5.16) operator, or
  - A is the left operand of the built-in comma (`,`) operator (5.18); or [...]

The thread-local semantics collects these syntactic dependencies in the data dependence relation, dd, and that is passed to the memory model as part of the pre-execution. The relation is a partial order that identifies a subset of sequenced-before where there is data dependence between memory actions. The restrictions imposed on the relation in *well_formed_threads* are:

```
strict_partial_order Xo.actions Xo.dd ∧
Xo.dd subset Xo.sb
```

B.1.6 Well-formed threads

The action set, location kinds and relations of the pre-execution are generated by the thread-local semantics. The pre-executions that the thread-local semantics generates should all satisfy the basic properties established above. The *well_formed_threads* predicate gathers those requirements together:

```
let well_formed_threads ((Xo, _, _) : (PRE_EXECUTION * EXECUTION_WITNESS *
RELATION_LIST)) =
  (∀ a ∈ Xo.actions. well_formed_action a) ∧
  actions_respect_location_kinds Xo.actions Xo.lk ∧
  blocking_observed Xo.actions Xo.sb ∧
  inj_on aid_of Xo.actions ∧
  relation_over Xo.actions Xo.sb ∧
  relation_over Xo.actions Xo.asw ∧
  threadwise Xo.actions Xo.sb ∧
  interthread Xo.actions Xo.asw ∧
  isStrictPartialOrder Xo.sb ∧
  isStrictPartialOrder Xo.dd ∧
  Xo.dd ⊆ Xo.sb ∧
  indeterminate_sequencing Xo ∧
  isIrreflexive (sbasw Xo) ∧
  finite_prefixes (sbasw Xo) Xo.actions
```

B.2 Execution witness

The execution witness represents the dynamic behaviour of memory in an execution. It is defined in the model as a record containing several relations:

```
type EXECUTION_WITNESS =
  ⦃ rf : SET (ACTION * ACTION);
    mo : SET (ACTION * ACTION);
```

sc : SET (ACTION $*$ ACTION);
lo : SET (ACTION $*$ ACTION);
tot : SET (ACTION $*$ ACTION);
⟫

Each model given in Chapter 3 used some subset of the relations. The standard model used rf, mo, lo and sc, and each of these are justified by the standard below.

B.2.1 Reads-from

The behaviour of reads from memory is described in **1.10**:

1.10p3

> The value of an object visible to a thread T at a particular point is the initial value of the object, a value assigned to the object by T, or a value assigned to the object by another thread, according to the rules below. [...]

The excerpt requires a read to read its value from a particular write. Consequently, the model records the particular write that a read reads from, and collects all such relationships together in the reads-from relation. The model checks some basic sanity properties over the relation: that it is an injective relation from write actions to read actions at the same location, and that each read returns the same value as the related write:

```
let well_formed_rf (Xo, Xw, _) =
  ∀ (a, b) ∈ Xw.rf.
    a ∈ Xo.actions ∧ b ∈ Xo.actions ∧
    loc_of a = loc_of b ∧
    is_write a ∧ is_read b ∧
    value_read_by b = value_written_by a ∧
    ∀ a' ∈ Xo.actions. (a', b) ∈ Xw.rf ⟶ a = a'
```

In the same passage, there is a note that alludes to the possibility of undefined behaviour that will be discussed in Section B.5.3:

1.10p3

> [...][*Note:* In some cases, there may instead be undefined behavior. Much of this section is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs. — *end note*]

B.2.2 Modification order

Modification order is introduced in **1.10p6**:

1.10p6

> All modifications to a particular atomic object M occur in some particular total order, called the *modification order* of M. [...]

The model includes a modification-order relation in the execution witness, that collects together the orders on all atomic locations. The passage requires each modification order to be a total order over the writes at its location. The restriction is imposed by the *consistent_mo* predicate. It requires that mo is transitive and irreflexive, that it agrees with happens-before, and that for every pair of actions, they are related by modification order if and only if they are writes to the same atomic location.

```
let consistent_mo (Xo, Xw, _) =
  relation_over Xo.actions Xw.mo ∧
  isTransitive Xw.mo ∧
  isIrreflexive Xw.mo ∧
  ∀ a ∈ Xo.actions b ∈ Xo.actions.
    ((a, b) ∈ Xw.mo ∨ (b, a) ∈ Xw.mo)
    = ( (¬ (a = b)) ∧
        is_write a ∧ is_write b ∧
        (loc_of a = loc_of b) ∧
        is_at_atomic_location Xo.lk a )
```

Together these restrictions require mo to be a total order over the writes at each location, as the standard requires. There are many other requirements that relate to modification order, including a further restriction required by the passage that introduced modification order:

1.10p6

> If A and B are modifications of an atomic object M and A happens before (as defined below) B, then A shall precede B in the modification order of M, which is defined below. [*Note:* This states that the modification orders must respect the "happens before" relationship. — *end note*] [...]

This requirement will be described in Section A together with the set of coherence requirements that link modificaiton-order, happens-before and reads-from.

Modification order and read-modify-writes Modification order ensures the atomicity of read-modify-write actions. The standard restricts the writes that may be read by read-modify-write actions:

29.3p12

> Atomic read-modify-write operations shall always read the last value (in the modification order) written before the write associated with the read-modify-write operation.

The corresponding predicate in the model checks that reads-from edges to read-modify-write actions come from the immediately preceding write in modification order, establishing the restriction from the standard directly:

```
let rmw_atomicity (Xo, Xw, _) =
  ∀ b ∈ Xo.actions a ∈ Xo.actions.
    is_RMW b ⟶ (adjacent_less_than Xw.mo Xo.actions a b = ((a, b) ∈ Xw.rf))
```

B.2.3 Lock order

Total orders over all of the lock and unlock actions on mutex locations are introduced in **30.4.1.2**:

30.4.1.2p5

> [...]The lock and unlock operations on a single mutex shall appear to occur in a single total order. [*Note:* this can be viewed as the modification order (1.10) of the mutex. — *end note*] [...]

The model captures these per-location orders as a single relation that is simply the union over all locations in the execution. The *locks_only_consistent_lo* predicate checks that the lock order relation behaves accordingly. Given the note in the passage above, the predicate is similar to the restriction on modification order, except that in this case, the relation is over the set of all lock and unlock actions at each mutex location, and the predicate checks that lock order and happens-before agree:

```
let locks_only_consistent_lo (Xo, Xw, ("hb", hb) :: _) =
  relation_over Xo.actions Xw.lo ∧
  isTransitive Xw.lo ∧
```

```
  isIrreflexive Xw.lo ∧
  ∀ a ∈ Xo.actions b ∈ Xo.actions.
    ((a, b) ∈ Xw.lo ⟶ ¬ ((b, a) ∈ hb)) ∧
    ( ((a, b) ∈ Xw.lo ∨ (b, a) ∈ Xw.lo)
      =
      ( (¬ (a = b)) ∧
        (is_lock a ∨ is_unlock a) ∧
        (is_lock b ∨ is_unlock b) ∧
        (loc_of a = loc_of b) ∧
        is_at_mutex_location Xo.lk a
      )
    )
```

Together, these restrictions require lock order to be an irreflexive total order over each mutex location. The restriction that makes lock order agree with happens-before, although not explicitly stated, is suggested by the note that draws a similarity between lock order and modification order.

B.2.4 SC order

The SC order is introduced in **29.3p3**:

29.3p3

> There shall be a single total order S on all `memory_order_seq_cst` operations, consistent with the "happens before" order and modification orders for all affected locations, [...]

The model captures the S order with the sc relation, and the restrictions required by the standard are imposed by the *sc_accesses_consistent_sc* predicate. The predicate is similar to those of modification order and lock order, but this relation is not restricted to a single location, it is over all SEQ_CST accesses, and it must agree with both happens-before and modification order:

```
let sc_accesses_consistent_sc (Xo, Xw, ("hb", hb) :: _) =
  relation_over Xo.actions Xw.sc ∧
  isTransitive Xw.sc ∧
  isIrreflexive Xw.sc ∧
  ∀ a ∈ Xo.actions b ∈ Xo.actions.
    ((a, b) ∈ Xw.sc ⟶ ¬ ((b, a) ∈ hb ∪ Xw.mo)) ∧
```

```
( ((a, b) ∈ Xw.sc ∨ (b, a) ∈ Xw.sc) =
  ( (¬ (a = b)) ∧ is_seq_cst a ∧ is_seq_cst b)
)
```

The predicate requires sc order to be an irreflexive total order over SEQ_CST accesses that agrees with happens-before and modification order, as required by the standard.

B.3 Calculated relations

Rather than form its judgement over the relations of the pre-execution and the execution witness directly, the memory model defines calculated relations that capture derivative intuitions, and the predicates of the model use these to decide what is allowed. The standard model uses the following calculated relations:

```
let standard_relations Xo Xw =
  let hrs = hypothetical_release_sequence_set Xo.actions Xo.lk Xw.mo in
  let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
  let sw = release_acquire_fenced_synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf Xw.lo rs h
  let cad = with_consume_cad_set Xo.actions Xo.sb Xo.dd Xw.rf in
  let dob = with_consume_dob_set Xo.actions Xw.rf rs cad in
  let ithb = inter_thread_happens_before Xo.actions Xo.sb sw dob in
  let hb = happens_before Xo.actions Xo.sb ithb in
  let vse = visible_side_effect_set Xo.actions hb in
  let vsses = standard_vsses Xo.actions Xo.lk Xw.mo hb vse in
  [ ("hb", hb);
    ("vse", vse);
    ("ithb", ithb);
    ("vsses", vsses);
    ("sw", sw);
    ("rs", rs);
    ("hrs", hrs);
    ("dob", dob);
    ("cad", cad) ]
```

Each relation is discussed below, and its definition is justified by text from the standard.

B.3.1 Release sequence

Release sequences are introduced in **1.10p7**:

1.10p7

A *release sequence* headed by a release operation A on an atomic object M is a maximal contiguous subsequence of side effects in the modification order of M, where the first operation is A, and every subsequent operation

- is performed by the same thread that performed A, or
- is an atomic read-modify-write operation.

This sequence is captured by the model in two functions. The first takes a head action as an argument, identifies actions on the same thread, and any read-modify-writes, i.e. the actions that may be in the tail of the release sequence:

```
let rs_element head a =
  (tid_of a = tid_of head) ∨ is_RMW a
```

The second function defines a relation, where for every pair (rel, b), rel is the head of a release sequence that contains b. The function first checks that rel is a release. Then the function checks that b is in the release sequence of rel: either b and rel are the same action, or b satisfies $rs\_element$, b follows rel in modification order, and every intervening action in modification order is in the release sequence:

```
let release_sequence_set actions lk mo =
  { (rel, b) | ∀ rel ∈ actions b ∈ actions |
    is_release rel ∧
    ( (b = rel) ∨
      ( (rel, b) ∈ mo ∧
        rs_element rel b ∧
        ∀ c ∈ actions.
          ((rel, c) ∈ mo ∧ (c, b) ∈ mo) ⟶ rs_element rel c ) ) }
```

This relation does not order the members of a particular release sequence, but that information is contained in modification order, and is never used in the judgement of the memory model.

B.3.2 Hypothetical release sequence

In the definition of the behaviour of release fences, the standard mentions hypothetical release sequences:

29.8p2

> A release fence A synchronizes with an acquire fence B if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

Although not an explicit definition, this relation decides the synchronisation behaviour of release fences, so the model does include the relation. In the passage above and the other mention of hypothetical release sequences in the standard, the head is an atomic write. Consequently, the definition is the same as that of release sequences, with the requirement that the head be a release relaxed, so that it only need be an atomic write:

```
let hypothetical_release_sequence_set actions lk mo =
  { (a, b) | ∀ a ∈ actions b ∈ actions |
    is_atomic_action a ∧
    is_write a ∧
    ( (b = a) ∨
      ( (a, b) ∈ mo ∧
        rs_element a b ∧
        ∀ c ∈ actions.
          ((a, c) ∈ mo ∧ (c, b) ∈ mo) ⟶ rs_element a c ) ) }
```

B.3.3 Synchronises with

The standard introduces synchronises-with in Section **1.10**:

1.10p8

> Certain library calls *synchronize with* other library calls performed by another thread. For example, an atomic store-release synchronizes with a load-acquire that takes its value from the store (29.3). [...]

The synchronises-with relation captures the majority of inter-thread ordering defined by the standard. The individual cases where library calls do create synchronisation are described throughout the rest of the standard, and not all synchronising features are supported by the thread-local semantics (handler functions, stream objects, thread exit notifications and futures). The sources of synchronisation that are supported are thread

creation, mutex accesses, atomic reads and writes, and atomic fences. Each of these will be dealt with in turn.

Thread creation Recall the quote from Section **30.3.1.2** that justified the existence of the additional-synchronises-with relation. Here the standard describes the synchronisation that arises from thread creation:

30.3.1.2p5

> `template <class F, class ...Args> explicit thread(F&& f, Args&&... args);`
>
> *Synchronization*: The completion of the invocation of the constructor synchronizes with the beginning of the invocation of the copy of f.

To capture this in the model, the calculation of the synchronises-with relation includes all of additional-synchronises-with:

```
let release_acquire_fenced_synchronizes_with actions sb asw rf lo rs hrs a b =
  (tid_of a ≠ tid_of b) ∧
  ((* thread sync *)
    (a, b) ∈ asw ∨
    [...] )
```

Mutex synchronisation Synchronisation resulting from locks is described in terms of the lock order relation. The standard describes the specifics of mutex synchronisation in section **30.4.1.2**, first on locks:

30.4.1.2p11

> *Synchronization:* Prior `unlock()` operations on the same object shall *synchronize with* (1.10) this operation.

and then on unlocks:

30.4.1.2p

> *Synchronization:* This operation *synchronizes with* (1.10) subsequent lock operations that obtain ownership on the same object.

In the model, blocked locks do not engage in synchronisation. The disjunct that captures mutex synchronisation therefore turns every edge in lock order from an unlock to a successful lock into a synchronises-with edge:

```
let release_acquire_fenced_synchronizes_with actions sb asw rf lo rs hrs a b =
  (tid_of a ≠ tid_of b) ∧
([...]   (* mutex sync *)
    (is_unlock a ∧ is_successful_lock b ∧ (a, b) ∈ lo) ∨
[...])
```

Release-acquire synchronisation Release-acquire synchronisation is described in section **29.3**:

29.3p2

> An atomic operation A that performs a release operation on an atomic object M synchronizes with an atomic operation B that performs an acquire operation on M and takes its value from any side effect in the release sequence headed by A.

The model already has a calculated relation that represents release sequences, so the formalisation of this requirement is straightforward. Synchronisation is created from release actions to acquire actions that read from writes in the release sequence of the release:

```
let release_acquire_fenced_synchronizes_with actions sb asw rf lo rs hrs a b =
  (tid_of a ≠ tid_of b) ∧
([...]   (* rel/acq sync *)
    (is_release a ∧ (is_acquire b ∧
      (∃c∈actions. (a, c) ∈ rs ∧ (c, b) ∈ rf) ∨
[...])
```

Fence synchronisation Section **29.8** of the standard details the synchronisation that is created by release and acquire fences:

29.8p2

> A release fence A synchronizes with an acquire fence B if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

This passage is relatively intricate. See Chapter 3 for a motivation for the existence of fences and an example of a program that relies on this requirement.

In the model, this synchronisation is formalised directly using the calculated relation that describes the hypothetical release sequence:

```
let release_acquire_fenced_synchronizes_with actions sb asw rf lo rs hrs a b =
  (tid_of a ≠ tid_of b) ∧
([...]
(* fence synchronisation *)
   (is_fence a ∧ is_release a ∧ is_fence b ∧ is_acquire b ∧
    ∃x∈actions y∈actions z∈actions.
      (a, x) ∈ sb ∧ (x, y) ∈ hrs ∧ (y, z) ∈ rf ∧ (y, b) ∈ sb) ∨
[...]
```

Partially-fenced synchronisation The standard defines the interaction of release and acquire fences with release and acquire atomics in two passages in Section **29.8**. The first describes the case where the acquire side of the usual release-acquire graph is an atomic acquire read, and the release side is a release fence followed by an atomic write:

29.8p3

> A release fence A synchronizes with an atomic operation B that performs an acquire operation on an atomic object M if there exists an atomic operation X such that A is sequenced before X, X modifies M, and B reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

The second passage describes the other case, where the release thread contains an atomic write-release, and the acquire thread atomically reads, and then has an acquire fence:

29.8p4

> An atomic operation A that is a release operation on an atomic object M synchronizes with an acquire fence B if there exists some atomic operation X on M such that X is sequenced before B and reads the value written by A or a value written by any side effect in the release sequence headed by A.

