PHILOSOPHICAL TRANSACTIONS A

rsta.royalsocietypublishing.org

Research

Article submitted to journal

Subject Areas:
Computer Science

Keywords:
Relaxed memory, verification, semantics, concurrency, heterogeneous systems, GPU

Author for correspondence:
Mark Batty
e-mail: mbatty@cantab.net

Compositional relaxed concurrency

Mark Batty

University of Kent

There is a broad design space for concurrent computer processors: they can be optimised for low power, low latency or high throughput. This freedom to tune each processor design to its niche has led to an increasing diversity of machines, from powerful pocketable devices, to those responsible for complex and critical tasks, such as car guidance systems.

Given this context, academic concurrency research sounds notes of both caution and optimism. Caution because recent work has uncovered flaws in the way we explain the subtle memory behaviour of concurrent systems: specifications have been shown to be incorrect, leading to bugs throughout the many layers of the system. And optimism because our tools and methods for verifying the correctness of concurrent code – although built above an idealised model of concurrency – are becoming more mature.

This paper looks at the way we specify the memory behaviour of concurrent systems and suggests a new direction. Currently there is a siloed approach, with each processor and programming language specified separately in an incomparable way. But this does not match the structure of our programs, which may use multiple processors and languages together. Instead we propose a *compositional* approach, where program components carry with them a description of the sort of concurrency they rely on, and there is a mechanism for composing these. This will not only support components written for the multiple varied processors found in a modern system, but also those that use idealised models of concurrency, providing a sound footing for mature verification techniques.

The advent of concurrency in computing has provided an extraordinary level of performance, allowing computer systems to pervade modern life, from personal computers and mobile phones, to critical-infrastructure control systems and even safety-critical automotive applications: in every sense, our lives depend on these machines.

For each niche that a particular computer occupies, we incorporate a processor whose concurrency scheme is tuned to that niche: typical CPUs have few cores, each optimised for low latency, whereas mobile-phone processors use a concurrency scheme tuned to low power usage, and graphics processors (GPUs) feature a vast array of small cores, providing high throughput. One element is common to the concurrency schemes of all mainstream processors: the cores of the processor communicate with one another by writing to and reading from a *shared memory*. Processor vendors avoid the cost of fully hiding microarchitectural details such as memory caching, buffering and value speculation by permitting unintuitive program executions that violate sequential consistency (SC), a model for the behaviour of memory where concurrent accesses are simply interleaved [23]. We say that these non-SC systems exhibit *relaxed* memory behaviour.

Despite their ubiquity, relaxed memory systems are extremely subtle and poorly understood. The behaviour of a given processor is typically delineated by an English-prose specification document, published by its vendor. Unfortunately, English prose is prone to ambiguity, omission and error, leading to major bugs in programming language specifications [11,13], deployed processors [3], compilers [25,32,37], and vendor-endorsed programming idioms [1] – it is clear that current engineering practice is severely lacking. At the same time, without a precise definition of the underlying concurrency scheme, rapidly maturing software verification techniques rely on the idealised assumption of sequential consistency, limiting their applicability.

There has been a strong response to this problem from academia, providing mechanised formal models that unambiguously specify relaxed memory behaviour, together with tools that execute the models, and proofs that validate key design goals. Many of these models were developed in close collaboration with the system designers [2,4,6,13,24,31,37] and some have even led to alterations in industrial specifications [13,37]. Recent work provides formal models of relaxed CPU concurrency [2,4,7,24,26,31], relaxed GPU concurrency [1,37], and the concurrency schemes of various programming languages [10,13,37]. The techniques used to develop, refine and validate these models are maturing, but most models are still in flux, with current work tackling fundamental problems in relaxed programming language concurrency [11,19,20,27,28], and separate work developing more accurate and complete processor models [16,17].

These memory models are limited in their coverage: they eschew as much detail as possible, narrowly focusing on the memory behaviour of a monolithic program written for a single processor architecture or programming language. But real programs are not monoliths, and nor are they written for a lone processor or language. Instead, they are an aggregate of rather different components, each written for one or another of the system's processors, or within one of a host of programming languages, each with its own concurrency rules. Moreover, each component is the creation of a different team of people, and is refined in isolation of the rest of the aggregate, both manually by its developer, but also automatically by optimising compilers. The disparate components are cemented together, aided by a set of conventions that govern how they ought to be combined, but each component is primarily understood in isolation. It is at the level of the component that we write our code and attempt to verify its correctness, but our existing models of concurrency, which can only judge the behaviour of a whole program, do not fit with this reality.

This document will suggest a new form for relaxed memory models. One that seeks to accommodate the view of a concurrent program as an aggregate of varied components. This model will ultimately form the basis of a reasoning principle which will enable verification above the varied collection of relaxed CPUs and GPUs in a modern system. The same approach will enable the separate verification of components using well-developed SC reasoning principles.

1. Relaxed concurrency

All mainstream processors use a shared memory for communication between cores, allowing subtle and unintuitive relaxed behaviours in order to admit performance-enhancing optimisations. Programming languages that compile down to these processors also allow relaxed behaviour to permit efficient compilation, but in addition they perform intricate compiler optimisations that introduce yet more relaxed behaviour. This section will give a flavour of the concurrency behaviour allowed by SC, a selection of mainstream processors, and the C++ language.

(a) Sequential consistency.

Advanced verification techniques typically adopt sequential consistency [23], an idealised model of concurrency, free from the odd behaviours induced by optimisations. Here, the memory accesses of each thread are interleaved, respecting the order that they appear in the thread, and they act on a single shared memory with each read of a variable taking the value of the previous write of that variable.

We will illustrate this model with a simple example program, called *store buffering*, listed below. It comprises two threads executing in parallel, with one drawn above the other, separated by a horizontal bar. We assume all variables start with value 0. The first thread stores `1` to `x` and then reads from `y`, storing the result in `r1`, and the second stores `1` to `y` and then reads from `x`, storing the result in `r2`. Here, `x` and `y` are variables in memory, and we highlight accesses of them by using calls to `load` and `store`. In contrast, `r1` and `r2` represent thread-local variables, and we do not consider their effect on memory.

Store buffering (SB)

```
store(x,1)
r1=load(y)
----------
store(y,1)
r2=load(x)
```

Under sequential consistency, what outcomes are allowed? Variables `x` and `y` start with value 0 and there are only writes of value 1, so there are four possibilities to consider: 1/1, 1/0, 0/1 and 0/0. We can construct interleavings of the accesses that are consistent with the first three outcomes. For example, the outcome 1/1 is produced by the interleaving: `store(x,1)`, `store(y,1)`, `r1=load(y)`, `r2=load(x)`. Note that there is no interleaving – consistent with the order of accesses in the program – that matches the outcome 0/0, so that behaviour is forbidden by sequential consistency.

(b) Processor concurrency models

Processors exhibit more complex memory behaviour because they feature optimisations over memory that are not hidden from the programmer.

x86 processors. Academic work on a formal model of the concurrency behaviour of x86 processors highlighted deficiencies in the processor architecture documentation [26,33]. Intel released further documents that fixed the problems, bringing their descriptions in line with the formal model. According to this model, each thread of an x86 processor has a buffer attached, and when a thread stores a value, that store enters the buffer before being flushed to memory at any later point. When a thread loads a value, it checks its local buffer first, and if there is no store of the variable sought, only then does it load directly from memory.

Returning to the SB example above, note that any SC outcome can be witnessed in the x86 model by flushing stores immediately following buffering. The model is more relaxed than SC however, because the outcome 0/0 is allowed by the following sequence of steps: buffer `store(x,1)` on the first thread, buffer `store(y,1)` on the second, `load(y)` from memory, `load(x)` from memory.