The corresponding rules in the model express the execution fragments described in the standard directly using the calculated relations release sequence and hypothetical release sequence:

```
let release_acquire_fenced_synchronizes_with actions sb asw rf lo rs hrs a b =
  (tid_of a ≠ tid_of b) ∧
([...]   (is_fence a ∧ is_release a ∧ is_acquire b ∧
     ∃x∈actions y∈actions.
        (a, x) ∈ sb ∧ (x, y) ∈ hrs ∧ (y, b) ∈ rf) ∨
    (is_release a ∧ is_fence b ∧ is_acquire b ∧
     ∃x∈actions y∈actions.
       (a, x) ∈ rs ∧ (x, y) ∈ rf ∧ (y, b) ∈ sb)
```

Synchronises with Each part of the function that calculates synchronises-with has now been related to passages from the standard, so the whole rule can be stated:

```
let release_acquire_fenced_synchronizes_with actions sb asw rf lo rs hrs a b =
  (tid_of a ≠ tid_of b) ∧
  ( (* thread sync *)
    (a, b) ∈ asw ∨
    (* mutex sync *)
    (is_unlock a ∧ is_lock b ∧ (a, b) ∈ lo) ∨
    (* rel/acq sync *)
    ( is_release a ∧ is_acquire b ∧
      (∃ c ∈ actions. (a, c) ∈ rs ∧ (c, b) ∈ rf) ) ∨
    (* fence synchronisation *)
    ( is_fence a ∧ is_release a ∧ is_fence b ∧ is_acquire b ∧
      ∃ x ∈ actions z ∈ actions y ∈ actions.
        (a, x) ∈ sb ∧ (x, z) ∈ hrs ∧ (z, y) ∈ rf ∧ (y, b) ∈ sb) ∨
    ( is_fence a ∧ is_release a ∧ is_acquire b ∧
      ∃ x ∈ actions y ∈ actions.
         (a, x) ∈ sb ∧ (x, y) ∈ hrs ∧ (y, b) ∈ rf ) ∨
    ( is_release a ∧ is_fence b ∧ is_acquire b ∧
```

$\exists\ y \in actions\ x \in actions.$
$(a,\ y) \in rs \wedge (y,\ x) \in rf \wedge (x,\ b) \in sb)\ )$

B.3.4 Carries a dependency to

The standard introduces weaker inter-thread ordering to express the semantics of consume atomics. Several relations take part in this definition, starting with carries-a-dependency-to in Section **1.10**:

1.10p9

> An evaluation A *carries a dependency* to an evaluation B if
>
> - the value of A is used as an operand of B, unless:
>   - B is an invocation of any specialization of `std::kill_dependency` (29.3), or
>   - A is the left operand of a built-in logical AND (`&&`, see 5.14) or logical OR (——, see 5.15) operator, or
>   - A is the left operand of a conditional (`?:`, see 5.16) operator, or
>   - A is the left operand of the built-in comma (`,`) operator (5.18);
>
>   or
> - A writes a scalar object or bit-field M, B reads the value written by A from M, and A is sequenced before B, or
> - for some evaluation X, A carries a dependency to X, and X carries a dependency to B.
>
> [*Note:* "Carries a dependency to" is a subset of "is sequenced before", and is similarly strictly intra-thread. — *end note*]

The model has a carries-a-dependency-to relation, cad, that uses the data-dependence relation, dd, previously defined to capture the ordering described by the first bullet in the passage. The second bullet is captured in the standard as the intersection of reads-from and sequenced-before. The final rule includes all transitive closures of the union of the first two rules. The formalisation of the cad relation is simply the transitive closure of the union of data dependence and read-from intersected with sequenced-before:

let $with\_consume\_cad\_set\ actions\ sb\ dd\ rf\ =$ transitiveClosure $(\ (rf \cap sb) \cup dd\ )$

B.3.5 Dependency ordered before

The next calculated relation defines an analogue of release-acquire synchronisation for release and consume atomics:

1.10p10

> An evaluation A is *dependency-ordered before* an evaluation B if
>
> - A performs a release operation on an atomic object M, and, in another thread, B performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or
> - for some evaluation X, A is dependency-ordered before X and X carries a dependency to B.
>
> [*Note:* The relation "is dependency-ordered before" is analogous to "synchronizes with", but uses release/consume in place of release/acquire. — *end note*]

The first case in the definition above is similar to the definition of release and acquire synchronisation: a release write is dependency ordered before a read if the release write heads a release sequence that contains a write, and the read reads-from that write. The second case extends dependency-ordered-before edges through all following carries-a-dependency-to edges. The definition of the dependency-ordered-before relation, dob, is formalised directly in the model using the rs and cad calculated relations:

```
let with_consume_dob actions rf rs cad w a =
  tid_of w ≠ tid_of a ∧
  ∃ w' ∈ actions r ∈ actions.
    is_consume r ∧
    (w, w') ∈ rs ∧ (w', r) ∈ rf ∧
    ( (r, a) ∈ cad ∨ (r = a) )
```

B.3.6 Inter-thread happens before

The standard defines inter-thread happens-before in Section **1.10**:

1.10p11

An evaluation A *inter-thread happens before* an evaluation B if

- A synchronizes with B, or
- A is dependency-ordered before B, or
- for some evaluation X
  - A synchronizes with X and X is sequenced before B, or
  - A is sequenced before X and X inter-thread happens before B, or
  - A inter-thread happens before X and X inter-thread happens before B.

[*Note:* The "inter-thread happens before" relation describes arbitrary concatenations of "sequenced before", "synchronizes with" and "dependency-ordered before" relationships, with two exceptions. The first exception is that a concatenation is not permitted to end with "dependency-ordered before" followed by "sequenced before". The reason for this limitation is that a consume operation participating in a "dependency-ordered before" relationship provides ordering only with respect to operations to which this consume operation actually carries a dependency. The reason that this limitation applies only to the end of such a concatenation is that any subsequent release operation will provide the required ordering for a prior consume operation. The second exception is that a concatenation is not permitted to consist entirely of "sequenced before". The reasons for this limitation are (1) to permit "inter-thread happens before" to be transitively closed and (2) the "happens before" relation, defined below, provides for relationships consisting entirely of "sequenced before". — *end note*]

As the note explains, at each dependency-ordered-before edge, the relation is only extended through dependencies. The definition of inter-thread happens-before, ithb, in the model is:

```
let inter_thread_happens_before actions sb sw dob =
  let r = sw ∪ dob ∪ (compose sw sb) in
    transitiveClosure (r ∪ (compose sb r))
```

This relation has been proven equivalent to a direct formalisation of the standard's recursive definition in the HOL4 theorem prover. The proof relies on the transitivity of sequenced before, and involves an induction over the transitive closure of model happens-before in one direction, and an induction over the rules that define the standard's relation in the other. All cases are trivial except the case where we have an edge in the standard's relation with a sequenced-before edge preceding an inter-thread-happens-before edge. To witness an edge in the model's happens-before relation, we consider two cases: one where

the inter-thread-happens-before is a single step in the model's relation, and another where it is a single step followed by an edge in transitively-closed inter-thread-happens-before. In either case, by transitivity of sequenced before the sequenced edge can be composed with the single-step inter-thread-happens-before edge to get an edge in transitively-closed inter-thread-happens-before, and we are done.

B.3.7 Happens-before

The standard defines happens-before in Section **1.10**:

1.10p12

> An evaluation A *happens before* an evaluation B if:
>
> - A is sequenced before B, or
> - A inter-thread happens before B.
>
> The implementation shall ensure that no program execution demonstrates a cycle in the "happens before" relation. [*Note:* This cycle would otherwise be possible only through the use of consume operations. — *end note*]

The definition of happens-before in the model follows directly from the text. It is simply the union of sequenced-before and inter-thread happens-before:

```
let happens_before actions sb ithb =
  sb ∪ ithb
```

The acyclicity of happens-before is guaranteed by the following predicate over consistent executions:

```
let consistent_hb (Xo, _, ("hb", hb) :: _) =
  isIrreflexive (transitiveClosure hb)
```

B.3.8 Visible side-effects

The standard defines visible side effects in Section **1.10**:

1.10p13

> A *visible side effect* A on a scalar object or bit-field M with respect to a value computation B of M satisfies the conditions:
>
> - A happens before B and
> - there is no other side effect X to M such that A happens before X and X happens before B.
>
> [...]

The model represents visible side-effects in a relation from each visible side-effect to any reads to which it is visible. Any two related actions must be related by happens-before, they must be a write and a read at the same location, and there must not be a happens-before-intervening write at the same location:

```
let visible_side_effect_set actions hb =
  { (a, b) | ∀ (a, b) ∈ hb |
    is_write a ∧ is_read b ∧ (loc_of a = loc_of b) ∧
    ¬ ( ∃ c ∈ actions. ¬ (c ∈ {a, b}) ∧
        is_write c ∧ (loc_of c = loc_of b) ∧
        (a, c) ∈ hb ∧ (c, b) ∈ hb) }
```

Visible side effects are used, amongst other things, to determine which writes may be read by non-atomic reads. Later in the same passage that defined visible side effects, the standard states:

1.10p13

> [...]The value of a non-atomic scalar object or bit-field M, as determined by evaluation B, shall be the value stored by the visible side effect A. [*Note:* If there is ambiguity about which side effect to a non-atomic object or bit-field is visible, then the behavior is either unspecified or undefined. — *end note*] [...]

It has not been established that the visible side effect of a non-atomic read is unique, and in fact in racy programs, it need not be. The note in the passage above is alluding to this. As a consequence of this possible ambiguity, the model permits non-atomic reads to read from any visible side effect:

```
let consistent_non_atomic_rf (Xo, Xw, _ :: ("vse", vse) :: _) =
  ∀ (w, r) ∈ Xw.rf. is_at_non_atomic_location Xo.lk r ⟶
    (w, r) ∈ vse
```

B.3.9 Visible sequence of side effects

The standard includes the notion of a visible sequence of side effects:

1.10p14

> The *visible sequence of side effects* on an atomic object M, with respect to a value computation B of M, is a maximal contiguous subsequence of side effects in the modification order of M, where the first side effect is visible with respect to B, and for every side effect, it is not the case that B happens before it. [...]

The model represents these sequences as a relation with edges from the set of writes that makes up each sequence, to its read. The ordering of the sequence is provided by modification order. The definition of the relation is a set comprehension that, for a given read, first finds the modification-order-maximal visible side effect of the read, and then relates this write to the read, as well as any modification-order-following writes that do not happen after the read, or have a modification-order intervening write that does:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{standard\_vsses}\ \mathit{actions}\ \mathit{lk}\ \mathit{mo}\ \mathit{hb}\ \mathit{vse}\ = \\
\quad \{\ (v,\ r)\ |\ \forall\ r \in \mathit{actions}\ v \in \mathit{actions}\ \mathit{head} \in \mathit{actions}\ | \\
\quad\ \mathrm{is\_at\_atomic\_location}\ \mathit{lk}\ r \wedge (\mathit{head},\ r) \in \mathit{vse} \wedge \\
\quad\ \neg\ (\exists\ v' \in \mathit{actions}.\ (v',\ r) \in \mathit{vse} \wedge (\mathit{head},\ v') \in \mathit{mo}) \wedge \\
\quad\ (\ v = \mathit{head} \vee \\
\quad\quad (\ (\mathit{head},\ v) \in \mathit{mo} \wedge \neg\ ((r,\ v) \in \mathit{hb}) \wedge \\
\quad\quad\ \forall\ w \in \mathit{actions}. \\
\quad\quad\quad ((\mathit{head},\ w) \in \mathit{mo} \wedge (w,\ v) \in \mathit{mo}) \longrightarrow \neg\ ((r,\ w) \in \mathit{hb}) \\
\quad\quad) \\
\quad\ ) \\
\quad \}
\end{array}
$$

In the same passage that defines visible sequences of side effects, the standard says that the write read by an atomic read is restricted to a write in the visible sequence of side effects of the read.

1.10p14

> [...]The value of an atomic object M, as determined by evaluation B, shall be the value stored by some operation in the visible sequence of M with respect to B. [...]

As we shall see in Chapter 5, this sequence was intended to enumerate the set of all writes that may be read by an atomic read, but it is made redundant by the coherence requirements. Nonetheless the mathematical model that follows the standard includes the restriction. Every read at an atomic location must read from a write in its visible sequence of side effects:

```
let standard_consistent_atomic_rf (Xo, Xw, _ :: _ :: _ :: ("vsses", vsses) :: _) =
  ∀ (w, r) ∈ Xw.rf. is_at_atomic_location Xo.lk r ∧ is_load r⟶
    (w, r) ∈ vsses
```

B.3.10 Summary of calculated relations

The model collects the calculated relations in a record of name-relation pairs:

```
let standard_relations Xo Xw =
  let hrs = hypothetical_release_sequence_set Xo.actions Xo.lk Xw.mo in
  let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
 let sw = release_acquire_fenced_synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf Xw.lo rs hrs in
  let cad = with_consume_cad_set Xo.actions Xo.sb Xo.dd Xw.rf in
  let dob = with_consume_dob_set Xo.actions Xw.rf rs cad in
  let ithb = inter_thread_happens_before Xo.actions Xo.sb sw dob in
  let hb = happens_before Xo.actions Xo.sb ithb in
  let vse = visible_side_effect_set Xo.actions hb in
  let vsses = standard_vsses Xo.actions Xo.lk Xw.mo hb vse in
  [ ("hb", hb);
    ("vse", vse);
    ("ithb", ithb);
    ("vsses", vsses);
    ("sw", sw);
    ("rs", rs);
    ("hrs", hrs);
    ("dob", dob);
    ("cad", cad) ]
```

B.4 The consistency predicate

This section reviews the consistency predicate, and justifies it with quotes from the standard. The consistency predicate is made up of a list of named predicates, each of which must hold for consistency. Several predicates have already been covered, and this section presents the remainder.

B.4.1 Coherence

The previous section introduced visible sequences of side-effects, which were originally intended to enumerate the writes that a particular atomic read could consistently read from. This relation was superseded by the four coherence requirements of Section **1.10**. Each is a relatively simple subgraph of reads and writes at a single location. The graphs are made up of modification order, happens-before and reads-from edges, and each represents a behaviour where the three relations disagree in some way.

CoRR In this behaviour, two reads on a single thread observe writes in an order that opposes modification order:

1.10p16

> If a value computation A of an atomic object M happens before a value computation B of M, and A takes its value from a side effect X on M, then the value computed by B shall either be the value stored by X or the value stored by a side effect Y on M, where Y follows X in the modification order of M. [*Note:* This requirement is known as read-read coherence. — *end note*]

This restriction is captured by the following part of the consistency predicate:

```
let coherent_memory_use (Xo, Xw, ("hb", hb) :: _) =
  (* CoRR *)
  (¬ (∃(a, b)∈Xw.rf (c, d)∈Xw.rf.
        (b, d) ∈ hb ∧ ((c, a) ∈ Xw.mo))) ∧
[...]
```

The other coherence requirements can be seen as derivatives of this where a read and write that participate in a reads-from edge are replaced with a single write in place of the read.

CoRW In this coherence violation, there is a cycle created by modification order, happens-before and reads-from:

1.10p17

> If a value computation A of an atomic object M happens before an operation B on M, then A shall take its value from a side effect X on M, where X precedes B in the modification order of M. [*Note:* This requirement is known as read-write coherence. — *end note*]

The model formalises this restriction as follows:

```
let coherent_memory_use (Xo, Xw, ("hb", hb) :: _) =
[...]   (* CoRW *)
   ((¬ (∃(a, b)∈Xw.rf  c∈Xo.actions.
          (b, c)  ∈  hb ∧ ((c, a)  ∈  Xw.mo))) ∧
[...]
```

CoWR Here the read reads from a write that is hidden by a modification-order-later write that happens before the read:

1.10p18

> If a side effect X on an atomic object M happens before a value computation B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M. [*Note:* This requirement is known as write-read coherence. — *end note*]

The model formalises this restriction as follows:

```
let coherent_memory_use (Xo, Xw, ("hb", hb) :: _) =
[...]   (* CoWR *)
   ((¬ (∃(a, b)∈Xw.rf  c∈Xo.actions.
          (c, b)  ∈  hb ∧ ((a, c)  ∈  Xw.mo))) ∧
[...]
```

CoWW Finally modification order and happens-before must agree:

1.10p15

> If an operation A that modifies an atomic object M happens before an operation B that modifies M, then A shall be earlier than B in the modification order of M. [*Note:* This requirement is known as write-write coherence. — *end note*]

The model formalises this restriction as follows:

```
let coherent_memory_use (Xo, Xw, ("hb", hb) :: _) =
[...]   (* CoWW *)
   (¬ (∃(a, b)∈hb. (b, a)  ∈  Xw.mo))))
```

The coherence restriction These four restrictions are combined in the *coherent_memory_use* restriction in the memory model:

```
let coherent_memory_use (Xo, Xw, ("hb", hb) :: _) =
  (* CoRR *)
  ( ¬ ( ∃ (a, b) ∈ Xw.rf (c, d) ∈ Xw.rf.
        (b, d) ∈ hb ∧ (c, a) ∈ Xw.mo ) ) ∧
  (* CoWR *)
  ( ¬ ( ∃ (a, b) ∈ Xw.rf c ∈ Xo.actions.
        (c, b) ∈ hb ∧ (a, c) ∈ Xw.mo ) ) ∧
  (* CoRW *)
  ( ¬ ( ∃ (a, b) ∈ Xw.rf c ∈ Xo.actions.
        (b, c) ∈ hb ∧ (c, a) ∈ Xw.mo ) ) ∧
  (* CoWW *)
  ( ¬ (∃ (a, b) ∈ hb. (b, a) ∈ Xw.mo) )
```

B.4.2 SC fences

The semantics of fences with sequentially consistent memory order are given in Section **29.3**. There are, broadly speaking, two sorts of restriction imposed: those that impact reads-from edges and those that impact modification order. For each case, there is a canonical restriction that uses two sequentially consistent fences, and then there are some derivative restrictions that govern the interaction between sequentially consistent fences and sequentially consistent atomic accesses. The model enforces stronger restrictions than the standard here — the standard omits the derivative cases for restricting modification order. There is an outstanding proposal to change this, and it has met with informal approval amongst the C++ committee. This proposal is further discussed in Chapter 5.

The canonical reads-from restriction The standard imposes the following restriction on the writes that may be read by an atomic read, in the presence of sequentially consistent fences:

29.3p4

> For an atomic operation B that reads the value of an atomic object M, if there is a `memory_order_seq_cst` fence X sequenced before B, then B observes either the last `memory_order_seq_cst` modification of M preceding X in the total order S or a later modification of M in its modification order.

Rather than identifying the writes that may be read from, as the standard does above, the model identifies those writes that may not be read. The restriction below forbids executions from reading writes that precede the last modification of the location before the fence:

```
let sc_fenced_sc_fences_heeded (Xo, Xw, ("hb", hb) :: _) =
  ∀f∈Xo.actions f'∈Xo.actions
      r∈Xo.actions
      w∈Xo.actions w'∈Xo.actions. ¬ (is_fence f ∧ (is_fence f' ∧
[...]         (* fence restriction N3291 29.3p6 *)
        (((w, w') ∈ Xw.mo ∧
          ((w', f) ∈ Xo.sb ∧
          ((f, f') ∈ Xw.sc ∧
          ((f', r) ∈ Xo.sb ∧
          ((w, r) ∈ Xw.rf))))) ∨
[...]
```

First read-from derivative In this passage, the standard describes the interaction of a sequentially consistent read with an atomic write followed in the same thread by a sequentially consistent fence:

29.3p5

> For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there is a `memory_order_seq_cst` fence X such that A is sequenced before X and B follows X in S, then B observes either the effects of A or a later modification of M in its modification order.

Again, the model formalises this by forbidding the read from reading writes earlier in the modification order than the one that precedes the fence.

```
let sc_fenced_sc_fences_heeded (Xo, Xw, ("hb", hb) :: _) =
  ∀f∈Xo.actions f'∈Xo.actions
      r∈Xo.actions
      w∈Xo.actions w'∈Xo.actions. ¬ (is_fence f ∧ (is_fence f' ∧
[...]         (* fence restriction N3291 29.3p5 *)
        (((w, w') ∈ Xw.mo ∧
          ((w', f) ∈ Xo.sb ∧
          ((f, r) ∈ Xw.sc ∧
          ((w, r) ∈ Xw.rf)))) ∨
[...]
```

Second read-from derivative In this passage, the standard describes the interaction of a sequentially consistent write with a sequentially consistent fence followed in the same thread by an atomic read:

29.3p6

> For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there are *memory_order_seq_cst* fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S, then B observes either the effects of A or a later modification of M in its modification order.

Again, the restriction in the model forbids rather than allows reads-from edges. This time, there is an atomic write in the writing thread, and the restriction forbids the read from reading a modification predecessor of this write:

```
let sc_fenced_sc_fences_heeded (Xo, Xw, ("hb", hb) :: _) =
  ∀f∈Xo.actions f'∈Xo.actions
       r∈Xo.actions
       w∈Xo.actions w'∈Xo.actions. ¬ (is_fence f ∧ (is_fence f' ∧
       ((* fence restriction N3291 29.3p4 *)
         ((w, w') ∈ Xw.mo ∧
           ((w', f) ∈ Xw.sc ∧
           ((f, r) ∈ Xo.sb ∧
           ((w, r) ∈ Xw.rf)))) ∨
[...]
```

Canonical modification order restriction The standard provides one further restriction: `sc` ordering over fences imposes modification order over writes in the fenced threads:

29.3p7

> For atomic operations A and B on an atomic object M, if there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S, then B occurs later than A in the modification order of M.