The store buffering test is a minimal example that illustrates a relaxed behaviour. We call such examples *litmus tests*, and in future tests, we will use comments in the code to indicate the outcome that corresponds to the relaxed behaviour (e.g. 0/0 here in SB).

Power and ARM processors. The Power and ARM processors make many more optimisations visible to programmers, and as a result exhibit far more relaxed behaviour. Their models are similar and are too intricate to describe here, but one can get a flavour of them by considering the *message-passing* litmus tests below.

Message passing (MP)

```
store(x,1)
store(y,1)
--------------------------
r1=load(y) //reads 1
r2=load(x) //reads 0
```

MP+sync+ctrl

```
store(x,1)
sync
store(y,1)
--------------------------
r1=load(y)
if(r1 == 1)
   {r2=load(x)}
```

MP captures the following intuition: the first thread writes some data (represented by the write to x), with the expectation that this may take some time. On completion, the same thread writes to a flag variable y. The second thread reads the flag y. If it is set, then the data in x should be ready to read, so the load of x should get value 1. If the indicated outcome 1/0 is permitted, then this so-called message passing idiom breaks down: the second thread can read the data before it is ready, despite being passed a message indicating readiness through the flag y. To see that the outcome 1/0 is a relaxed behaviour, note that no interleaving of the memory accesses of x and y leads to that outcome.

This behaviour is permitted by the architectural specifications of ARM and Power [16,31]. As a result, ARM and Power microarchitectures are permitted to perform optimisations that give rise to this outcome, e.g. by reordering the writing thread, reordering the reading thread, or by propagating the writes from the first thread out of order to the second. Indeed, this behaviour can be witnessed by empirically testing ARM and Power processors [5,16].

The programming idiom present in MP, of waiting for a flag to be set, is useful and the relaxed behaviour breaks code that relies on it. Thankfully, processors provide fences, barriers and dependencies that can be used to limit relaxed behaviours. The MP+sync+ctrl test above includes these additions. Power's sync instruction disables reordering of memory accesses across it and ensures that prior writes are propagated before those that follow (ARM's dmb is similar). The if statement creates a so-called *control dependency* from the first load to the second, disabling reordering. As a consequence of these additions, the relaxed behaviour is forbidden in MP+sync+ctrl. Note that the addition of the sync and control dependency cuts relaxed behaviour at the cost of disabling the processor's reordering optimisations, and therefore reduces performance.

(c) The C++ concurrency model.

Programming languages introduce a new layer of complexity: they can compile programs to be run on one sort of processor or another, and so must contend with the possibility of a choice of underlying memory models. In addition, they perform intricate optimisations that introduce yet more relaxed behaviours.

C++ is a programming language with a particularly well-developed concurrency specification [18] that has benefited from extremely close scrutiny from academia. The specification was refined during development, as an effort to produce a matching formal semantics exposed problems with it [8,13]. Subsequent work has identified further flaws [9,11,20, 35], and has even shown that these flaws make the model impossible to fix in its current form [11]. Several research groups are working on a replacement for the current model [11,19,20,27], but none has been adopted, so far.

The C++ concurrency model has some interesting features that distinguish it from processor models, and these will be of particular relevance in considering the question of compositional models. The first peculiarity extends from the desire to introduce concurrency without changing the sequential fragment of the language: neither the sequential behaviour, nor the compilation of sequential blocks of code should change.

Data races. The language separates memory accesses into those that may be used concurrently, the *atomics*, and those that should behave sequentially, the *non-atomics*. The non-atomics come with a convention over their use: programmers must ensure there are no variables with *data races* – writes made concurrently with other accesses, with either the write or the access being non-atomic. This convention provides the compiler with the assurance that no other thread will concurrently measure or perturb a block of non-atomic code, and so it can be optimised as if it is executed in a sequential setting. The penalty for violating the convention that outlaws data races is severe: the entire program is given *undefined behaviour*, and can do anything.

Atomic accesses. The atomic accesses provide a facility to write concurrent code with well defined behaviour. These accesses come in various different flavours. Among them, *relaxed* atomics behave in ways that subsume all relaxed behaviours of the plain memory accesses on the various target architectures (e.g. x86, Power, ARM), and *release* and *acquire* atomics provide ordering that can be used in the MP idiom. This additional ordering forbids relaxed behaviour, requiring the compiler to disable its own optimisations, as well as those of the target processor by emitting barriers, fences and dependencies. As a result, atomics with more ordering at the language level incur a greater performance penalty once compiled. Take MP+rel+acq below, the message passing idiom as it might appear in C++:

MP+rel+acq

$\texttt{store}_{\texttt{rlx}}\texttt{(x,1)}$

$\texttt{store}_{\texttt{rel}}\texttt{(y,1)}$

$\texttt{r1=load}_{\texttt{acq}}\texttt{(y) //reads 1}$

$\texttt{r2=load}_{\texttt{rlx}}\texttt{(x) //reads 0}$

MP+syncs

```
store(x,1)
sync
store(y,1)
```

```
r1=load(y)
sync
{r2=load(x)}
```

The rules of the C++ memory model hinge on a partial order of memory accesses called *happens-before*. Happens-before includes program order and store-release/load-acquire pairs where the load reads from the store. In the execution of MP+rel+acq where the acquire load reads 1, happens-before totally orders the accesses according to the sequence: $\texttt{store}_{\texttt{rlx}}\texttt{(x,1)}$, $\texttt{store}_{\texttt{rel}}\texttt{(y,1)}$, $\texttt{load}_{\texttt{acq}}\texttt{(y)}$, $\texttt{load}_{\texttt{rlx}}\texttt{(x)}$. Loads cannot read from values that are stale in happens-before, so $\texttt{load}_{\texttt{rlx}}\texttt{(x)}$ must read value 1 from the most recent store, and the outcome 1/0 is forbidden.

Now let us consider the implication of this semantics on the compilation of MP+rel+acq. For the moment, we shall ignore compiler optimisation, and consider only what instructions should be emitted to attain the correct behaviour on the Power architecture. If relaxed, release and acquire accesses all map to bare Power accesses, and we compile MP+rel+acq to MP over Power, then the outcome 1/0 is erroneously allowed. To fix this, we must forbid out-of-order propagation of the writes and reordering on each thread. This can be achieved by mapping release and acquire accesses to sequences of Power instructions that include `sync`s or dependencies, to enforce ordering. The MP+syncs test above is the result of compiling $\texttt{store}_{\texttt{rel}}\texttt{(y,1)}$ to `sync;store(y,1)` and $\texttt{load}_{\texttt{acq}}\texttt{(y)}$ to `load(y);sync`. There are more efficient compilation schemes that use a cheaper form of `sync` and dependencies to implement acquire loads, but this scheme is sufficient to correctly implement release and acquire atomics, forbidding the outcome 1/0.

Both LLVM and GCC feature mappings from atomics to the instructions of their target architectures. There has been much work seeking to verify these mappings [12,16,30], including the mechanical verification of the x86 and POWER mappings [13,22].

Optimisations. The MP test, implemented entirely with C++ relaxed accesses, would compile to unadorned MP on Power, allowing the outcome 1/0. And as one might expect, the C++ semantics allows the same outcome of relaxed MP. If we now consider what compiler optimisations would be valid for relaxed MP, it is clear that reordering of either thread's accesses is permitted, as it does not introduce any new behaviour. Contrast this with MP+rel+acq, where the semantics forbids the outcome 1/0: the compiler is not at liberty to reorder the operations in this test. This is just one way that compiler optimisations are intertwined with the memory model.