Once more, the restriction is expressed as a negation in the model. Rather than forcing

modification order to agree with sc order, the model forbids it from disagreeing, relying on the totality of modification order to ensure there is an edge that agrees.

```
let sc_fenced_sc_fences_heeded (Xo, Xw, ("hb", hb) :: _) =
  ∀f∈Xo.actions f'∈Xo.actions
      r∈Xo.actions
      w∈Xo.actions w'∈Xo.actions. ¬ (is_fence f ∧ (is_fence f' ∧
[...]          (* SC fences impose mo N3291 29.3p7 *)
        (((w', f) ∈ Xo.sb ∧
          (f, f') ∈ Xw.sc ∧
          (f', w) ∈ Xo.sb ∧
          (w, w') ∈ Xw.mo)))) ∨
[...]
```

For the two derivatives, see Chapter 3

B.4.3 SC reads

The standard imposes additional restrictions on sequentially consistent reads:

29.3p3

> There shall be a single total order S on all `memory_order_seq_cst` operations, consistent with the "happens before" order and modification orders for all affected locations, such that each `memory_order_seq_cst` operation B that loads a value from an atomic object M observes one of the following values:
>
> - the result of the last modification A of M that precedes B in S, if it exists, or
> - if A exists, the result of some modification of M in the visible sequence of side effects with respect to B that is not `memory_order_seq_cst` and that does not happen before A, or
> - if A does not exist, the result of some modification of M in the visible sequence of side effects with respect to B that is not `memory_order_seq_cst`.
>
> [...]

The standard describes the behaviour in three cases, but by phrasing the restriction differently, the model coalesces the last two. In the case that the sequentially consistent read reads from a sequentially consistent write, the model requires that there is no sc intervening write to the same location. This matches the first case in the passage above.

To cover the second and third cases, where the write is not sequentially consistent, the model forbids executions where the write has a happens-before successor that is a sequentially consistent write, and that write is sc ordered before the read. This restriction encompasses both the case where there is no sc predecessor of the read, and the case where there is. The restriction in the model is therefore:

$$
\begin{aligned}
&\textsf{let } \mathit{sc\_accesses\_sc\_reads\_restricted}\ (Xo,\ Xw,\ (\text{“}hb\text{”},\ hb) :: \_)\ = \\
&\quad \forall\ (w,\ r) \in Xw.rf.\ \text{is\_seq\_cst}\ r \longrightarrow \\
&\quad\ (\ \text{is\_seq\_cst}\ w \wedge (w,\ r) \in Xw.sc \wedge \\
&\quad\quad \neg\ (\exists\ w' \in Xo.actions. \\
&\quad\quad\quad \text{is\_write}\ w' \wedge (\text{loc\_of}\ w = \text{loc\_of}\ w') \wedge \\
&\quad\quad\quad (w,\ w') \in Xw.sc \wedge (w',\ r) \in Xw.sc\ )\ ) \vee \\
&\quad\ (\ \neg\ (\text{is\_seq\_cst}\ w) \wedge \\
&\quad\quad \neg\ (\exists\ w' \in Xo.actions. \\
&\quad\quad\quad \text{is\_write}\ w' \wedge (\text{loc\_of}\ w = \text{loc\_of}\ w') \wedge \\
&\quad\quad\quad (w,\ w') \in hb \wedge (w',\ r) \in Xw.sc\ )\ )
\end{aligned}
$$

B.4.4 Consistent locks

The behaviour of locks and unlocks is governed by the lock order, lo. The *locks_only_consistent_lo* predicate has already been presented and justified, as has the synchronisation that arises from the use of locks and unlocks. The consistency predicate defines how locks may behave in an execution with its *locks_only_consistent_locks* conjunct. To justify this predicate, first consider the semantics of locks as defined in Section **30.4.1.2** of the standard:

30.4.1.2p6–30.4.1.2p9

> The expression `m.lock()` shall be well-formed and have the following semantics:
>
> *Requires:* If `m` is of type `std::mutex` or `std::timed_mutex`, the calling thread does not own the mutex.
>
> *Effects:* Blocks the calling thread until ownership of the mutex can be obtained for the calling thread.
>
> *Postcondition:* The calling thread owns the mutex.

30.4.1.2p21–30.4.1.2p23

> The expression `m.unlock()` shall be well-formed and have the following semantics:
>
> *Requires:* The calling thread shall own the mutex.
>
> *Effects:* Releases the calling thread's ownership of the mutex.

The formal model does not model ownership of mutexes, but instead identifies orders of mutex actions that correctly use ownership. The requirements forbid executions with a lock order where two successful locks on the same mutex are adjacent in lo, and the *locks_only_consistent_locks* predicate in the model captures that:

```
let locks_only_consistent_locks (Xo, Xw, _) =
  (∀ (a, c) ∈ Xw.lo.
    is_successful_lock a ∧ is_successful_lock c
    ⟶
    (∃ b ∈ Xo.actions. is_unlock b ∧ (a, b) ∈ Xw.lo ∧ (b, c) ∈ Xw.lo))
```

According to the standard, the first lock acquires ownership of the mutex, and the second violates the specification by acquiring ownership that was not relinquished with an intervening unlock. Recall that the thread-local semantics generates blocked lock actions at each lock because locks can arbitrarily block on some targets like Power and ARM processors. These block actions cover the cases where locks would block because of deadlock.

B.4.5 Read determinacy

The behaviour of reads from memory is described in **1.10**:

1.10p3

> The value of an object visible to a thread T at a particular point is the initial value of the object, a value assigned to the object by T, or a value assigned to the object by another thread, according to the rules below. [*Note:* In some cases, there may instead be undefined behavior. Much of this section is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs. — *end note*]

This restriction requires reads with a visible side effect to read from some write at the same location. It does not describe the behaviour of the program if there are no writes visible to a given read, but does allude to the possibility of undefined behaviour in the note. In this case, as we shall see in Section B.5.3, the model provides the program with undefined behaviour. For the purposes of calculating the consistent executions, a read that lacks a related write can return any value. The *det_read* predicate forces reads with visible side effects to have a write related to them by reads-from, and leaves all other reads unconstrained:

```
let det_read (Xo, Xw, _ :: ("vse", vse) :: _) =
  ∀ r ∈ Xo.actions.
    is_load r ⟶
    (∃ w ∈ Xo.actions. (w, r) ∈ vse) =
    (∃ w' ∈ Xo.actions. (w', r) ∈ Xw.rf)
```

B.4.6 The full consistency predicate

This section has described the relationship between each part of the consistency predicate and the standard. The predicate, in its entirety, is expressed as a tree whose leaves are name-predicate pairs, and whose branches are names. This allows some predicates to be grouped, and in the tree below, the restrictions that decide consistent read values lie on the *consistent_rf* branch:

```
let standard_consistent_execution =
  Node [ ("assumptions", Leaf assumptions);
         ("tot_empty", Leaf tot_empty);
         ("well_formed_threads", Leaf well_formed_threads);
         ("well_formed_rf", Leaf well_formed_rf);
         ("locks_only_consistent_locks", Leaf locks_only_consistent_locks);
         ("locks_only_consistent_lo", Leaf locks_only_consistent_lo);
         ("consistent_mo", Leaf consistent_mo);
         ("sc_accesses_consistent_sc", Leaf sc_accesses_consistent_sc);
         ("sc_fenced_sc_fences_heeded", Leaf sc_fenced_sc_fences_heeded);
         ("consistent_hb", Leaf consistent_hb);
         ("consistent_rf",
           Node [ ("det_read", Leaf det_read);
                  ("consistent_non_atomic_rf", Leaf consistent_non_atomic_rf);
                  ("standard_consistent_atomic_rf",
                    Leaf standard_consistent_atomic_rf);
                  ("coherent_memory_use", Leaf coherent_memory_use);
                  ("rmw_atomicity", Leaf rmw_atomicity);
```

```
  ( "sc_accesses_sc_reads_restricted",
    Leaf sc_accesses_sc_reads_restricted) ]) ]
```

B.5 Undefined behaviour

The standard introduces undefined behaviour in Section **1.3.24**:

1.3.24

> **undefined behavior**
> behavior for which this International Standard imposes no requirements [*Note:* Undefined behavior may be expected when this International Standard omits any explicit definition of behavior or when a program uses an erroneous construct or erroneous data. [...]— *end note*]

The note is particularly important for the interpretation of future passages: it implies that the standard simply fails to define the behaviour of some programs, but it may identify faults that lead explicitly to undefined behaviour as well. The majority of the undefined behaviour recognised in the model is explicitly identified by the standard. The model works out instances of undefined behaviour in an execution using a named list of functions that identify either a single faulting action, or a pair of actions engaging in a fault together. The type of the elements in this list is *fault_setgen*:

```
type FAULT_SETGEN =
    ONE of (STRING * (CANDIDATE_EXECUTION → SET (ACTION)))
  | TWO of (STRING * (CANDIDATE_EXECUTION → SET (ACTION * ACTION)))
```

The rest of this section identifies sources of undefined behaviour described by the standard, and for each case, relates the text to a function from executions to actions or pairs of actions in the model. The section concludes by drawing all of these sources together in a record that captures all of the undefined behaviour tracked in the model.

B.5.1 Unsequenced races

Section **1.9** identifies insufficient sequenced-before ordering as a source of undefined behaviour:

1.9p15

> [...]If a side effect on a scalar object is unsequenced relative to either another side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined.

This is an instance where undefined behaviour is the result of a pair of actions, so the model captures the instances of undefined behaviour as a relation. In particular, the relation contains pairs of distinct actions at the same location, on the same thread, where at least one of the actions is a write, and the pair are unrelated by sequenced before:

```
let unsequenced_races (Xo, _, _) =
  { (a, b) | ∀ a ∈ Xo.actions b ∈ Xo.actions |
    is_at_non_atomic_location Xo.lk a ∧
    ¬ (a = b) ∧ (loc_of a = loc_of b) ∧ (is_write a ∨ is_write b) ∧
    (tid_of a = tid_of b) ∧
    ¬ ((a, b) ∈ Xo.sb ∨ (b, a) ∈ Xo.sb) }
```

B.5.2 Data races

Concurrent programs must order their non-atomic accesses to memory sufficiently, or face undefined behaviour. Pairs of accesses with insufficient happens-before ordering are faults, called data races. The standard defines them in terms of conflicting accesses in Section **1.10**:

1.10p4

> Two expression evaluations *conflict* if one of them modifies a memory location (1.7) and the other one accesses or modifies the same memory location.

The definition of data race is given later in the section:

1.10p21

> The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior. [...]

The model unpacks the definition of conflict, and defines the set of data races in a program as the set of pairs of distinct actions at the same location where the actions are on different threads, at least one is a write, at most one is atomic, and where the pair is unrelated by happens-before:

```
let data_races (Xo, Xw, ("hb", hb) :: _) =
  { (a, b) | ∀ a ∈ Xo.actions b ∈ Xo.actions |
    ¬ (a = b) ∧ (loc_of a = loc_of b) ∧ (is_write a ∨ is_write b) ∧
    (tid_of a ≠ tid_of b) ∧
    ¬ (is_atomic_action a ∧ is_atomic_action b) ∧
    ¬ ((a, b) ∈ hb ∨ (b, a) ∈ hb) }
```

Racy initialisation It is important to note that initialisation on atomic locations is performed with a non-atomic write, that will race with happens-before unordered atomic accesses of the location. The standard provides two ways to initialise atomic variables. The first is via a macro:

29.6.5p6

> `#define ATOMIC_VAR_INIT(value)` see below
> The macro expands to a token sequence suitable for constant initialization of an atomic variable of static storage duration of a type that is initialization-compatible with value. [*Note:* This operation may need to initialize locks. — *end note*] Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race. [*Example:*
> `atomic<int> v = ATOMIC_VAR_INIT(5);`
> — *end example*]

Here the standard explicitly identifies the possibility of a data race with the initialisation. The introduction of the other syntax for atomic initialisation includes a similar note. This syntax allows the initialisation of an existing atomic object:

29.6.5p8

> `void atomic_init(volatile A *object, C desired) noexcept;`
> `void atomic_init(A *object, C desired) noexcept;`
> Effects: Non-atomically initializes `*object` with value `desired`. [. . .][*Note:* Concurrent access from another thread, even via an atomic operation, constitutes a data race. — *end note*]

The model captures these potential races in the thread-local semantics by expressing initialisation writes at atomic locations as non-atomic writes.

B.5.3 Indeterminate read

The standard defines which writes a given read may read from when there is a visible side effect of the read. In the absence of such a side effect, the standard is not precise. The following paragraph defines which values a read may take in the normative text, and then proclaims that some unspecified cases have undefined behaviour in the following note.

1.10p3

> The value of an object visible to a thread T at a particular point is the initial value of the object, a value assigned to the object by T, or a value assigned to the object by another thread, according to the rules below. [*Note:* In some cases, there may instead be undefined behavior. [[. . .]] — *end note*]

In the model, we provide programs that have reads with no visible side effects with undefined behaviour:

$$\textsf{let}\ \mathit{indeterminate\_reads}\ (Xo,\ Xw,\ \_)\ =$$
$$\{b \mid \forall\ b \in Xo.\mathit{actions} \mid \text{is\_read}\ b \wedge \neg\ (\exists\ a \in Xo.\mathit{actions}.\ (a,\ b) \in Xw.\mathit{rf})\}$$

B.5.4 Bad mutex use

The standard places preconditions on the use of locks and unlocks over mutex locations. If a program contains an execution that does not satisfy these preconditions then Section **17.6.4.11** tells us that the program has undefined behaviour:

17.6.4.11p1

> Violation of the preconditions specified in a function Requires: paragraph results in undefined behavior unless the function Throws: paragraph specifies throwing an exception when the precondition is violated.

The requirement over calls to `m.lock()` is provided in Section **30.4.1.2**:

30.4.1.2p7

> *Requires:* If `m` is of type `std::mutex` or `std::timed_mutex`, the calling thread does not own the mutex.

The requirement over calls to `m.unlock()` is later in the same section:

30.4.1.2p22

> *Requires:* The calling thread shall own the mutex.

The ownership of the mutex is embedded in the lock ordering over the mutex actions. Consequently, these sources of undefined behaviour are identified in the model using the lock order. The model first defines what it is to be a good mutex:

```
let locks_only_good_mutex_use actions lk sb lo a =
  (* violated requirement: The calling thread shall own the mutex. *)
  ( is_unlock a
    ⟶
    ( ∃ al ∈ actions.
        is_successful_lock al ∧ (al, a) ∈ sb ∧ (al, a) ∈ lo ∧
        ∀ au ∈ actions.
          is_unlock au ⟶ ¬ ((al, au) ∈ lo ∧ (au, a) ∈ lo)
    )
  ) ∧
    (* violated requirement: The calling thread does not own the mutex.
*)
  ( is_lock a
    ⟶
    ∀ al ∈ actions.
      is_successful_lock al ∧ (al, a) ∈ sb ∧ (al, a) ∈ lo
      ⟶
      ∃ au ∈ actions.
        is_unlock au ∧ (al, au) ∈ lo ∧ (au, a) ∈ lo
  )
```

The first conjunct requires every unlock to be immediately preceded in lock order by a lock that is sequenced before the unlock. This implies that the program should not unlock a mutex twice. The second conjunct requires every pair of locks where the first is

successful to have an intervening unlock. This implies that the program should not lock a mutex twice. Faulty mutex accesses are then identified as the set of accesses that fail the *good_mutex_use* requirement:

```
let locks_only_bad_mutexes (Xo, Xw, _) =
  { a | ∀ a ∈ Xo.actions |
    ¬ (locks_only_good_mutex_use Xo.actions Xo.lk Xo.sb Xw.lo a)}
```

B.5.5 Undefined Behaviour

The undefined behaviour captured by the standard model is collected together in the following list:

```
let locks_only_undefined_behaviour =
  [ Two ("unsequenced_races", unsequenced_races);
    Two ("data_races", data_races);
    One ("indeterminate_reads", indeterminate_reads);
    One ("locks_only_bad_mutexes", locks_only_bad_mutexes) ]
```

B.6 The memory model

The information discussed in the previous sections is collected together in a single record that represents the memory model that the standard uses. This contains the consistency predicate, the relation calculation, the functions that identify undefined behaviour and flags that identify which relations are used in the execution witness:

```
let standard_memory_model =
  ⦃ consistent = standard_consistent_execution;
    relation_calculation = standard_relations;
    undefined = locks_only_undefined_behaviour;
    relation_flags =
      ⦃ rf_flag = true;
        mo_flag = true;
        sc_flag = true;
        lo_flag = true;
        tot_flag = false ⦄
  ⦄
```

B.7 Top-level judgement

At the highest level, the standard claims to describe the behaviour of a program in terms of an abstract machine. Section **1.9** reads:

1.9p1

> The semantic descriptions in this International Standard define a parameterised nondeterministic abstract machine. [...]

This is not quite true: the standard does not describe a machine that can be incrementally executed; instead, the language is defined in terms of candidate executions. Nonetheless, a later passage describes the behaviour of programs that contain undefined behaviour, and refers to the definition in the standard as an abstract machine:

1.9p5

> A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

This passage justifies the top-level judgement of the model: the set of consistent executions of a program is calculated, and if any of these executions contains an instance of undefined behaviour then the behaviour of the program is undefined, otherwise, the behaviour is the set of consistent executions.

The model captures this with a function that takes a memory-model record, a model condition, a thread-local semantics, and a program. The function calculates the consistent executions of the program whose pre-executions are valid according to the thread-local semantics. The function then checks for model condition violations or undefined behaviour and returns either the set of consistent executions, or undefined behaviour:

```
let behaviour M condition opsem (p : PROGRAM) =
  let consistent_executions =
    { (Xo, Xw, rl) |
        opsem p Xo ∧
        apply_tree M.consistent (Xo, Xw, rl) ∧
        rl = M.relation_calculation Xo Xw } in
  if condition consistent_executions ∧
    ∀ X ∈ consistent_executions.
```

```
        each_empty M.undefined X
  then Defined (observable_filter consistent_executions)
  else Undefined
```

There are no model condition restrictions on programs that use the standard model, so in presenting the function from programs to their behaviour under the memory model from the standard, the following model condition is used:

```
let true_condition _ = true
```

Appendix C

Lem models

This appendix reproduces the formal definitions that make up the memory models of Chapter 3 and the theorems of Chapter 6. These definitions are automatically typeset from the Lem definitions.

C.1 Auxiliary definitions

let $inj\_on\ f\ A\ =\ (\forall\ x \in A.\ (\forall\ y \in A.\ (f\ x = f\ y) \longrightarrow (x = y)))$

let $strict\_total\_order\_over\ s\ ord\ =$
$\quad$ relation_over $s\ ord \wedge$ isTotalOrderOn $ord\ s$

let $adjacent\_less\_than\ ord\ s\ x\ y\ =$
$\quad (x,\ y) \in ord \wedge \neg\ (\exists\ z \in s.\ (x,\ z) \in ord \wedge (z,\ y) \in ord)$

let $adjacent\_less\_than\_such\_that\ pred\ ord\ s\ x\ y\ =$
$\quad pred\ x \wedge (x,\ y) \in ord \wedge \neg\ (\exists\ z \in s.\ pred\ z \wedge (x,\ z) \in ord \wedge (z,\ y) \in ord)$

val $finite\_prefixes\ :\ \forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{SET}\ (\alpha\ *\ \alpha)\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{BOOL}$

let $finite\_prefixes\ r\ s\ =$
$\quad \forall\ b \in s.$ finite $\{\ a \mid \forall\ a \mid (a,\ b) \in r\}$

val $minimal\_elements : \forall\ \alpha.\ \text{SET}\ \alpha\ \rightarrow\ \text{SET}\ (\alpha\ *\ \alpha)\ \rightarrow\ \text{SET}\ \alpha$

let $minimal\_elements\ s\ r\ =\ s$

C.2 Types

The base types are defined for each backend. The base types for the HOL backend are reproduced here.

type AID_IMPL = STRING

```
type PROGRAM_IMPL = NAT

type TID_IMPL = TID_HOL of NAT

type LOCATION_IMPL = LOC_HOL of NAT

type CVALUE_IMPL = CVALUE_HOL of NAT

type AID = AID_IMPL

type PROGRAM = PROGRAM_IMPL

type TID = TID_IMPL

type LOCATION = LOCATION_IMPL

type CVALUE = CVALUE_IMPL

type MEMORY_ORDER =
  | NA
  | SEQ_CST
  | RELAXED
  | RELEASE
  | ACQUIRE
  | CONSUME
  | ACQ_REL

type LOCK_OUTCOME =
    LOCKED
  | BLOCKED

type ACTION =
  | LOCK of AID * TID * LOCATION * LOCK_OUTCOME
  | UNLOCK of AID * TID * LOCATION
  | LOAD of AID * TID * MEMORY_ORDER * LOCATION * CVALUE
  | STORE of AID * TID * MEMORY_ORDER * LOCATION * CVALUE
  | RMW of AID * TID * MEMORY_ORDER * LOCATION * CVALUE * CVALUE
  | FENCE of AID * TID * MEMORY_ORDER
  | BLOCKED_RMW of AID * TID * LOCATION

type LOCATION_KIND =
    MUTEX
  | NON_ATOMIC
  | ATOMIC

type PRE_EXECUTION =
  {| actions : SET (ACTION);
     threads : SET (TID);
```

```
    lk : LOCATION → LOCATION_KIND;
    sb : SET (ACTION * ACTION) ;
    asw : SET (ACTION * ACTION) ;
    dd : SET (ACTION * ACTION) ;
  |⟩

type ORDER_KIND =
    GLOBAL_ORDER
  | PER_LOCATION_ORDER

type RELATION_USAGE_FLAGS =
  ⟨| rf_flag : BOOL;
    mo_flag : BOOL;
    sc_flag : BOOL;
    lo_flag : BOOL;
    tot_flag : BOOL; |⟩

type EXECUTION_WITNESS =
  ⟨| rf : SET (ACTION * ACTION);
    mo : SET (ACTION * ACTION);
    sc : SET (ACTION * ACTION);
    lo : SET (ACTION * ACTION);
    tot : SET (ACTION * ACTION);
  |⟩

type RELATION_LIST = LIST (STRING * SET (ACTION * ACTION))

type CANDIDATE_EXECUTION = (PRE_EXECUTION * EXECUTION_WITNESS *
RELATION_LIST)

type OBSERVABLE_EXECUTION = (PRE_EXECUTION * EXECUTION_WITNESS)

type PROGRAM_BEHAVIOURS =
    DEFINED of SET (OBSERVABLE_EXECUTION)
  | UNDEFINED

type RF_OBSERVABLE_EXECUTION = (PRE_EXECUTION * SET (ACTION * ACTION))

type RF_PROGRAM_BEHAVIOURS =
    RF_DEFINED of SET (RF_OBSERVABLE_EXECUTION)
  | RF_UNDEFINED

type NAMED_PREDICATE_TREE =
    LEAF of (CANDIDATE_EXECUTION → BOOL)
  | NODE of LIST (STRING * NAMED_PREDICATE_TREE)

val named_predicate_tree_measure : ∀. NAMED_PREDICATE_TREE → NAT
```

```
let named_predicate_tree_measure t =
  match t with
    | Leaf _ → 0
    | Node l → 1 + length l
  end

let rec apply_tree pred_tree X =
  match pred_tree with
    | Leaf p → p X
    | Node l → List.all (fun (name, branch) → apply_tree branch X) l
  end

type FAULT_SETGEN =
    ONE of (STRING * (CANDIDATE_EXECUTION → SET (ACTION)))
  | TWO of (STRING * (CANDIDATE_EXECUTION → SET (ACTION * ACTION)))

let is_fault faults_list (Xo, Xw, rl) a =
    let is_particular_fault f =
      match f with
      | One (_name, setgen) → (a ∈ (setgen (Xo, Xw, rl)))
      | Two (_name, setgen) →
          ∃ b ∈ Xo.actions.
            ((a, b) ∈ (setgen (Xo, Xw, rl))) ∨ ((b, a) ∈ (setgen (Xo, Xw, rl))) end in
    List.any is_particular_fault faults_list

let each_empty faults_list X =
    let faults_empty f =
      match f with
      | One (_name, setgen) → null (setgen X)
      | Two (_name, setgen) → null (setgen X) end in
    List.all faults_empty faults_list

type OPSEM_T = PROGRAM → PRE_EXECUTION → BOOL

type CONDITION_T = SET CANDIDATE_EXECUTION → BOOL

let true_condition _ = true

val statically_satisfied : ∀. CONDITION_T → OPSEM_T → PROGRAM → BOOL

let statically_satisfied condition opsem (p : PROGRAM) =
  let Xs = {(Xo, Xw, rl) | opsem p Xo} in
  condition Xs

type MEMORY_MODEL =
  ⦃ consistent : NAMED_PREDICATE_TREE;
```

relation_calculation : PRE_EXECUTION → EXECUTION_WITNESS → RELATION_LIST;
undefined : LIST (FAULT_SETGEN);
relation_flags : RELATION_USAGE_FLAGS;
⟫

val *observable_filter* : ∀. SET (CANDIDATE_EXECUTION) → SET (OBSERVABLE_EXECUTION)

let *observable_filter X* = $\{(Xo, Xw) \mid \exists\, rl.\ (Xo, Xw, rl) \in X\}$

val *behaviour* : ∀. MEMORY_MODEL → CONDITION_T → OPSEM_T → PROGRAM → PROGRAM_BEHAVIOURS

let *behaviour M condition opsem* (*p* : PROGRAM) =
  let *consistent_executions* =
    { (*Xo*, *Xw*, *rl*) |
      opsem p Xo ∧
      apply_tree *M.consistent* (*Xo*, *Xw*, *rl*) ∧
      rl = *M.relation_calculation Xo Xw* } in
  if *condition consistent_executions* ∧
    ∀ *X* ∈ *consistent_executions*.
      each_empty *M.undefined X*
  then Defined (observable_filter *consistent_executions*)
  else Undefined

val *rf_observable_filter* : ∀. SET (CANDIDATE_EXECUTION) → SET (RF_OBSERVABLE_EXECUTION)

let *rf_observable_filter X* = $\{(Xo, Xw.rf) \mid \exists\, rl.\ (Xo, Xw, rl) \in X\}$

val *rf_behaviour* : ∀. MEMORY_MODEL → CONDITION_T → OPSEM_T → PROGRAM → RF_PROGRAM_BEHAVIOURS

let *rf_behaviour M condition opsem* (*p* : PROGRAM) =
  let *consistent_executions* =
    { (*Xo*, *Xw*, *rl*) |
      opsem p Xo ∧
      apply_tree *M.consistent* (*Xo*, *Xw*, *rl*) ∧
      rl = *M.relation_calculation Xo Xw* } in
  if *condition consistent_executions* ∧
    ∀ *X* ∈ *consistent_executions*.
      each_empty *M.undefined X*
  then rf_Defined (rf_observable_filter *consistent_executions*)
  else rf_Undefined

C.3 Projection functions

```
let aid_of a =
  match a with
  | Lock aid _ _ _ → aid
  | Unlock aid _ _ → aid
  | Load aid _ _ _ _ → aid
  | Store aid _ _ _ _ → aid
  | RMW aid _ _ _ _ _ → aid
  | Fence aid _ _ → aid
  | Blocked_rmw aid _ _ → aid
  end

let tid_of a =
  match a with
  | Lock _ tid _ _ → tid
  | Unlock _ tid _ → tid
  | Load _ tid _ _ _ → tid
  | Store _ tid _ _ _ → tid
  | RMW _ tid _ _ _ _ → tid
  | Fence _ tid _ → tid
  | Blocked_rmw _ tid _ → tid
  end

let loc_of a =
  match a with
  | Lock _ _ l _ → Just l
  | Unlock _ _ l → Just l
  | Load _ _ _ l _ → Just l
  | Store _ _ _ l _ → Just l
  | RMW _ _ _ l _ _ → Just l
  | Fence _ _ _ → Nothing
  | Blocked_rmw _ _ l → Just l
  end

let value_read_by a =
  match a with
  | Load _ _ _ _ v → Just v
  | RMW _ _ _ _ v _ → Just v
  | _ → Nothing
  end

let value_written_by a =
```

```
    match a with
    | Store _ _ _ _ v → Just v
    | RMW _ _ _ _ _ v → Just v
    | _ → Nothing
    end

let is_lock a =
    match a with
    | Lock _ _ _ _ → true
    | _ → false
    end

let is_successful_lock a =
    match a with
    | Lock _ _ _ Locked → true
    | _ → false
    end

let is_blocked_lock a =
    match a with
    | Lock _ _ _ Blocked → true
    | _ → false
    end

let is_unlock a =
    match a with
    | Unlock _ _ _ → true
    | _ → false
    end

let is_atomic_load a =
    match a with
    | Load _ _ mo _ _ → mo ≠ NA
    | _ → false
    end

let is_atomic_store a =
    match a with
    | Store _ _ mo _ _ → mo ≠ NA
    | _ → false
    end

let is_RMW a =
    match a with
```

```
  | RMW _ _ _ _ _ _ → true
  | _ → false
  end

let is_blocked_rmw a =
  match a with
  | Blocked_rmw _ _ _ → true
  | _ → false
  end

let is_NA_load a =
  match a with
  | Load _ _ mo _ _ → mo = NA
  | _ → false
  end

let is_NA_store a =
  match a with
  | Store _ _ mo _ _ → mo = NA
  | _ → false
  end

let is_load a =
  match a with
  | Load _ _ _ _ _ → true
  | _ → false
  end

let is_store a =
  match a with
  | Store _ _ _ _ _ → true
  | _ → false
  end

let is_fence a =
  match a with
  | Fence _ _ _ → true
  | _ → false
  end

let is_atomic_action a =
  match a with
  | Load _ _ mo _ _ → mo ≠ NA
  | Store _ _ mo _ _ → mo ≠ NA
```

```
  | RMW _ _ _ _ _ _ → true
  | Blocked_rmw _ _ _ → true
  | _ → false
  end

let is_read a =
  match a with
  | Load _ _ _ _ _ → true
  | RMW _ _ _ _ _ _ → true
  | _ → false
  end

let is_write a =
  match a with
  | Store _ _ _ _ _ → true
  | RMW _ _ _ _ _ _ → true
  | _ → false
  end

let is_acquire a =
  match a with
  | Load _ _ mo _ _ → mo ∈ {Acquire, Seq_cst}
  | RMW _ _ mo _ _ _ → mo ∈ {Acquire, Acq_rel, Seq_cst}
  | Fence _ _ mo → mo ∈ {Acquire, Consume, Acq_rel, Seq_cst}
  | _ → false
  end

let is_release a =
  match a with
  | Store _ _ mo _ _ → mo ∈ {Release, Seq_cst}
  | RMW _ _ mo _ _ _ → mo ∈ {Release, Acq_rel, Seq_cst}
  | Fence _ _ mo → mo ∈ {Release, Acq_rel, Seq_cst}
  | _ → false
  end

let is_consume a =
  match a with
  | Load _ _ mo _ _ → mo = Consume
  | _ → false
  end

let is_seq_cst a =
  match a with
  | Load _ _ mo _ _ → mo = Seq_cst
```

```
  | Store _ _ mo _ _ → mo = Seq_cst
  | RMW _ _ mo _ _ _ → mo = Seq_cst
  | Fence _ _ mo → mo = Seq_cst
  | _ → false
  end
```

C.4 Well-formed threads

```
let threadwise s rel = ∀ (a, b) ∈ rel. tid_of a = tid_of b

let interthread s rel = ∀ (a, b) ∈ rel. tid_of a ≠ tid_of b

let locationwise s rel = ∀ (a, b) ∈ rel. loc_of a = loc_of b

let per_location_total s rel =
  ∀ a ∈ s b ∈ s. loc_of a = loc_of b ⟶
    (a, b) ∈ rel ∨ (b, a) ∈ rel ∨ (a = b)

let actions_respect_location_kinds actions lk =
∀ a ∈ actions. match a with
  | Lock _ _ l _ → lk l = Mutex
  | Unlock _ _ l → lk l = Mutex
  | Load _ _ mo l _ →
      (mo = NA ∧ lk l = Non_Atomic) ∨ (mo ≠ NA ∧ lk l = Atomic)
  | Store _ _ mo l _ →
      (mo = NA ∧ lk l = Non_Atomic) ∨ lk l = Atomic
  | RMW _ _ _ l _ _ → lk l = Atomic
  | Fence _ _ _ → true
  | Blocked_rmw _ _ l → lk l = Atomic
  end

let is_at_mutex_location lk a =
  match loc_of a with
  | Just l → (lk l = Mutex)
  | Nothing → false
  end

let is_at_non_atomic_location lk a =
  match loc_of a with
  | Just l → (lk l = Non_Atomic)
  | Nothing → false
  end

let is_at_atomic_location lk a =
```

```
    match loc_of a with
    | Just l → (lk l = Atomic)
    | Nothing → false
    end

let locations_of actions =
{ l | ∀ Just l ∈ { (loc_of a) | ∀ a ∈ actions | true } | true}

let well_formed_action a =
  match a with
    | Load _ _ mo _ _ → mo ∈ {NA, Relaxed, Acquire, Seq_cst, Consume}
    | Store _ _ mo _ _ → mo ∈ {NA, Relaxed, Release, Seq_cst}
    | RMW _ _ mo _ _ _ → mo ∈ {Relaxed, Release, Acquire, Acq_rel, Seq_cst}
    | Fence _ _ mo → mo ∈ {Relaxed, Release, Acquire, Acq_rel, Consume, Seq_cst}
    | _ → true
    end

val assumptions : (PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST) → BOOL

let assumptions (Xo, Xw, _) =
  finite_prefixes Xw.rf Xo.actions ∧
  finite_prefixes Xw.mo Xo.actions ∧
  finite_prefixes Xw.sc Xo.actions ∧
  finite_prefixes Xw.lo Xo.actions

let blocking_observed actions sb =
    (∀ a ∈ actions.
      (is_blocked_rmw a ∨ is_blocked_lock a)
      ⟶
      ¬ (∃ b ∈ actions. (a, b) ∈ sb))

let indeterminate_sequencing Xo =
    ∀ a ∈ Xo.actions b ∈ Xo.actions.
      (tid_of a = tid_of b) ∧ (a ≠ b) ∧
      ¬ (is_at_non_atomic_location Xo.lk a ∧ is_at_non_atomic_location Xo.lk b) ⟶
      (a, b) ∈ Xo.sb ∨ (b, a) ∈ Xo.sb

let sbasw Xo = transitiveClosure (Xo.sb ∪ Xo.asw)

val well_formed_threads : (PRE_EXECUTION * EXECUTION_WITNESS *
RELATION_LIST) → BOOL

let well_formed_threads ((Xo, _, _) : (PRE_EXECUTION * EXECUTION_WITNESS *
RELATION_LIST)) =
    (∀ a ∈ Xo.actions. well_formed_action a) ∧
    actions_respect_location_kinds Xo.actions Xo.lk ∧
```

blocking_observed $Xo.actions$ $Xo.sb$ $\wedge$
inj_on aid_of $Xo.actions$ $\wedge$
relation_over $Xo.actions$ $Xo.sb$ $\wedge$
relation_over $Xo.actions$ $Xo.asw$ $\wedge$
threadwise $Xo.actions$ $Xo.sb$ $\wedge$
interthread $Xo.actions$ $Xo.asw$ $\wedge$
isStrictPartialOrder $Xo.sb$ $\wedge$
isStrictPartialOrder $Xo.dd$ $\wedge$
$Xo.dd \subseteq Xo.sb$ $\wedge$
indeterminate_sequencing Xo $\wedge$
isIrreflexive (sbasw Xo) $\wedge$
finite_prefixes (sbasw Xo) $Xo.actions$

C.5 Assumptions on the thread-local semantics for Theorem 13

```
let pre_execution_mask Xo A =
  let B = A ∩ Xo.actions in
  ⦃ actions = B;
    threads = Xo.threads;
    lk = Xo.lk;
    sb = relRestrict Xo.sb B;
    asw = relRestrict Xo.asw B;
    dd = relRestrict Xo.dd B
  ⦄

let replace_read_value a v =
  match a with
  | Lock aid tid loc out → Lock aid tid loc Blocked
  | Unlock aid tid loc → a
  | Load aid tid ord loc rval → Load aid tid ord loc v
  | Store aid tid ord loc wval → a
  | RMW aid tid ord loc rval wval → RMW aid tid ord loc v wval
  | Fence aid tid ord → a
  | Blocked_rmw aid tid loc → a
  end

val downclosed : ∀ . SET (ACTION) → SET (ACTION * ACTION) → BOOL

let
    downclosed A R = ∀ a b. b ∈ A ∧ (a, b) ∈ R ⟶ a ∈ A

let is_prefix opsem p Xo A =
```

$opsem\ p\ Xo \wedge A \subseteq Xo.actions \wedge$ downclosed A (sbasw Xo) $\wedge$ finite A

let *fringe_set* $Xo\ A\ =$ minimal_elements ($\setminus\ Xo.actions\ A$) (sbasw Xo)

val *relation_plug* : $\forall$. SET (ACTION $*$ ACTION) $\rightarrow$ ACTION $\rightarrow$ ACTION $\rightarrow$ SET (ACTION $*$ ACTION)

let

$relation\_plug\ R\ a\ a' =$

$\{\ (x,\ y) \mid ((x,\ y) \in R \wedge (x \neq a) \wedge (y \neq a)) \vee$

$((a,\ y) \in R \wedge (x = a') \wedge (y \neq a)) \vee$

$((x,\ a) \in R \wedge (x \neq a) \wedge (y = a')) \vee$

$((a,\ a) \in R \wedge (x = a') \wedge (y = a'))$

$\}$

let

$relation\_plug\ R\ a\ a' = \{\}$

let *pre_execution_plug* $Xo\ a\ a' =$

$\langle\!|$ $actions = (\setminus\ Xo.actions\ \{a\}) \cup \{a'\};$

$threads = Xo.threads;$

$lk = Xo.lk;$

$sb =$ relation_plug $Xo.sb\ a\ a';$

$asw =$ relation_plug $Xo.asw\ a\ a';$

$dd =$ relation_plug $Xo.dd\ a\ a'$

$|\!\rangle$

let *same_prefix* $Xo_1\ Xo_2\ A =$

let $AF = A \cup$ fringe_set $Xo_1\ A$ in

(pre_execution_mask $Xo_1\ AF$ = pre_execution_mask $Xo_2\ AF$) $\wedge$

(fringe_set $Xo_1\ A$ = fringe_set $Xo_2\ A$)

val *receptiveness* : $\forall$. (PROGRAM $\rightarrow$ PRE_EXECUTION $\rightarrow$ BOOL) $\rightarrow$ BOOL

let

$receptiveness\ opsem =$

$\forall\ p\ Xo\ A\ a.$

is_prefix $opsem\ p\ Xo\ A \wedge$

$a \in$ fringe_set $Xo\ A \wedge$

(is_read $a \vee$ is_successful_lock a)