Dependencies. Both Power's `sync` and C++11's release and acquire have an associated cost. In some instances where ordering is required, there is a cheaper alternative: one can use the dependencies inherent in the program to create order, as in the following examples:

MP + sync + ctrl

```
store(x,1)
sync
store(y,1)
-----------------
r1=load(y)
if(r1 == 1)
    r2=load(x)
```

MP + sync + false ctrl

```
store(x,1)
sync
store(y,1)
-----------------
r1=load(y)
if(r1 == 1)
    r2=load(x)
else
    r2=load(x)
```

In the two programs above, the reading thread's `sync` has been replaced with a conditional statement between the reads. In the first example, whether the second load is performed depends on the value read from `x`, and in the second example it does not. We call dependencies of the second form *false* because although the operations necessary to create a dependency are present, in practice there is none. The treatment of dependencies differs substantially between processor and programming-language memory models. Processor models provide ordering to all dependencies, real or false, and as a consequence, they are used as a cheap way to create ordering. On the other hand, programming-language models seek to admit optmisations that remove false dependencies, so they must not recognise them as a legitimate way to create ordering. In C++, no dependencies create order. This permits optimisations, but also leaves the model fundamentally flawed [11]. Defining a memory model that provides ordering to real dependencies and not false ones is an open research problem with much recent progress [11,19,20,27,28].

2. Verification of program transformations

Recent work verifying program transformations under a C++-like memory model [14] provides the kernel for our vision of a compositional approach to memory models.

The work considers *peep-hole* optimisations, where a block of code is transformed in isolation, leaving the rest of the program untouched. The optimisation will be considered sound if the properties of the original code hold over the transformed code – we call this *observational refinement*.

The reasoning principle is based on a limited form of composition: an optimisation that takes a block of code P_1 and transforms it to P_2 is considered in an arbitrary context with a block-shaped hole, $C(-)$, and we write $C(P_1)$ for the composition of $C(-)$ with P_1. If in every possible context, $C(-)$, $C(P_2)$ is an observational refinement of $C(P_1)$, then the optimisation $P_1 \rightsquigarrow P_2$ is sound.

Take for example P_1, P_2 and C below. The optimisation $P_1 \rightsquigarrow P_2$, called redundant read elimination, has been shown to be sound in C++ [14,35]. The context C is a variant of the MP test, with a block-shaped hole, `{-}`, in place of the usual load of the flag variable `y`. By composing the context with each block, we can form two whole programs: $C(P_1)$ and $C(P_2)$. In $C(P_1)$, the relaxed outcome 1/0 is forbidden. Soundness of the optimisation ensures that 1/0 is forbidden in $C(P_2)$ as well. In both cases the store-release/load-acquire pattern creates happens-before ordering that forbids `load`$_{\texttt{rlx}}$`(x)` from reading the stale value of `x`. Indeed, it

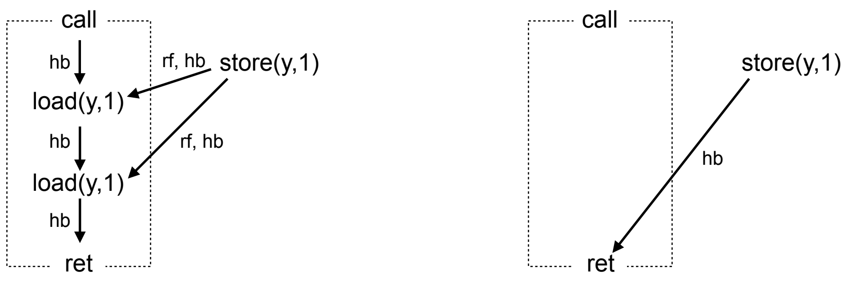

Figure 1. *Left:* Execution of block P_1. *Right:* corresponding history.

would not be sound to optimise P_2 further to $P_3 =$ `{r1=load`$_{\texttt{rlx}}$`(y)}` as this would break the store-release/load-acquire pattern and allow the outcome 1/0.