$\longrightarrow$

$\forall\ v.$

let $a' =$ replace_read_value $a\ v$ in

$\exists\ Xo'.$

is_prefix $opsem\ p\ Xo'\ A \wedge$

```
    a' ∈ fringe_set Xo' A ∧
    same_prefix Xo' (pre_execution_plug Xo a a') A

let holds_over_prefix opsem p Xo A P =
  is_prefix opsem p Xo A ∧ P (pre_execution_mask Xo A)

val extends_prefix : ∀. PRE_EXECUTION → SET (ACTION) → SET (ACTION) → BOOL

let
    extends_prefix Xo A A' =
  let fs = fringe_set Xo A in
  fs ≠ {} ∧
  ∃ fs'.
    (∀ a. a ∈ fs ⟶ a ∈ fs' ∨ ∃ v. replace_read_value a v ∈ fs') ∧
    (A ∪ fs') ⊆ A'

val produce_well_formed_threads : ∀. OPSEM_T → BOOL

let
    produce_well_formed_threads (opsem : OPSEM_T) =
    ∀ Xo p. ∃ Xw rl. opsem p Xo ⟶ well_formed_threads (Xo, Xw, rl)

let opsem_assumptions opsem =
  receptiveness opsem ∧
  produce_well_formed_threads opsem
```

C.6 Single-thread memory model

```
let visible_side_effect_set actions hb =
    { (a, b) | ∀ (a, b) ∈ hb |
      is_write a ∧ is_read b ∧ (loc_of a = loc_of b) ∧
      ¬ ( ∃ c ∈ actions. ¬ (c ∈ {a, b}) ∧
          is_write c ∧ (loc_of c = loc_of b) ∧
          (a, c) ∈ hb ∧ (c, b) ∈ hb) }

val det_read : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let det_read (Xo, Xw, _ :: ("vse", vse) :: _) =
    ∀ r ∈ Xo.actions.
      is_load r ⟶
      (∃ w ∈ Xo.actions. (w, r) ∈ vse) =
      (∃ w' ∈ Xo.actions. (w', r) ∈ Xw.rf)

val consistent_non_atomic_rf : PRE_EXECUTION * EXECUTION_WITNESS *
RELATION_LIST → BOOL
```

```
let consistent_non_atomic_rf (Xo, Xw, _ :: ("vse", vse) :: _) =
  ∀ (w, r) ∈ Xw.rf. is_at_non_atomic_location Xo.lk r ⟶
    (w, r) ∈ vse

val well_formed_rf : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let well_formed_rf (Xo, Xw, _) =
  ∀ (a, b) ∈ Xw.rf.
    a ∈ Xo.actions ∧ b ∈ Xo.actions ∧
    loc_of a = loc_of b ∧
    is_write a ∧ is_read b ∧
    value_read_by b = value_written_by a ∧
    ∀ a' ∈ Xo.actions. (a', b) ∈ Xw.rf ⟶ a = a'

val sc_mo_lo_empty : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let sc_mo_lo_empty (_, Xw, _) = null Xw.sc ∧ null Xw.mo ∧ null Xw.lo

val sc_mo_empty : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let sc_mo_empty (_, Xw, _) = null Xw.sc ∧ null Xw.mo

val sc_empty : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let sc_empty (_, Xw, _) = (null Xw.sc)

val tot_empty : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let tot_empty (_, Xw, _) = (null Xw.tot)

let single_thread_relations Xo Xw =
  let hb = Xo.sb in
  let vse = visible_side_effect_set Xo.actions hb in
  [ ("hb", hb);
    ("vse", vse) ]

let single_thread_consistent_execution =
 Node [ ("assumptions", Leaf assumptions);
        ("sc_mo_lo_empty", Leaf sc_mo_lo_empty);
        ("tot_empty", Leaf tot_empty);
        ("well_formed_threads", Leaf well_formed_threads);
        ("well_formed_rf", Leaf well_formed_rf);
        ("consistent_rf",
          Node [ ("det_read", Leaf det_read);
                 ("consistent_non_atomic_rf", Leaf consistent_non_atomic_rf) ]) ]
```

```
val indeterminate_reads : CANDIDATE_EXECUTION → SET ACTION

let indeterminate_reads (Xo, Xw, _) =
  {b | ∀ b ∈ Xo.actions | is_read b ∧ ¬ (∃ a ∈ Xo.actions. (a, b) ∈ Xw.rf)}

val unsequenced_races : CANDIDATE_EXECUTION → SET (ACTION * ACTION)

let unsequenced_races (Xo, _, _) =
  { (a, b) | ∀ a ∈ Xo.actions b ∈ Xo.actions |
    is_at_non_atomic_location Xo.lk a ∧
    ¬ (a = b) ∧ (loc_of a = loc_of b) ∧ (is_write a ∨ is_write b) ∧
    (tid_of a = tid_of b) ∧
    ¬ ((a, b) ∈ Xo.sb ∨ (b, a) ∈ Xo.sb) }

let single_thread_undefined_behaviour =
 [ Two ("unsequenced_races", unsequenced_races);
   One ("indeterminate_reads", indeterminate_reads) ]

val single_thread_condition : ∀. CONDITION_T

let single_thread_condition (Xs : SET CANDIDATE_EXECUTION) =
 ∀ (Xo, Xw, rl) ∈ Xs.
 ∃ b ∈ Xo.actions. ∀ a ∈ Xo.actions.
   (tid_of a = tid_of b) ∧
   match (loc_of a) with
   | Nothing → false
   | Just l → (Xo.lk l = Non_Atomic)
   end

let single_thread_memory_model =
 ⦃ consistent = single_thread_consistent_execution;
   relation_calculation = single_thread_relations;
   undefined = single_thread_undefined_behaviour;
   relation_flags =
     ⦃ rf_flag = true;
       mo_flag = false;
       sc_flag = false;
       lo_flag = false;
       tot_flag = false ⦄
 ⦄

val single_thread_behaviour : ∀. OPSEM_T → PROGRAM → PROGRAM_BEHAVIOURS

let single_thread_behaviour opsem (p : PROGRAM) =
 behaviour single_thread_memory_model single_thread_condition opsem p
```

C.7 Locks-only memory model

```
let locks_only_sw actions asw lo a b =
  (tid_of a ≠ tid_of b) ∧
  ( (* thread sync *)
    (a, b) ∈ asw ∨
    (* mutex sync *)
    (is_unlock a ∧ is_lock b ∧ (a, b) ∈ lo)
  )

let locks_only_sw_set actions asw lo =
    { (a, b) | ∀ a ∈ actions b ∈ actions |
       locks_only_sw actions asw lo a b }

let no_consume_hb sb sw =
    transitiveClosure (sb ∪ sw)

let locks_only_relations Xo Xw =
    let sw = locks_only_sw_set Xo.actions Xo.asw Xw.lo in
    let hb = no_consume_hb Xo.sb sw in
    let vse = visible_side_effect_set Xo.actions hb in
    [ ("hb", hb);
      ("vse", vse);
      ("sw", sw) ]

let locks_only_consistent_lo (Xo, Xw, ("hb", hb) :: _) =
    relation_over Xo.actions Xw.lo ∧
    isTransitive Xw.lo ∧
    isIrreflexive Xw.lo ∧
    ∀ a ∈ Xo.actions b ∈ Xo.actions.
      ((a, b) ∈ Xw.lo ⟶ ¬ ((b, a) ∈ hb)) ∧
      ( ((a, b) ∈ Xw.lo ∨ (b, a) ∈ Xw.lo)
        =
        ( (¬ (a = b)) ∧
          (is_lock a ∨ is_unlock a) ∧
          (is_lock b ∨ is_unlock b) ∧
          (loc_of a = loc_of b) ∧
          is_at_mutex_location Xo.lk a
        )
      )
```

val *locks_only_consistent_locks* : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

```
let locks_only_consistent_locks (Xo, Xw, _) =
  (∀ (a, c) ∈ Xw.lo.
    is_successful_lock a ∧ is_successful_lock c
    ⟶
    (∃ b ∈ Xo.actions. is_unlock b ∧ (a, b) ∈ Xw.lo ∧ (b, c) ∈ Xw.lo))

val consistent_hb : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let consistent_hb (Xo, _, ("hb", hb) :: _) =
  isIrreflexive (transitiveClosure hb)

let locks_only_consistent_execution =
 Node [ ("assumptions", Leaf assumptions);
      ("sc_mo_empty", Leaf sc_mo_empty);
      ("tot_empty", Leaf tot_empty);
      ("well_formed_threads", Leaf well_formed_threads);
      ("well_formed_rf", Leaf well_formed_rf);
      ("locks_only_consistent_locks", Leaf locks_only_consistent_locks);
      ("locks_only_consistent_lo", Leaf locks_only_consistent_lo);
      ("consistent_hb", Leaf consistent_hb);
      ("consistent_rf",
       Node [ ("det_read", Leaf det_read);
            ("consistent_non_atomic_rf", Leaf consistent_non_atomic_rf) ]) ]

let locks_only_good_mutex_use actions lk sb lo a =
  (* violated requirement: The calling thread shall own the mutex. *)
  ( is_unlock a
    ⟶
    ( ∃ al ∈ actions.
       is_successful_lock al ∧ (al, a) ∈ sb ∧ (al, a) ∈ lo ∧
       ∀ au ∈ actions.
         is_unlock au ⟶ ¬ ((al, au) ∈ lo ∧ (au, a) ∈ lo)
    )
  ) ∧
  (* violated requirement: The calling thread does not own the mutex. *)
  ( is_lock a
    ⟶
    ∀ al ∈ actions.
      is_successful_lock al ∧ (al, a) ∈ sb ∧ (al, a) ∈ lo
      ⟶
      ∃ au ∈ actions.
        is_unlock au ∧ (al, au) ∈ lo ∧ (au, a) ∈ lo
```

```
  )

val locks_only_bad_mutexes : CANDIDATE_EXECUTION → SET ACTION

let locks_only_bad_mutexes (Xo, Xw, _) =
  { a | ∀ a ∈ Xo.actions |
    ¬ (locks_only_good_mutex_use Xo.actions Xo.lk Xo.sb Xw.lo a)}

val data_races : CANDIDATE_EXECUTION → SET (ACTION * ACTION)

let data_races (Xo, Xw, ("hb", hb) :: _) =
    { (a, b) | ∀ a ∈ Xo.actions b ∈ Xo.actions |
        ¬ (a = b) ∧ (loc_of a = loc_of b) ∧ (is_write a ∨ is_write b) ∧
        (tid_of a ≠ tid_of b) ∧
        ¬ (is_atomic_action a ∧ is_atomic_action b) ∧
        ¬ ((a, b) ∈ hb ∨ (b, a) ∈ hb) }

let locks_only_undefined_behaviour =
  [ Two ("unsequenced_races", unsequenced_races);
    Two ("data_races", data_races);
    One ("indeterminate_reads", indeterminate_reads);
    One ("locks_only_bad_mutexes", locks_only_bad_mutexes) ]

val locks_only_condition : ∀. CONDITION_T

let locks_only_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    ∀ a ∈ Xo.actions.
      match (loc_of a) with
      | Nothing → false
      | Just l → (Xo.lk l ∈ {Mutex, Non_Atomic})
      end

let locks_only_memory_model =
  ⦃ consistent = locks_only_consistent_execution;
    relation_calculation = locks_only_relations;
    undefined = locks_only_undefined_behaviour;
    relation_flags =
      ⦃ rf_flag = true;
        mo_flag = false;
        sc_flag = false;
        lo_flag = true;
        tot_flag = false ⦄
  ⦄
```

val *locks_only_behaviour* : ∀ . OPSEM_T → PROGRAM → PROGRAM_BEHAVIOURS

let *locks_only_behaviour opsem* (*p* : PROGRAM) =
  behaviour locks_only_memory_model locks_only_condition *opsem p*

C.8 Relaxed-only memory model

val *consistent_atomic_rf* : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let *consistent_atomic_rf* (*Xo*, *Xw*, (*"hb"*, *hb*) :: _) =
  ∀ (*w*, *r*) ∈ *Xw.rf*. is_at_atomic_location *Xo.lk r* ∧ is_load *r* ⟶
    ¬ ((*r*, *w*) ∈ *hb*)

val *rmw_atomicity* : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let *rmw_atomicity* (*Xo*, *Xw*, _) =
  ∀ *b* ∈ *Xo.actions a* ∈ *Xo.actions*.
    is_RMW *b* ⟶ (adjacent_less_than *Xw.mo Xo.actions a b* = ((*a*, *b*) ∈ *Xw.rf*))

val *coherent_memory_use* : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let *coherent_memory_use* (*Xo*, *Xw*, (*"hb"*, *hb*) :: _) =
  `(* CoRR *)`
  (¬ (∃ (*a*, *b*) ∈ *Xw.rf* (*c*, *d*) ∈ *Xw.rf*.
      (*b*, *d*) ∈ *hb* ∧ (*c*, *a*) ∈ *Xw.mo*)) ∧
  `(* CoWR *)`
  (¬ (∃ (*a*, *b*) ∈ *Xw.rf c* ∈ *Xo.actions*.
      (*c*, *b*) ∈ *hb* ∧ (*a*, *c*) ∈ *Xw.mo*)) ∧
  `(* CoRW *)`
  (¬ (∃ (*a*, *b*) ∈ *Xw.rf c* ∈ *Xo.actions*.
      (*b*, *c*) ∈ *hb* ∧ (*c*, *a*) ∈ *Xw.mo*)) ∧
  `(* CoWW *)`
  (¬ (∃ (*a*, *b*) ∈ *hb*. (*b*, *a*) ∈ *Xw.mo*))

val *consistent_mo* : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let *consistent_mo* (*Xo*, *Xw*, _) =
  relation_over *Xo.actions Xw.mo* ∧
  isTransitive *Xw.mo* ∧
  isIrreflexive *Xw.mo* ∧
  ∀ *a* ∈ *Xo.actions b* ∈ *Xo.actions*.
    ((*a*, *b*) ∈ *Xw.mo* ∨ (*b*, *a*) ∈ *Xw.mo*)

```
    = ( (¬ (a = b)) ∧
        is_write a ∧ is_write b ∧
        (loc_of a = loc_of b) ∧
        is_at_atomic_location Xo.lk a )

let relaxed_only_consistent_execution =
  Node [ ("assumptions", Leaf assumptions);
         ("sc_empty", Leaf sc_empty);
         ("tot_empty", Leaf tot_empty);
         ("well_formed_threads", Leaf well_formed_threads);
         ("well_formed_rf", Leaf well_formed_rf);
         ("locks_only_consistent_locks", Leaf locks_only_consistent_locks);
         ("locks_only_consistent_lo", Leaf locks_only_consistent_lo);
         ("consistent_mo", Leaf consistent_mo);
         ("consistent_hb", Leaf consistent_hb);
         ("consistent_rf",
           Node [ ("det_read", Leaf det_read);
                  ("consistent_non_atomic_rf", Leaf consistent_non_atomic_rf);
                  ("consistent_atomic_rf", Leaf consistent_atomic_rf);
                  ("coherent_memory_use", Leaf coherent_memory_use);
                  ("rmw_atomicity", Leaf rmw_atomicity) ]) ]

val relaxed_only_condition : ∀. CONDITION_T

let relaxed_only_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    ∀ a ∈ Xo.actions.
      match a with
      | Lock _ _ _ _ → true
      | Unlock _ _ _ → true
      | Load _ _ mo _ _ → mo ∈ {NA, Relaxed}
      | Store _ _ mo _ _ → mo ∈ {NA, Relaxed}
      | RMW _ _ mo _ _ _ → mo ∈ {Relaxed}
      | Fence _ _ _ → false
      | Blocked_rmw _ _ _ → true
      end

let relaxed_only_memory_model =
  ⦃ consistent = relaxed_only_consistent_execution;
    relation_calculation = locks_only_relations;
    undefined = locks_only_undefined_behaviour;
    relation_flags =
```

```
    ⦃| rf_flag  = true;
       mo_flag  = true;
       sc_flag  = false;
       lo_flag  = true;
       tot_flag = false |⦄
  ⦄
```

val *relaxed_only_behaviour* : ∀. OPSEM_T → PROGRAM → PROGRAM_BEHAVIOURS

let *relaxed_only_behaviour opsem* (*p* : PROGRAM) =
  behaviour relaxed_only_memory_model relaxed_only_condition *opsem p*

C.9 Release-acquire memory model

val *release_acquire_coherent_memory_use* : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let *release_acquire_coherent_memory_use* (*Xo*, *Xw*, (*"hb"*, *hb*) :: _) =
  `(* CoWR *)`
  (¬ (∃ (*a*, *b*) ∈ *Xw.rf* *c* ∈ *Xo.actions*.
        (*c*, *b*) ∈ *hb* ∧ (*a*, *c*) ∈ *Xw.mo*)) ∧
  `(* CoWW *)`
  (¬ (∃ (*a*, *b*) ∈ *hb*. (*b*, *a*) ∈ *Xw.mo*))

val *atomic_initialisation_first* : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let *atomic_initialisation_first* (*Xo*, _, _) =
  ∀ *a* ∈ *Xo.actions* *b* ∈ *Xo.actions*.
    is_at_atomic_location *Xo.lk* *a* ∧ is_NA_store *a* ∧
    is_write *b* ∧ (loc_of *a* = loc_of *b*) ∧ (*a* ≠ *b*) ⟶
    ((*a*, *b*) ∈ transitiveClosure (*Xo.sb* ∪ *Xo.asw*)) ∧ ¬ (is_NA_store *b*)

val *release_acquire_condition* : ∀. CONDITION_T

let *release_acquire_condition* (*Xs* : SET CANDIDATE_EXECUTION) =
  ∀ (*Xo*, *Xw*, *rl*) ∈ *Xs*.
    atomic_initialisation_first (*Xo*, *Xw*, *rl*) ∧
    ∀ *a* ∈ *Xo.actions*.
      match *a* with
      | Lock _ _ _ _ → true
      | Unlock _ _ _ → true
      | Load _ _ *mo* _ _ → (*mo* ∈ {NA, Acquire})
      | Store _ _ *mo* _ _ → (*mo* ∈ {NA, Release})

```
    | RMW _ _ mo _ _ _ → mo = Acq_rel
    | Fence _ _ _ → false
    | Blocked_rmw _ _ _ → true
    end

let release_acquire_synchronizes_with actions sb asw rf lo a b =
  (tid_of a ≠ tid_of b) ∧
  ( (* thread sync *)
    (a, b) ∈ asw ∨
    (* mutex sync *)
    (is_unlock a ∧ is_lock b ∧ (a, b) ∈ lo) ∨
    (* rel/acq sync *)
    ( is_release a ∧ is_acquire b ∧ (a, b) ∈ rf )
  )

let release_acquire_synchronizes_with_set actions sb asw rf lo =
    { (a, b) | ∀ a ∈ actions b ∈ actions |
      release_acquire_synchronizes_with actions sb asw rf lo a b}

let release_acquire_relations Xo Xw =
    let sw = release_acquire_synchronizes_with_set
                Xo.actions Xo.sb Xo.asw Xw.rf Xw.lo in
    let hb = no_consume_hb Xo.sb sw in
    let vse = visible_side_effect_set Xo.actions hb in
    [ ("hb", hb);
      ("vse", vse);
      ("sw", sw) ]

let release_acquire_consistent_execution =
  Node [ ("assumptions", Leaf assumptions);
         ("sc_empty", Leaf sc_empty);
         ("tot_empty", Leaf tot_empty);
         ("well_formed_threads", Leaf well_formed_threads);
         ("well_formed_rf", Leaf well_formed_rf);
         ("locks_only_consistent_locks", Leaf locks_only_consistent_locks);
         ("locks_only_consistent_lo", Leaf locks_only_consistent_lo);
         ("consistent_mo", Leaf consistent_mo);
         ("consistent_hb", Leaf consistent_hb);
         ("consistent_rf",
           Node [ ("det_read", Leaf det_read);
                  ("consistent_non_atomic_rf", Leaf consistent_non_atomic_rf);
                  ("consistent_atomic_rf", Leaf consistent_atomic_rf);
```

```
        ("release_acquire_coherent_memory_use", Leaf release_acquire_coherent_memory_use);
            ("rmw_atomicity", Leaf rmw_atomicity) ]) ]

let release_acquire_memory_model =
  {| consistent = relaxed_only_consistent_execution;
     relation_calculation = release_acquire_relations;
     undefined = locks_only_undefined_behaviour;
     relation_flags =
       {| rf_flag = true;
          mo_flag = true;
          sc_flag = false;
          lo_flag = true;
          tot_flag = false |}
  |}

val release_acquire_behaviour : forall. OPSEM_T -> PROGRAM -> PROGRAM_BEHAVIOURS

let release_acquire_behaviour opsem (p : PROGRAM) =
  behaviour release_acquire_memory_model release_acquire_condition opsem p
```

C.10 Release-acquire-relaxed memory model

```
val release_acquire_relaxed_condition : forall. CONDITION_T

let release_acquire_relaxed_condition (Xs : SET CANDIDATE_EXECUTION) =
  forall (Xo, Xw, rl) ∈ Xs.
    forall a ∈ Xo.actions.
      match a with
      | Lock _ _ _ -> true
      | Unlock _ _ -> true
      | Load _ _ mo _ -> (mo ∈ {NA, Acquire, Relaxed})
      | Store _ _ mo _ -> (mo ∈ {NA, Release, Relaxed})
      | RMW _ _ mo _ _ -> (mo ∈ {Acq_rel, Acquire, Release, Relaxed})
      | Fence _ _ -> false
      | Blocked_rmw _ _ -> true
      end

let release_acquire_relaxed_synchronizes_with actions sb asw rf lo rs a b =
  (tid_of a ≠ tid_of b) ∧
  ( (* thread sync *)
    (a, b) ∈ asw ∨
    (* mutex sync *)
    (is_unlock a ∧ is_lock b ∧ (a, b) ∈ lo) ∨
```

```
    (* rel/acq sync *)
    ( is_release a ∧ is_acquire b ∧
      (∃ c ∈ actions. (a, c) ∈ rs ∧ (c, b) ∈ rf) )
  )

let rs_element head a =
    (tid_of a = tid_of head) ∨ is_RMW a

let release_sequence_set actions lk mo =
  { (rel, b) | ∀ rel ∈ actions b ∈ actions |
    is_release rel ∧
    ( (b = rel) ∨
      ( (rel, b) ∈ mo ∧
        rs_element rel b ∧
        ∀ c ∈ actions.
          ((rel, c) ∈ mo ∧ (c, b) ∈ mo) ⟶ rs_element rel c ) ) }

let release_acquire_relaxed_synchronizes_with_set actions sb asw rf lo rs =
    { (a, b) | ∀ a ∈ actions b ∈ actions |
      release_acquire_relaxed_synchronizes_with actions sb asw rf lo rs a b}

let release_acquire_relaxed_relations Xo Xw =
    let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
  let sw = release_acquire_relaxed_synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf Xw.lo rs in
    let hb = no_consume_hb Xo.sb sw in
    let vse = visible_side_effect_set Xo.actions hb in
    [ ("hb", hb);
      ("vse", vse);
      ("sw", sw);
      ("rs", rs) ]

let release_acquire_relaxed_memory_model =
  ⦃| consistent = relaxed_only_consistent_execution;
     relation_calculation = release_acquire_relaxed_relations;
     undefined = locks_only_undefined_behaviour;
     relation_flags =
       ⦃| rf_flag = true;
          mo_flag = true;
          sc_flag = false;
          lo_flag = true;
          tot_flag = false |⦄
  |⦄
```

val *release_acquire_relaxed_behaviour* : ∀. OPSEM_T → PROGRAM → PROGRAM_BEHAVIOURS

let *release_acquire_relaxed_behaviour opsem* (*p* : PROGRAM) =
behaviour release_acquire_relaxed_memory_model release_acquire_relaxed_condition *opsem p*

C.11 Release-acquire-fenced memory model

val *release_acquire_fenced_condition* : ∀. CONDITION_T

let *release_acquire_fenced_condition* (*Xs* : SET CANDIDATE_EXECUTION) =
∀ (*Xo*, *Xw*, *rl*) ∈ *Xs*.
∀ *a* ∈ *Xo.actions*.
match *a* with
| Lock _ _ _ _ → true
| Unlock _ _ _ → true
| Load _ _ *mo* _ _ → (*mo* ∈ {NA, Acquire, Relaxed})
| Store _ _ *mo* _ _ → (*mo* ∈ {NA, Release, Relaxed})
| RMW _ _ *mo* _ _ _ → (*mo* ∈ {Acq_rel, Acquire, Release, Relaxed})
| Fence _ _ *mo* → (*mo* ∈ {Release, Acquire, Relaxed})
| Blocked_rmw _ _ _ → true
end

let *release_acquire_fenced_synchronizes_with actions sb asw rf lo rs hrs a b* =
(tid_of *a* ≠ tid_of *b*) ∧
(`(* thread sync *)`
(*a*, *b*) ∈ *asw* ∨
`(* mutex sync *)`
(is_unlock *a* ∧ is_lock *b* ∧ (*a*, *b*) ∈ *lo*) ∨
`(* rel/acq sync *)`
(is_release *a* ∧ is_acquire *b* ∧
(∃ *c* ∈ *actions*. (*a*, *c*) ∈ *rs* ∧ (*c*, *b*) ∈ *rf*)) ∨
`(* fence synchronisation *)`
(is_fence *a* ∧ is_release *a* ∧ is_fence *b* ∧ is_acquire *b* ∧
∃ *x* ∈ *actions z* ∈ *actions y* ∈ *actions*.
(*a*, *x*) ∈ *sb* ∧ (*x*, *z*) ∈ *hrs* ∧ (*z*, *y*) ∈ *rf* ∧ (*y*, *b*) ∈ *sb*) ∨
(is_fence *a* ∧ is_release *a* ∧ is_acquire *b* ∧
∃ *x* ∈ *actions y* ∈ *actions*.
(*a*, *x*) ∈ *sb* ∧ (*x*, *y*) ∈ *hrs* ∧ (*y*, *b*) ∈ *rf*) ∨
(is_release *a* ∧ is_fence *b* ∧ is_acquire *b* ∧
∃ *y* ∈ *actions x* ∈ *actions*.

```
      (a, y) ∈ rs ∧ (y, x) ∈ rf ∧ (x, b) ∈ sb) )

let hypothetical_release_sequence_set actions lk mo =
  { (a, b) | ∀ a ∈ actions b ∈ actions |
    is_atomic_action a ∧
    is_write a ∧
    ( (b = a) ∨
      ( (a, b) ∈ mo ∧
        rs_element a b ∧
        ∀ c ∈ actions.
          ((a, c) ∈ mo ∧ (c, b) ∈ mo) ⟶ rs_element a c ) ) }

let release_acquire_fenced_synchronizes_with_set actions sb asw rf lo rs hrs =
    { (a, b) | ∀ a ∈ actions b ∈ actions |
      release_acquire_fenced_synchronizes_with actions sb asw rf lo rs hrs a b}

let release_acquire_fenced_relations Xo Xw =
    let hrs = hypothetical_release_sequence_set Xo.actions Xo.lk Xw.mo in
    let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
  let sw = release_acquire_fenced_synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf Xw.lo rs hrs in
    let hb = no_consume_hb Xo.sb sw in
    let vse = visible_side_effect_set Xo.actions hb in
    [ ("hb", hb);
      ("vse", vse);
      ("sw", sw);
      ("rs", rs);
      ("hrs", hrs) ]

let release_acquire_fenced_memory_model =
  ⦃ consistent = relaxed_only_consistent_execution;
    relation_calculation = release_acquire_fenced_relations;
    undefined = locks_only_undefined_behaviour;
    relation_flags =
      ⦃ rf_flag = true;
        mo_flag = true;
        sc_flag = false;
        lo_flag = true;
        tot_flag = false ⦄
  ⦄

val release_acquire_fenced_behaviour : ∀. OPSEM_T → PROGRAM → PROGRAM_BEHAVIOURS

let release_acquire_fenced_behaviour opsem (p : PROGRAM) =
```

behaviour release_acquire_fenced_memory_model release_acquire_fenced_condition *opsem p*

C.12 SC-accesses memory model

```
val sc_accesses_condition : ∀. CONDITION_T

let sc_accesses_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    ∀ a ∈ Xo.actions.
      match a with
      | Lock _ _ _ _ → true
      | Unlock _ _ _ → true
      | Load _ _ mo _ _ → (mo ∈ {NA, Acquire, Relaxed, Seq_cst})
      | Store _ _ mo _ _ → (mo ∈ {NA, Release, Relaxed, Seq_cst})
      | RMW _ _ mo _ _ _ → (mo ∈ {Acq_rel, Acquire, Release, Relaxed, Seq_cst})
      | Fence _ _ mo → (mo ∈ {Release, Acquire, Relaxed})
      | Blocked_rmw _ _ _ → true
      end

val sc_accesses_consistent_sc : PRE_EXECUTION * EXECUTION_WITNESS *
RELATION_LIST → BOOL

let sc_accesses_consistent_sc (Xo, Xw, ("hb", hb) :: _) =
    relation_over Xo.actions Xw.sc ∧
    isTransitive Xw.sc ∧
    isIrreflexive Xw.sc ∧
    ∀ a ∈ Xo.actions b ∈ Xo.actions.
      ((a, b) ∈ Xw.sc ⟶ ¬ ((b, a) ∈ hb ∪ Xw.mo)) ∧
      ( ((a, b) ∈ Xw.sc ∨ (b, a) ∈ Xw.sc) =
        ( (¬ (a = b)) ∧ is_seq_cst a ∧ is_seq_cst b)
      )

val sc_accesses_sc_reads_restricted : PRE_EXECUTION * EXECUTION_WITNESS *
RELATION_LIST → BOOL

let sc_accesses_sc_reads_restricted (Xo, Xw, ("hb", hb) :: _) =
    ∀ (w, r) ∈ Xw.rf. is_seq_cst r ⟶
      ( is_seq_cst w ∧ (w, r) ∈ Xw.sc ∧
        ¬ (∃ w' ∈ Xo.actions.
          is_write w' ∧ (loc_of w = loc_of w') ∧
          (w, w') ∈ Xw.sc ∧ (w', r) ∈ Xw.sc ) ) ∨
      ( ¬ (is_seq_cst w) ∧
        ¬ (∃ w' ∈ Xo.actions.
```

```
        is_write w' ∧ (loc_of w = loc_of w') ∧
        (w, w') ∈ hb ∧ (w', r) ∈ Xw.sc ) )

let sc_accesses_consistent_execution =
  Node [ ("assumptions", Leaf assumptions);
         ("tot_empty", Leaf tot_empty);
         ("well_formed_threads", Leaf well_formed_threads);
         ("well_formed_rf", Leaf well_formed_rf);
         ("locks_only_consistent_locks", Leaf locks_only_consistent_locks);
         ("locks_only_consistent_lo", Leaf locks_only_consistent_lo);
         ("consistent_mo", Leaf consistent_mo);
         ("sc_accesses_consistent_sc", Leaf sc_accesses_consistent_sc);
         ("consistent_hb", Leaf consistent_hb);
         ("consistent_rf",
          Node [ ("det_read", Leaf det_read);
                 ("consistent_non_atomic_rf", Leaf consistent_non_atomic_rf);
                 ("consistent_atomic_rf", Leaf consistent_atomic_rf);
                 ("coherent_memory_use", Leaf coherent_memory_use);
                 ("rmw_atomicity", Leaf rmw_atomicity);
                 ("sc_accesses_sc_reads_restricted", Leaf sc_accesses_sc_reads_restricted) ]) ]

let sc_accesses_memory_model =
  ⦃| consistent = sc_accesses_consistent_execution;
     relation_calculation = release_acquire_fenced_relations;
     undefined = locks_only_undefined_behaviour;
     relation_flags =
       ⦃| rf_flag = true;
          mo_flag = true;
          sc_flag = true;
          lo_flag = true;
          tot_flag = false |⦄
  |⦄

val sc_accesses_behaviour : ∀. OPSEM_T → PROGRAM → PROGRAM_BEHAVIOURS

let sc_accesses_behaviour opsem (p : PROGRAM) =
  behaviour sc_accesses_memory_model sc_accesses_condition opsem p
```

C.13 SC-fenced memory model

```
val sc_fenced_condition : ∀. CONDITION_T

let sc_fenced_condition (Xs : SET CANDIDATE_EXECUTION) =
```

```
  ∀ (Xo, Xw, rl) ∈ Xs.
    ∀ a ∈ Xo.actions.
      match a with
      | Lock _ _ _ _ → true
      | Unlock _ _ _ → true
      | Load _ _ mo _ _ → (mo ∈ {NA, Acquire, Relaxed, Seq_cst})
      | Store _ _ mo _ _ → (mo ∈ {NA, Release, Relaxed, Seq_cst})
      | RMW _ _ mo _ _ _ → (mo ∈ {Acq_rel, Acquire, Release, Relaxed, Seq_cst})
      | Fence _ _ mo → (mo ∈ {Release, Acquire, Relaxed, Seq_cst})
      | Blocked_rmw _ _ _ → true
      end

val sc_fenced_sc_fences_heeded : PRE_EXECUTION * EXECUTION_WITNESS *
RELATION_LIST → BOOL

let sc_fenced_sc_fences_heeded (Xo, Xw, _) =
    ∀ f ∈ Xo.actions f' ∈ Xo.actions
          r ∈ Xo.actions
          w ∈ Xo.actions w' ∈ Xo.actions.
      ¬ ( is_fence f ∧ is_fence f' ∧
          ( (* fence restriction N3291 29.3p4 *)
            ( (w, w') ∈ Xw.mo ∧
              (w', f) ∈ Xw.sc ∧
              (f, r) ∈ Xo.sb ∧
              (w, r) ∈ Xw.rf ) ∨
            (* fence restriction N3291 29.3p5 *)
            ( (w, w') ∈ Xw.mo ∧
              (w', f) ∈ Xo.sb ∧
              (f, r) ∈ Xw.sc ∧
              (w, r) ∈ Xw.rf ) ∨
            (* fence restriction N3291 29.3p6 *)
            ( (w, w') ∈ Xw.mo ∧
              (w', f) ∈ Xo.sb ∧
              (f, f') ∈ Xw.sc ∧
              (f', r) ∈ Xo.sb ∧
              (w, r) ∈ Xw.rf ) ∨
            (* SC fences impose mo N3291 29.3p7 *)
            ( (w', f) ∈ Xo.sb ∧
              (f, f') ∈ Xw.sc ∧
              (f', w) ∈ Xo.sb ∧
              (w, w') ∈ Xw.mo ) ∨
```

```
          (* N3291 29.3p7, w collapsed first write*)
          ( (w', f) ∈ Xw.sc ∧
            (f, w) ∈ Xo.sb ∧
            (w, w') ∈ Xw.mo ) ∨
          (* N3291 29.3p7, w collapsed second write*)
          ( (w', f) ∈ Xo.sb ∧
            (f, w) ∈ Xw.sc ∧
            (w, w') ∈ Xw.mo ) ) )

let sc_fenced_consistent_execution =
  Node [ ("assumptions", Leaf assumptions);
         ("tot_empty", Leaf tot_empty);
         ("well_formed_threads", Leaf well_formed_threads);
         ("well_formed_rf", Leaf well_formed_rf);
         ("locks_only_consistent_locks", Leaf locks_only_consistent_locks);
         ("locks_only_consistent_lo", Leaf locks_only_consistent_lo);
         ("consistent_mo", Leaf consistent_mo);
         ("sc_accesses_consistent_sc", Leaf sc_accesses_consistent_sc);
         ("sc_fenced_sc_fences_heeded", Leaf sc_fenced_sc_fences_heeded);
         ("consistent_hb", Leaf consistent_hb);
         ("consistent_rf",
           Node [ ("det_read", Leaf det_read);
                  ("consistent_non_atomic_rf", Leaf consistent_non_atomic_rf);
                  ("consistent_atomic_rf", Leaf consistent_atomic_rf);
                  ("coherent_memory_use", Leaf coherent_memory_use);
                  ("rmw_atomicity", Leaf rmw_atomicity);
                  ("sc_accesses_sc_reads_restricted", Leaf sc_accesses_sc_reads_restricted) ]) ]

let sc_fenced_memory_model =
  ⦃| consistent = sc_fenced_consistent_execution;
     relation_calculation = release_acquire_fenced_relations;
     undefined = locks_only_undefined_behaviour;
     relation_flags =
       ⦃| rf_flag = true;
          mo_flag = true;
          sc_flag = true;
          lo_flag = true;
          tot_flag = false |⦄
  |⦄

val sc_fenced_behaviour : ∀. OPSEM_T → PROGRAM → PROGRAM_BEHAVIOURS
```

```
let sc_fenced_behaviour opsem (p : PROGRAM) =
  behaviour sc_fenced_memory_model sc_fenced_condition opsem p
```