Block P_1
```
r1=load_acq(y)
r1=load_acq(y)
```

Block P_2
```
r1=load_acq(y)
```

Context C
```
store_rlx(x,1)
store_rel(y,1)
--------------
{-}
if(r1 == 1)
    r2=load_rlx(x)
```

In order to compare two blocks of code, a *denotation* of a code block, written $[\![-]\!]$, is defined. The denotation works by calculating the executions of the code block under a limited set of representative contexts. Each execution is masked to exclude memory interactions that occur entirely within the block or entirely within the context, leaving only the interactions between the two. These masked executions are called *histories*, and together they form the denotation.

For our example blocks P_1 and P_2 above, the context C generates a set of executions whose histories are included in the denotation. The execution of block P_1 with outcome 1/1 is drawn in Figure 1 together with the corresponding history. Each is a trace of memory accesses, with *call* and *ret* bounding the part that arises from the block, and tracking the values of local variables, e.g. `r1`. The execution includes rf edges indicating which store a given load reads from, and hb edges where internal load-acquire accesses pair with the store-release in the context to create happens-before ordering. This ordering is captured in the history as a happens-before edge from the context store to the block return (this edge is missing from the corresponding history of P_3). In the history, all of the internal structure of the block is erased, including the fact that there are two loads in P_1.

A pair of denotations, $[\![P_1]\!]$ and $[\![P_2]\!]$, can be compared with an *abstraction relation*, written $[\![P_2]\!] \sqsubseteq [\![P_1]\!]$. The abstraction relation ensures that the histories of $[\![P_2]\!]$ are a subset of those in $[\![P_1]\!]$, so P_1 can interact with its context in at least as many ways as P_2, as is the case for our example above. The paper provides a proof that abstraction implies observational refinement, so to validate the optimisation $P_1 \rightsquigarrow P_2$, one only need show that $[\![P_2]\!] \sqsubseteq [\![P_1]\!]$, and this can be checked automatically with a tool.

The notions of denotation and abstraction will turn out to be central in our vision of a compositional approach to relaxed memory.

3. Compositional relaxed memory

The current crop of relaxed memory models can only be used to calculate the behaviour of a *whole* program written under a single memory model. Instead, we would like to consider a program as an aggregate of components over different models, composed together.

Composing a block P with a context $C(-)$ is only one sort of composition – there are a variety of others: e.g. sequential composition, parallel composition, and composition through function calls. A compositional semantics will have to describe all of the various ways in which code can be composed. Let us capture the gamut of syntactic composition with a new operator $\cdot$. If $P \cdot C$ represents a whole program, then we can use existing models to calculate its behaviour under memory model M, writing this as $[\![P \cdot C]\!]_M$.

In developing this compositional model of relaxed memory, we will need a denotation that captures the interaction between a component and its context in a way that is sensitive to the memory model used within the component, and agnostic to those models used in the context. We shall write $[\![P]\!]_M$ for the denotation of a component of code, P, under memory model M.

These denotations will be combined using an operator, $\bullet$, that produces a new denotation representing the behaviour of the two components combined: e.g. $[\![P_1]\!]_{M_1} \bullet [\![P_2]\!]_{M_2}$.

Denotations will be compared with an abstraction relation, $\sqsubseteq$. Given a pair of denotations, $[\![P_1]\!]_{M_1}$ and $[\![P_2]\!]_{M_2}$, if $[\![P_1]\!]_{M_1} \sqsubseteq [\![P_2]\!]_{M_2}$, then the first is an observational refinement of the second and P_1 can substitute P_2 in any context without introducing new behaviour.

(a) Memory-model aware denotation

Let us explore the shape of the memory-model aware denotation. The denotation must distill the ways in which a component interacts with its surrounding code. In reviewing SC, processor models and the C++ model, we saw some of the diversity that we must cover and several ways in which code might interact. Here, we list aspects of the execution that make up the interface between components, and therefore must be tracked.