C.14 With-consume memory model

```
let with_consume_cad_set actions sb dd rf = transitiveClosure ( (rf ∩ sb) ∪ dd )

let with_consume_dob actions rf rs cad w a =
    tid_of w ≠ tid_of a ∧
    ∃ w' ∈ actions r ∈ actions.
      is_consume r ∧
      (w, w') ∈ rs ∧ (w', r) ∈ rf ∧
      ( (r, a) ∈ cad ∨ (r = a) )

let dependency_ordered_before actions rf rs cad a d =
     a ∈ actions ∧ d ∈ actions ∧
     ( ∃ b ∈ actions. is_release a ∧ is_consume b ∧
       (∃ e ∈ actions. (a, e) ∈ rs ∧ (e, b) ∈ rf) ∧
       ( (b, d) ∈ cad ∨ (b = d) ) )

let with_consume_dob_set actions rf rs cad =
    { (a, b) | ∀ a ∈ actions b ∈ actions |
       dependency_ordered_before actions rf rs cad a b}

let compose R1 R2 =
    { (w, z) | ∀ (w, x) ∈ R1 (y, z) ∈ R2 | (x = y) }

let inter_thread_happens_before actions sb sw dob =
    let r = sw ∪ dob ∪ (compose sw sb) in
      transitiveClosure (r ∪ (compose sb r))

let happens_before actions sb ithb =
    sb ∪ ithb

let with_consume_relations Xo Xw =
    let hrs = hypothetical_release_sequence_set Xo.actions Xo.lk Xw.mo in
    let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
   let sw = release_acquire_fenced_synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf Xw.lo rs hrs in
    let cad = with_consume_cad_set Xo.actions Xo.sb Xo.dd Xw.rf in
    let dob = with_consume_dob_set Xo.actions Xw.rf rs cad in
    let ithb = inter_thread_happens_before Xo.actions Xo.sb sw dob in
    let hb = happens_before Xo.actions Xo.sb ithb in
    let vse = visible_side_effect_set Xo.actions hb in
    [ ("hb", hb);
```

```
    ("vse", vse);
    ("ithb", ithb);
    ("sw", sw);
    ("rs", rs);
    ("hrs", hrs);
    ("dob", dob);
    ("cad", cad) ]

let with_consume_consistent_execution =
  Node [ ("assumptions", Leaf assumptions);
         ("tot_empty", Leaf tot_empty);
         ("well_formed_threads", Leaf well_formed_threads);
         ("well_formed_rf", Leaf well_formed_rf);
         ("locks_only_consistent_locks", Leaf locks_only_consistent_locks);
         ("locks_only_consistent_lo", Leaf locks_only_consistent_lo);
         ("consistent_mo", Leaf consistent_mo);
         ("sc_accesses_consistent_sc", Leaf sc_accesses_consistent_sc);
         ("sc_fenced_sc_fences_heeded", Leaf sc_fenced_sc_fences_heeded);
         ("consistent_hb", Leaf consistent_hb);
         ("consistent_rf",
           Node [ ("det_read", Leaf det_read);
                  ("consistent_non_atomic_rf", Leaf consistent_non_atomic_rf);
                  ("consistent_atomic_rf", Leaf consistent_atomic_rf);
                  ("coherent_memory_use", Leaf coherent_memory_use);
                  ("rmw_atomicity", Leaf rmw_atomicity);
                  ("sc_accesses_sc_reads_restricted", Leaf sc_accesses_sc_reads_restricted) ]) ]

let with_consume_memory_model =
  {| consistent = with_consume_consistent_execution;
     relation_calculation = with_consume_relations;
     undefined = locks_only_undefined_behaviour;
     relation_flags =
       {| rf_flag = true;
          mo_flag = true;
          sc_flag = true;
          lo_flag = true;
          tot_flag = false |}
  |}

val with_consume_behaviour : ∀. OPSEM_T → PROGRAM → PROGRAM_BEHAVIOURS

let with_consume_behaviour opsem (p : PROGRAM) =
```

behaviour with_consume_memory_model true_condition *opsem p*

C.15 Standard memory model

```
let standard_vsses actions lk mo hb vse =
  { (v, r) | ∀ r ∈ actions v ∈ actions head ∈ actions |
    is_at_atomic_location lk r ∧ (head, r) ∈ vse ∧
    ¬ (∃ v' ∈ actions. (v', r) ∈ vse ∧ (head, v') ∈ mo) ∧
    ( v = head ∨
      ( (head, v) ∈ mo ∧ ¬ ((r, v) ∈ hb) ∧
        ∀ w ∈ actions.
          ((head, w) ∈ mo ∧ (w, v) ∈ mo) ⟶ ¬ ((r, w) ∈ hb)
      )
    )
  }

let standard_relations Xo Xw =
    let hrs = hypothetical_release_sequence_set Xo.actions Xo.lk Xw.mo in
    let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
  let sw = release_acquire_fenced_synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf Xw.lo rs hrs in
    let cad = with_consume_cad_set Xo.actions Xo.sb Xo.dd Xw.rf in
    let dob = with_consume_dob_set Xo.actions Xw.rf rs cad in
    let ithb = inter_thread_happens_before Xo.actions Xo.sb sw dob in
    let hb = happens_before Xo.actions Xo.sb ithb in
    let vse = visible_side_effect_set Xo.actions hb in
    let vsses = standard_vsses Xo.actions Xo.lk Xw.mo hb vse in
    [ ("hb", hb);
      ("vse", vse);
      ("ithb", ithb);
      ("vsses", vsses);
      ("sw", sw);
      ("rs", rs);
      ("hrs", hrs);
      ("dob", dob);
      ("cad", cad) ]

val standard_consistent_atomic_rf : PRE_EXECUTION * EXECUTION_WITNESS *
RELATION_LIST → BOOL

let standard_consistent_atomic_rf (Xo, Xw, _ :: _ :: _ :: ("vsses", vsses) :: _) =
    ∀ (w, r) ∈ Xw.rf. is_at_atomic_location Xo.lk r ∧ is_load r⟶
```

```
      (w, r) ∈ vsses

let standard_consistent_execution =
  Node [ ("assumptions", Leaf assumptions);
       ("tot_empty", Leaf tot_empty);
       ("well_formed_threads", Leaf well_formed_threads);
       ("well_formed_rf", Leaf well_formed_rf);
       ("locks_only_consistent_locks", Leaf locks_only_consistent_locks);
       ("locks_only_consistent_lo", Leaf locks_only_consistent_lo);
       ("consistent_mo", Leaf consistent_mo);
       ("sc_accesses_consistent_sc", Leaf sc_accesses_consistent_sc);
       ("sc_fenced_sc_fences_heeded", Leaf sc_fenced_sc_fences_heeded);
       ("consistent_hb", Leaf consistent_hb);
       ("consistent_rf",
        Node [ ("det_read", Leaf det_read);
             ("consistent_non_atomic_rf", Leaf consistent_non_atomic_rf);
             ("standard_consistent_atomic_rf",
              Leaf standard_consistent_atomic_rf);
             ("coherent_memory_use", Leaf coherent_memory_use);
             ("rmw_atomicity", Leaf rmw_atomicity);
             ("sc_accesses_sc_reads_restricted",
              Leaf sc_accesses_sc_reads_restricted) ]) ]

let standard_memory_model =
  ⦃| consistent = standard_consistent_execution;
     relation_calculation = standard_relations;
     undefined = locks_only_undefined_behaviour;
     relation_flags =
       ⦃| rf_flag = true;
          mo_flag = true;
          sc_flag = true;
          lo_flag = true;
          tot_flag = false |⦄
  |⦄

val standard_behaviour : ∀. OPSEM_T → PROGRAM → PROGRAM_BEHAVIOURS

let standard_behaviour opsem (p : PROGRAM) =
  behaviour standard_memory_model true_condition opsem p
```

C.16 Release-acquire-SC memory model

```
val release_acquire_SC_condition : ∀. CONDITION_T
```

```
let release_acquire_SC_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    atomic_initialisation_first (Xo, Xw, rl) ∧
    ∀ a ∈ Xo.actions.
      match a with
      | Lock _ _ _ _ → true
      | Unlock _ _ _ → true
      | Load _ _ mo _ _ → (mo ∈ {NA, Acquire, Seq_cst})
      | Store _ _ mo _ _ → (mo ∈ {NA, Release, Seq_cst})
      | RMW _ _ mo _ _ _ → (mo ∈ {Acq_rel, Seq_cst})
      | Fence _ _ mo → (mo ∈ {Seq_cst})
      | Blocked_rmw _ _ _ → true
      end

let release_acquire_SC_memory_model =
  ⦃ consistent = sc_fenced_consistent_execution;
    relation_calculation = release_acquire_relations;
    undefined = locks_only_undefined_behaviour;
    relation_flags =
      ⦃ rf_flag = true;
        mo_flag = true;
        sc_flag = true;
        lo_flag = true;
        tot_flag = false ⦄
  ⦄

val release_acquire_SC_behaviour : ∀. OPSEM_T → PROGRAM →
PROGRAM_BEHAVIOURS

let release_acquire_SC_behaviour opsem (p : PROGRAM) =
  behaviour release_acquire_SC_memory_model release_acquire_SC_condition opsem p

val release_acquire_SC_rf_behaviour : ∀. OPSEM_T → PROGRAM →
RF_PROGRAM_BEHAVIOURS

let release_acquire_SC_rf_behaviour opsem (p : PROGRAM) =
  rf_behaviour release_acquire_SC_memory_model release_acquire_SC_condition opsem p
```

C.17 SC memory model

```
val SC_condition : ∀. CONDITION_T

let SC_condition (Xs : SET CANDIDATE_EXECUTION) =
```

```
  ∀ (Xo, Xw, rl) ∈ Xs.
    atomic_initialisation_first (Xo, Xw, rl) ∧
    ∀ a ∈ Xo.actions.
      match a with
      | Lock _ _ _ _ → true
      | Unlock _ _ _ → true
      | Load _ _ mo _ _ → (mo ∈ {NA, Seq_cst})
      | Store _ _ mo _ _ → (mo ∈ {NA, Seq_cst})
      | RMW _ _ mo _ _ _ → (mo ∈ {Seq_cst})
      | Fence _ _ mo → false
      | Blocked_rmw _ _ _ → true
      end

let SC_memory_model =
  ⦃ consistent = sc_accesses_consistent_execution;
    relation_calculation = release_acquire_relations;
    undefined = locks_only_undefined_behaviour;
    relation_flags =
      ⦃ rf_flag = true;
        mo_flag = true;
        sc_flag = true;
        lo_flag = true;
        tot_flag = false ⦄
  ⦄

val SC_behaviour : ∀. OPSEM_T → PROGRAM → PROGRAM_BEHAVIOURS

let SC_behaviour opsem (p : PROGRAM) =
  behaviour SC_memory_model SC_condition opsem p
```

C.18 Total memory model

```
val atomic_initialisation_before_all : PRE_EXECUTION * EXECUTION_WITNESS *
RELATION_LIST → BOOL

let atomic_initialisation_before_all (Xo, _, _) =
  ∀ a ∈ Xo.actions b ∈ Xo.actions.
    is_at_atomic_location Xo.lk a ∧ is_NA_store a ∧
    (loc_of a = loc_of b) ∧ (a ≠ b) ⟶
    ((a, b) ∈ transitiveClosure (Xo.sb ∪ Xo.asw)) ∧ ¬ (is_NA_store b)

val bounded_executions : ∀. SET CANDIDATE_EXECUTION → BOOL
```

```
let bounded_executions (Xs : SET CANDIDATE_EXECUTION) =
  ∃ N. ∀ (Xo, Xw, rl) ∈ Xs.
    finite Xo.actions ∧
    size Xo.actions < N

val tot_condition : ∀. CONDITION_T

let tot_condition (Xs : SET CANDIDATE_EXECUTION) =
  bounded_executions Xs ∧
  ∀ (Xo, Xw, rl) ∈ Xs.
    atomic_initialisation_before_all (Xo, Xw, rl) ∧
    ∀ a ∈ Xo.actions.
      match a with
      | Lock _ _ _ _ → true
      | Unlock _ _ _ → true
      | Load _ _ mo _ _ → (mo ∈ {NA, Seq_cst})
      | Store _ _ mo _ _ → (mo ∈ {NA, Seq_cst})
      | RMW _ _ mo _ _ _ → (mo ∈ {Seq_cst})
      | Fence _ _ mo → false
      | Blocked_rmw _ _ _ → true
      end

let tot_relations Xo Xw =
    let vse = visible_side_effect_set Xo.actions Xw.tot in
    [ ("empty", {});
      ("vse", vse);
    ]

val tot_det_read : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let tot_det_read (Xo, Xw, _ :: ("vse", vse) :: _) =
    ∀ r ∈ Xo.actions.
      (∃ w ∈ Xo.actions. (w, r) ∈ vse) =
      (∃ w' ∈ Xo.actions. (w', r) ∈ Xw.rf)

val tot_consistent_rf : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST →
BOOL

let tot_consistent_rf (Xo, Xw, _ :: ("vse", vse) :: _) =
    ∀ (w, r) ∈ Xw.rf. (w, r) ∈ vse

val tot_consistent_locks : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST →
BOOL

let tot_consistent_locks (Xo, Xw, _) =
    (∀ (a, c) ∈ Xw.tot.
```

```
    is_successful_lock a ∧ is_successful_lock c ∧ (loc_of a = loc_of c)
    ⟶
  (∃ b ∈ Xo.actions. (loc_of a = loc_of b) ∧ is_unlock b ∧ (a, b) ∈ Xw.tot ∧ (b, c) ∈ Xw.tot))

val tot_consistent_tot : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let tot_consistent_tot (Xo, Xw, _) =
  relation_over Xo.actions Xw.tot ∧
  isTransitive Xw.tot ∧
  isIrreflexive Xw.tot ∧
  isTrichotomousOn Xw.tot Xo.actions ∧
  Xo.sb ⊆ Xw.tot ∧
  Xo.asw ⊆ Xw.tot ∧
  finite_prefixes Xw.tot Xo.actions

val tot_assumptions : PRE_EXECUTION * EXECUTION_WITNESS * RELATION_LIST → BOOL

let tot_assumptions (Xo, Xw, _) =
 finite_prefixes Xw.rf Xo.actions

let tot_consistent_execution =
 Node [ ("tot_assumptions", Leaf tot_assumptions);
        ("well_formed_threads", Leaf well_formed_threads);
        ("well_formed_rf", Leaf well_formed_rf);
        ("tot_conistent_tot", Leaf tot_consistent_tot);
        ("tot_consistent_locks", Leaf tot_consistent_locks);
        ("consistent_rf",
          Node [ ("det_read", Leaf tot_det_read);
                 ("tot_consistent_rf", Leaf tot_consistent_rf)
               ]
        )
      ]

let tot_bad_mutexes (Xo, Xw, _) =
 { a | ∀ a ∈ Xo.actions |
   let lo = { (a, b) | ∀ a ∈ Xo.actions b ∈ Xo.actions |
              ((a, b) ∈ Xw.tot) ∧ (loc_of a = loc_of b) ∧
              is_at_mutex_location Xo.lk a
            } in
   ¬ (locks_only_good_mutex_use Xo.actions Xo.lk Xo.sb lo a)}

let tot_data_races (Xo, Xw, _) =
```

```
    { (a, b) | ∀ a ∈ Xo.actions b ∈ Xo.actions |
        ¬ (a = b) ∧ (loc_of a = loc_of b) ∧ (is_write a ∨ is_write b) ∧
        (tid_of a ≠ tid_of b) ∧
        ¬ (is_atomic_action a ∧ is_atomic_action b) ∧
        ¬ ((a, b) ∈ Xo.asw) ∧
        (a, b) ∈ Xw.tot ∧
        ¬ (∃ c ∈ Xo.actions. ((a, c) ∈ Xw.tot) ∧ ((c, b) ∈ Xw.tot)) }

let tot_undefined_behaviour =
  [ Two ("unsequenced_races", unsequenced_races);
    Two ("data_races", tot_data_races);
    One ("indeterminate_reads", indeterminate_reads);
    One ("tot_bad_mutexes", tot_bad_mutexes) ]

let tot_memory_model =
  ⦃ consistent = tot_consistent_execution;
    relation_calculation = tot_relations;
    undefined = tot_undefined_behaviour;
    relation_flags =
      ⦃ rf_flag = true;
        mo_flag = false;
        sc_flag = false;
        lo_flag = false;
        tot_flag = true; ⦄
  ⦄

val tot_behaviour : ∀. OPSEM_T → PROGRAM → PROGRAM_BEHAVIOURS

let tot_behaviour opsem (p : PROGRAM) =
  behaviour tot_memory_model tot_condition opsem p

val tot_rf_behaviour : ∀. OPSEM_T → PROGRAM → RF_PROGRAM_BEHAVIOURS

let tot_rf_behaviour opsem (p : PROGRAM) =
  rf_behaviour tot_memory_model tot_condition opsem p
```

C.19 Theorems

```
val cond : ∀. (PROGRAM → PRE_EXECUTION → BOOL) → PROGRAM → BOOL

theorem {hol, isabelle, tex} thm_0 :
(∀ opsem p.
  (behaviour with_consume_memory_model true_condition opsem p =
   behaviour standard_memory_model true_condition opsem p))
```

theorem $\{hol,\ isabelle,\ tex\}\ thm_1$:
$(\forall\ opsem\ p.$
statically_satisfied sc_fenced_condition $opsem\ p \longrightarrow$
(behaviour sc_fenced_memory_model sc_fenced_condition $opsem\ p =$
behaviour with_consume_memory_model true_condition $opsem\ p$))

theorem $\{hol,\ isabelle,\ tex\}\ thm_2$:
$(\forall\ opsem\ p.$
statically_satisfied sc_accesses_condition $opsem\ p \longrightarrow$
(behaviour sc_accesses_memory_model sc_accesses_condition $opsem\ p =$
behaviour sc_fenced_memory_model sc_fenced_condition $opsem\ p$))

theorem $\{hol,\ isabelle,\ tex\}\ thm_3$:
$(\forall\ opsem\ p.$
statically_satisfied release_acquire_fenced_condition $opsem\ p \longrightarrow$
(behaviour release_acquire_fenced_memory_model release_acquire_fenced_condition $opsem\ p =$
behaviour sc_accesses_memory_model sc_accesses_condition $opsem\ p$))

theorem $\{hol,\ isabelle,\ tex\}\ thm_4$:
$(\forall\ opsem\ p.$
statically_satisfied release_acquire_relaxed_condition $opsem\ p \longrightarrow$
(behaviour release_acquire_relaxed_memory_model release_acquire_relaxed_condition $opsem\ p =$
behaviour release_acquire_fenced_memory_model release_acquire_fenced_condition $opsem\ p$))

theorem $\{hol,\ isabelle,\ tex\}\ thm_6$:
$(\forall\ opsem\ p.$
statically_satisfied relaxed_only_condition $opsem\ p \longrightarrow$
(behaviour relaxed_only_memory_model relaxed_only_condition $opsem\ p =$
behaviour release_acquire_relaxed_memory_model release_acquire_relaxed_condition $opsem\ p$))

theorem $\{hol,\ isabelle,\ tex\}\ thm_7$:
$(\forall\ opsem\ p.$
statically_satisfied locks_only_condition $opsem\ p \longrightarrow$
(behaviour locks_only_memory_model locks_only_condition $opsem\ p =$
behaviour release_acquire_memory_model release_acquire_condition $opsem\ p$))

theorem $\{hol,\ isabelle,\ tex\}\ thm_8$:
$(\forall\ opsem\ p.$
statically_satisfied locks_only_condition $opsem\ p \longrightarrow$
(behaviour locks_only_memory_model locks_only_condition $opsem\ p =$
behaviour relaxed_only_memory_model relaxed_only_condition $opsem\ p$))