Values read. Even in a sequential setting, the values that are read from a component form part of the interaction with its context, and so the denotation will have to track the relationship between the value of variables in the context and those written by the component, and vice versa. The call and return of the block history of Figure 1 serve this purpose, and may be reused.

Ordering. Consider two components executing under the C++ model: $P_1 = \texttt{load}_{\texttt{acq}}\texttt{(y)}$ and $P_2 = \texttt{load}_{\texttt{rlx}}\texttt{(y)}$. Take $C(-)$ to be the context such that $C \cdot P_1$ is the MP+rel+acq litmus test. If we were to replace P_1 with P_2, then the litmus test would admit a new outcome – the relaxed behaviour 1/0. This means that P_2 does not observationally refine P_1, and the denotations of the two components must differ. The difference between the relaxed load and the acquire load is that the acquire load creates ordering. It is clear then that our denotation will have to be sensitive to ordering.

Again, the history in Figure 1 contains this information by tracking happens-before between the block and memory accesses from the context. This is a good starting point for a compositional denotation, but happens-before is specific to the C++ memory model, and other models define ordering differently. When components written under different models are composed, e.g. $[\![P_1]\!]_{M_1} \bullet [\![P_2]\!]_{M_2}$, it will be necessary to capture the ordering provided by P_2 in terms that fit with M_1 and vice versa. This might be achieved by including a pairwise translation between models M_1 and M_2 in the definition of the composition operator $\bullet$, or the denotation of each model might be defined in an overarching model that reifies the guarantees provided by each component.

Dependencies. Recall that dependencies are a cheap way to reliably create ordering on hardware models. On language models, dependencies may or may not create ordering. If a language model does provide ordering to dependencies, one must take care not to rely on false dependencies that might be optimised away.

Let the component P_1 be the second thread of the MP+sync+ctrl litmus test, and let P_2 be its false-dependency counterpart. Because language models ignore the false dependency and provide ordering to the real one, in composition with the first thread of the MP+sync+ctrl litmus test, P_1 and P_2 exhibit different behaviours. This means that their denotations must differ, and that dependencies must be tracked.

The history of Figure 1 is built above a model that cannot correctly distinguish real and fake dependencies [11], and as a consequence it does not either. Several new models of concurrency that deal correctly with dependencies [11,19,20,27] might support an improved denotation.

Conventions. In some memory models, a requirement is placed on the programmer obliging them to avoid writing certain sorts of code. This is known as a *catch-fire* semantics, because if the requirement is violated, then the model no longer constrains the behaviour of the program, and it can do anything including going up in flames.

A catch-fire semantics is useful for enforcing *conventions* on the programmer. One might imagine a convention capturing a global invariant, or a protocol for the use of a library. If the programmer obeys the convention, then they get a predictable semantics. If not, then all bets are off. This means that the convention can be assumed to hold, and used as an invariant during reasoning and compilation. If the assumption is violated then the program can do anything, and there is no need to compile it correctly.

C++ makes use of this mechanism, forbidding races to establish an invariant that permits sequential optimisation. Suppose we have two components, say P_1 and P_2, where separately neither violates the convention of race freedom. Internally, these components write to the same variable non-atomically, so when they are composed in parallel, they form a race. In order to provide observational refinement, our abstraction relation will have to be sensitive to conventions, and our denotations must track enough information to check them.

4. Goals of the approach

With a denotation, composition operator and abstraction relation, it will be possible to start considering the properties of code made up of varied components. This will enable compositional verification: verification of program components without the need to consider a concrete context, verification above the varied CPU/GPU systems that we use today, and the application of mature SC verification techniques in the relaxed setting. To make this more concrete, we propose three verification goals – built upon our hypothetical definitions – that cannot be addressed with the current approach to memory models.