theorem $\{hol,\ isabelle,\ tex\}\ thm_9$:
$(\forall\ opsem\ p.$

statically_satisfied single_thread_condition *opsem p* ⟶
(behaviour single_thread_memory_model single_thread_condition *opsem p* =
behaviour locks_only_memory_model locks_only_condition *opsem p*))

theorem {*hol*, *isabelle*, *tex*} thm_{10} :
(∀ *opsem p*.
statically_satisfied release_acquire_SC_condition *opsem p* ⟶
(behaviour sc_fenced_memory_model sc_fenced_condition *opsem p* =
behaviour release_acquire_SC_memory_model release_acquire_SC_condition *opsem p*))

theorem {*hol*, *isabelle*, *tex*} thm_5 :
(∀ *opsem p*.
statically_satisfied release_acquire_condition *opsem p* ⟶
(behaviour release_acquire_memory_model release_acquire_condition *opsem p* =
behaviour release_acquire_SC_memory_model release_acquire_SC_condition *opsem p*))

theorem {*hol*, *isabelle*, *tex*} thm_{11} :
(∀ *opsem p*.
statically_satisfied SC_condition *opsem p* ⟶
(behaviour SC_memory_model SC_condition *opsem p* =
behaviour release_acquire_SC_memory_model release_acquire_SC_condition *opsem p*))

theorem {*hol*, *isabelle*, *tex*} thm_{12} :
(∀ *opsem p*.
statically_satisfied locks_only_condition *opsem p* ⟶
(behaviour SC_memory_model SC_condition *opsem p* =
behaviour locks_only_memory_model locks_only_condition *opsem p*))

theorem {*hol*, *isabelle*, *tex*} *bigthm* :
(∀ *opsem p*.
opsem_assumptions *opsem* ∧
statically_satisfied tot_condition *opsem p* ⟶
(rf_behaviour SC_memory_model SC_condition *opsem p* =
rf_behaviour tot_memory_model tot_condition *opsem p*))

theorem {*hol*, *isabelle*, *tex*} thm_{14} :
(∀ *opsem p*.
statically_satisfied SC_condition *opsem p* ⟶
(behaviour SC_memory_model SC_condition *opsem p* =
behaviour sc_accesses_memory_model sc_accesses_condition *opsem p*))

theorem {*hol*, *isabelle*, *tex*} thm_{15} :
(∀ *opsem p*.
statically_satisfied release_acquire_condition *opsem p* ⟶

```
  (behaviour release_acquire_memory_model release_acquire_condition opsem p =
  behaviour release_acquire_relaxed_memory_model release_acquire_relaxed_condition opsem p))

theorem {hol, isabelle, tex} thm_16 :
(∀ opsem p.
  opsem_assumptions opsem ∧
  statically_satisfied tot_condition opsem p ∧
  statically_satisfied locks_only_condition opsem p
  ⟶
  (rf_behaviour locks_only_memory_model locks_only_condition opsem p =
   rf_behaviour tot_memory_model tot_condition opsem p))

val release_acquire_no_locks_condition : ∀. CONDITION_T

let release_acquire_no_locks_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    ∀ a ∈ Xo.actions.
      match a with
      | Lock _ _ _ _ → false
      | Unlock _ _ _ → false
      | Load _ _ mo _ _ → (mo ∈ {NA, Acquire})
      | Store _ _ mo _ _ → (mo ∈ {NA, Release})
      | RMW _ _ mo _ _ _ → mo = Acq_rel
      | Fence _ _ _ → false
      | Blocked_rmw _ _ _ → true
      end

let release_acquire_no_locks_synchronizes_with actions sb asw rf a b =
  (tid_of a ≠ tid_of b) ∧
  ( (* thread sync *)
    (a, b) ∈ asw ∨
    (* rel/acq sync *)
    ( is_release a ∧ is_acquire b ∧ (a, b) ∈ rf )
  )

let release_acquire_no_locks_synchronizes_with_set actions sb asw rf =
    { (a, b) | ∀ a ∈ actions b ∈ actions |
       release_acquire_no_locks_synchronizes_with actions sb asw rf a b}

let release_acquire_no_locks_relations Xo Xw =
    let sw = release_acquire_no_locks_synchronizes_with_set
               Xo.actions Xo.sb Xo.asw Xw.rf in
    let hb = no_consume_hb Xo.sb sw in
```

```
    let vse = visible_side_effect_set Xo.actions hb in
    [ ("hb", hb);
      ("vse", vse);
      ("sw", sw) ]

let sc_lo_empty (_, Xw, _) = null Xw.sc ∧ null Xw.lo

let release_acquire_no_locks_consistent_execution =
  Node [ ("assumptions", Leaf assumptions);
         ("sc_lo_empty", Leaf sc_empty);
         ("tot_empty", Leaf tot_empty);
         ("well_formed_threads", Leaf well_formed_threads);
         ("well_formed_rf", Leaf well_formed_rf);
         ("consistent_mo", Leaf consistent_mo);
         ("consistent_hb", Leaf consistent_hb);
         ("consistent_rf",
           Node [ ("det_read", Leaf det_read);
                  ("consistent_non_atomic_rf", Leaf consistent_non_atomic_rf);
                  ("consistent_atomic_rf", Leaf consistent_atomic_rf);
                  ("coherent_memory_use", Leaf coherent_memory_use);
                  ("rmw_atomicity", Leaf rmw_atomicity) ]) ]

let release_acquire_no_locks_undefined_behaviour =
  [ Two ("unsequenced_races", unsequenced_races);
    Two ("data_races", data_races);
    One ("indeterminate_reads", indeterminate_reads); ]

let release_acquire_no_locks_memory_model =
  ⦃ consistent = release_acquire_no_locks_consistent_execution;
    relation_calculation = release_acquire_no_locks_relations;
    undefined = release_acquire_no_locks_undefined_behaviour;
    relation_flags =
      ⦃ rf_flag = true;
        mo_flag = true;
        sc_flag = false;
        lo_flag = true;
        tot_flag = false ⦄
  ⦄

val release_acquire_no_locks_behaviour : ∀. OPSEM_T → PROGRAM →
PROGRAM_BEHAVIOURS

let release_acquire_no_locks_behaviour opsem (p : PROGRAM) =
 behaviour release_acquire_no_locks_memory_model release_acquire_no_locks_condition opsem p
```

```
val release_acquire_lifetime_no_locks_condition : ∀. CONDITION_T

let release_acquire_lifetime_no_locks_condition (Xs : SET CANDIDATE_EXECUTION) =
  ∀ (Xo, Xw, rl) ∈ Xs.
    ∀ a ∈ Xo.actions.
      match a with
      | Lock _ _ _ _ → false
      | Unlock _ _ _ → false
      | Load _ _ mo _ _ → (mo ∈ {NA, Acquire})
      | Store _ _ mo _ _ → (mo ∈ {NA, Release})
      | RMW _ _ mo _ _ _ → mo = Acq_rel
      | Fence _ _ _ → false
      | Blocked_rmw _ _ _ → true
      end
```

Bibliography

[1] Defect report summary for C11. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/summary.htm`.

[2] Thesis web page. `http://www.cl.cam.ac.uk/~mjb220/thesis`.

[3] Book reviews. *IEEE Concurrency*, 2(3):82–85, 1994.

[4] *Programming Languages — C++*. 1998. SO/IEC 14882:1998.

[5] *Programming Languages — C++*. 2003. ISO/IEC 14882:2003.

[6] *Programming Languages — C++*. 2007. ISO/IEC TR 19768:2007.

[7] *Power ISA Version 2.06*. IBM, 2009.

[8] *Programming Languages — C*. 2011. ISO/IEC 9899:2011. A non-final but recent version is available at `http://www.open-std.org/jtc1/sc22/wg14/docs/n1539.pdf`.

[9] I. I. 14882:2003. C++ standard core language active issues, revision 87. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3833.html#1466`. WG21 N3833.

[10] S. V. Adve. *Designing Memory Consistency Models for Shared-memory Multiprocessors*. PhD thesis, Madison, WI, USA, 1993. UMI Order No. GAX94-07354.

[11] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, Dec. 1996.

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

[13] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proc. SPAA*, 1993.

[14] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *DAMP '09: Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 13–24, New York, NY, USA, 2008. ACM.

[15] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. *arXiv preprint arXiv:1312.1411*, 2013.

[16] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proceedings of the 22nd International Conference on Computer Aided Verification*, CAV'10, pages 258–272, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*, TACAS'11/ETAPS'11, pages 41–44, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats. *arXiv preprint arXiv:1308.6810*, 2013.

[19] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 487–498, New York, NY, USA, 2011. ACM.

[20] M. Batty. Comments on the C++ memory model following a partial formalization attempt. `http://www.open-std.org/JTC1/sc22/WG21/docs/papers/2009/n2955.html`. N2955=09-0145.

[21] M. Batty. Defect Report 407. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_407.htm`.

[22] M. Batty. "happens before" can not be cyclic. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_401.htm`.

[23] M. Batty. Missing ordering constraints. `http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-defects.html#2130`. I2130.

[24] M. Batty. Visible sequences of side effects are redundant. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_406.htm`.

[25] M. Batty, M. Dodds, and A. 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.

[26] 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.

[27] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency: The post-Rapperswil model. http://www.cl.cam.ac.uk/~pes20/cpp/tech.pdf. A revision of N3132.

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

[29] M. Batty and P. Sewell. Notes on the draft C++ memory model. http://www.cl.cam.ac.uk/~mjb220/cpp/cpp2440.pdf.

[30] P. Becker, editor. *Programming Languages — C++.* 2011. ISO/IEC 14882:2011. A non-final but recent version is available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.

[31] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Proc. ITP*, 2010.

[32] J. C. Blanchette, T. Weber, M. Batty, S. Owens, and S. Sarkar. Nitpicking C++ concurrency. In *Proc. PPDP*, 2011.

[33] H. Boehm. The atomic_ops project. http://github.com/ivmai/libatomic_ops.

[34] H. Boehm and L. Crowl. C++ atomic types and operations. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2393.html. N2381 = 07-0241.

[35] H.-J. Boehm. A memory model for C++: Strawman proposal. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1942.html. WG21/N1942.

[36] H.-J. Boehm. Threads cannot be implemented as a library. In *Proc. PLDI*, 2005.

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

[38] H.-J. Boehm, D. Lea, and B. Pugh. Memory model for multithreaded C++: August 2005 status update. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1876.pdf. WG21/N1876.

[39] www.cl.cam.ac.uk/users/pes20/cpp.

[40] S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In *Programming Languages and Systems*, pages 87–107. Springer, 2012.

[41] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *Proceedings of the 16th European Conference on Programming*, ESOP'07, pages 331–346, Berlin, Heidelberg, 2007. Springer-Verlag.

[42] L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, 1978.

[43] W. W. Collier. *Reasoning about parallel architectures.* Prentice Hall, 1992.

[44] I. Corporation. *IBM Enterprise Systems Architecture/370: Principles of Operation.* 1988. Publication number SA22-7200-0.

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

[46] D. Demange, V. Laporte, L. Zhao, S. Jagannathan, D. Pichardie, and J. Vitek. Plan B: A buffered memory model for Java. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 329–342, New York, NY, USA, 2013. ACM.

[47] E. W. Dijkstra. The origin of concurrent programming. chapter Cooperating Sequential Processes, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[48] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. *SIGARCH Comput. Archit. News*, 14(2):434–442, May 1986.

[49] C. M. Ellison. *A formal semantics of C with applications.* PhD thesis, University of Illinois, 2012.

[50] A. Gotsman, M. Musuvathi, and H. Yang. Show no weakness: sequentially consistent specifications of TSO libraries. In *Distributed Computing*, pages 31–45. Springer, 2012.

[51] B. Hedquist. ISO/IEC FCD 14882, C++0X, national body comments. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3102.pdf`. N3102=10-0092.

[52] M. Huisman and G. Petri. The Java memory model: a formal explanation. In *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, 2007.

[53] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6.* pub-IEEE-STD, pub-IEEE-STD:adr, 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992, Open Group Technical Standard Base Specifications, Issue 6.

[54] Intel. A formal specification of Intel Itanium processor family memory ordering. `http://www.intel.com/design/itanium/downloads/251429.htm`, Oct. 2002.

[55] ISO. C++ standards committee papers. `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/`.

[56] R. Jagadeesan, G. Petri, C. Pitcher, and J. Riely. Quarantining weakness. In *Programming Languages and Systems*, pages 492–511. Springer, 2013.

[57] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

[58] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, USA, Oct. 2009. ACM.

[59] R. Krebbers. Aliasing restrictions of C11 formalized in Coq. In *Certified Programs and Proofs*, pages 50–65. Springer, 2013.

[60] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[61] N. M. Lê, A. Guatto, A. Cohen, and A. Pop. Correct and efficient bounded FIFO queues. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2013 25th International Symposium on*, pages 144–151. IEEE, 2013.

[62] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 69–80, New York, NY, USA, 2013. ACM.

[63] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[64] G. Li, M. Delisi, G. Gopalakrishnan, and R. M. Kirby. Formal specification of the mpi-2.0 standard in tla+. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 283–284, New York, NY, USA, 2008. ACM.

[65] G. Li, R. Palmer, M. Delisi, G. Gopalakrishnan, and R. M. Kirby. Formal specification of mpi 2.0: Case study in specifying a practical concurrent programming api. *Sci. Comput. Program.*, 76(2):65–81, 2011.

[66] A. Lochbihler. Java and the Java memory model—a unified, machine-checked formalisation. In *Programming Languages and Systems*, pages 497–517. Springer, 2012.

[67] A. Lochbihler. *A Machine-Checked, Type-Safe Model of Java Concurrency: Language, Virtual Machine, Memory Model, and Verified Compiler*. KIT Scientific Publishing, 2012.

[68] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *CAV*, 2012.

[69] J. Manson. *The Java Memory Model*. PhD thesis, University of Maryland, 2004.

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

[71] L. Maranget, S. Sarkar, and P. Sewell. A tutorial introduction to the ARM and POWER relaxed memory models.

[72] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A case for an SC-preserving compiler. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 199–210, New York, NY, USA, 2011. ACM.

[73] P. E. McKenney, M. Batty, C. Nelson, H. Boehm, A. Williams, S. Owens, S. Sarkar, P. Sewell, T. Weber, M. Wong, and L. Crowl. Omnibus memory model and atomics paper. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3125.html`. N3125=10-0115.

[74] P. E. McKenney, M. Batty, C. Nelson, H. Boehm, A. Williams, S. Owens, S. Sarkar, P. Sewell, T. Weber, M. Wong, L. Crowl, and B. Kosnik. Omnibus memory model and atomics paper. ISO/IEC JTC1 SC22 WG21 N3196 = 10-0186 - 2010-11-11 `http://open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3196.htm`.

[75] P. E. McKenney, M. Batty, C. Nelson, N. Maclaren, H. Boehm, A. Williams, P. Dimov, and L. Crowl. Explicit initializers for atomics. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3057.html`. N3057=10-0047.

[76] P. E. McKenney, M. Batty, C. Nelson, N. Maclaren, H. Boehm, A. Williams, P. Dimov, and L. Crowl. Updates to C++ memory model based on formalization. `http://open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3045.html`, 2010. ISO/IEC JTC1 SC22 WG21 Working Paper N3045 = 10-0035.

[77] P. E. McKenney, M. Batty, C. Nelson, N. Maclaren, H. Boehm, A. Williams, P. Dimov, and L. Crowl. Updates to C++ memory model based on formalization. , 2010. ISO/IEC JTC1 SC22 WG21 Working Paper N3074 = 10-0064.

[78] P. E. McKenney and R. Silvera. Example POWER implementation for C/C++ memory model. `http://www.rdrop.com/users/paulmck/scalability/paper/N2745r.2010.02.19a.html`, 2010.

[79] P. E. McKenney and J. Walpole. What is RCU, fundamentally? Linux Weekly News, `http://lwn.net/Articles/262464/`.

[80] A. Meredith. C++ standard library defect report list (revision R85). `http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-defects.html`. n3822.

[81] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15:491–504, June 2004.

[82] W. M. Miller. Additional core language issue resolutions for Issaquah. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3914.html`. WG21 N3833.

[83] W. M. Miller. C++ standard core language active issues, revision 86. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3806.html`. n3086.

[84] R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 187–196, New York, NY, USA, 2013. ACM.

[85] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. In *ICFP '14: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ACM Press, Sept. 2014.

[86] A. Munshi, editor. *The OpenCL Specification*. 2013. A non-final but recent version is available at `http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`.

[87] B. Norris and B. Demsky. CDSchecker: Checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 131–150, New York, NY, USA, 2013. ACM.

[88] M. Norrish. Deterministic expressions in C. In S. D. Swierstra, editor, *Programming Languages and Systems: 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 147–161. Spring, March 1999.

[89] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *Proc. ECOOP*, 2010.

[90] S. Owens, P. Böhm, F. Zappa Nardelli, and P. Sewell. Lem: A lightweight tool for heavyweight semantics. In *Proc. ITP, LNCS 6898*, pages 363–369, 2011. "Rough Diamond" section.

[91] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proc. TPHOLs*, 2009.

[92] R. Palmer, M. DeLisi, G. Gopalakrishnan, and R. Kirby. An approach to formal-

ization and analysis of message passing libraries. In S. Leue and P. Merino, editors, *Formal Methods for Industrial Critical Systems*, volume 4916 of *Lecture Notes in Computer Science*, pages 164–181. Springer Berlin Heidelberg, 2008.

[93] B. Rajaram, V. Nagarajan, S. Sarkar, and M. Elver. Fast RMWs for TSO: Semantics and implementation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 61–72, New York, NY, USA, 2013. ACM.

[94] T. Ramananandro. *Les objets en C++: sémantique formelle mécanisée et compilation vérifiée*. PhD thesis, Université Paris-Diderot-Paris VII, 2012.

[95] T. Ramananandro, G. Dos Reis, and X. Leroy. Formal verification of object layout for C++ multiple inheritance. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 67–80, New York, NY, USA, 2011. ACM.

[96] 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.

[97] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 175–186, New York, NY, USA, 2011. ACM.

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

[99] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. POPL*, 2009.

[100] D. Schwartz-Narbonne. *Assertions for debugging parallel programs*. PhD thesis, Princeton University, 2013.

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

[102] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *Proc. POPL*, 2011.

[103] 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.

[104] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *C. ACM*, 53(7):89–97, 2010.

[105] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *TOPLAS*, 10:282–312, 1988.

[106] A. Terekhov. Brief example ARM implementation for C/C++ memory model. `cpp-threads` mailing list, `http://www.decadent.org.uk/pipermail/cpp-threads/2008-December/001950.html`, Dec. 2008.

[107] A. Terekhov. Brief tentative example x86 implementation for C/C++ memory model. `cpp-threads` mailing list, `http://www.decadent.org.uk/pipermail/cpp-threads/2008-December/001933.html`, Dec. 2008.

[108] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[109] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. 2014. under submission.

[110] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages' Applications*, OOPSLA 13, pages 867–884, New York, NY, USA, 2013. ACM.

[111] M. Wong, B. Kosnik, and M. Batty. Coherence requirements detailed. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3136.html`. N3136 = 10-0126.

Index

Last updated: Saturday 29th November, 2014