Denotational C++. The first goal is a denotational relaxed memory model for C++. In this model, the semantics of one's program would be built from the ground up by composing the denotations of smaller components. To ensure soundness and completeness, one must show that denotational composition is equivalent to syntactic composition, i.e. where $=$ is the symmetric closure of $\sqsubseteq$:

$$[\![P_1 \cdot P_2]\!]_{\mathrm{C++}} = [\![P_1]\!]_{\mathrm{C++}} \bullet [\![P_2]\!]_{\mathrm{C++}}$$

Heterogeneous hardware semantics. The second goal is to reason about programs that are the composition of components with differing memory models. One property to target might be compilation soundness for the heterogeneous language OpenCL. OpenCL can target both the CPU and the GPU of a system. In the following example, we consider the compilation of OpenCL component P to two components: one part to be executed on the x86 CPU, $P_{\rightsquigarrow \mathrm{x86}}$, and the other, $P_{\rightsquigarrow \mathrm{PTX}}$, to be executed under the Nvidia PTX memory model on the GPU. We would like to know that the composition of these two compiled components is an observational refinement of the original code:

$$[\![P_{\rightsquigarrow \text{x86}}]\!]_{\text{x86}} \bullet [\![P_{\rightsquigarrow \text{PTX}}]\!]_{\text{PTX}} \sqsubseteq [\![P]\!]_{\text{OpenCL}}$$

Support SC verification techniques over SC components. One of the central design goals of C++ concurrency is to support a property called DRF-SC. The property states that if a whole program $P \cdot C$ is data-race free and uses a restricted set of the concurrency primitives, it will behave as if it is sequentially consistent, i.e. $[\![P \cdot C]\!]_{\text{C++}} \sqsubseteq [\![P \cdot C]\!]_{\text{SC}}$. At first, this seems like a valuable property that might support SC verification techniques in real-world C++ code. Unfortunately, the whole-program nature of the property makes sure that it is unlikely to apply in practice: the restrictions on the use of concurrency primitives apply not only to P, but to C too.

With the approach imagined here, we can state a much more usable property, composing race-free and concurrency-feature-restricted component P with a context that is not restricted, C, while maintaining the ability to reason about P using SC verification tools. We call this property *component SC*:

$$[\![P \cdot C]\!]_{\text{C++}} \sqsubseteq [\![P]\!]_{\text{SC}} \bullet [\![C]\!]_{\text{C++}}$$

Component SC strengthens the memory model that underlies P, so that with some adaptation we may now use SC logics like CoLoSL [29] for validating properties of P. In fact, there is a range of state-of-the-art concurrent logics with weaker underlying memory models (OGRA [21], RSL [36], FSL [15], GPS [34]). Our imagined denotational approach allows one to choose the ideal model for each component: strong enough to use a less intricate logic, yet weak enough to admit a high-performance implementation.

5. Conclusion

Recent work has made tremendous strides in understanding the exotic behaviours exhibited by mainstream concurrent systems. A robust method for measuring, modelling and verifying the goals of each concurrency design has emerged, and techniques developed in academia are having strong impacts on industrial practice. The result is a series of rapidly improving models of the concurrency of individual processors and programming languages. This relaxed memory research is matched by an equally dynamic and industrially-relevant development of SC verification techniques.

Unfortunately, each of these strains of research has a mismatch that limits its applicability: relaxed models are non-compositional, applying only to whole programs written in a single paradigm, and SC verification techniques maintain the idealised assumption of SC.

This paper has suggested a path forward, by developing a compositional form of memory model that matches the way we really think about programs: as aggregates of varied components. It has provided some insights about how this model might look, introducing the concepts of denotation, composition and abstraction. And it has posed some concrete goals for the approach: a denotational semantics for C++, the verification of compilation of a heterogeneous language, and the application of advanced SC reasoning techniques in the relaxed setting.

Competing Interests. 'The author declares that he has no competing interests'.

Acknowledgements. I am grateful to Mike Dodds and John Wickerson for discussing the ideas presented here.

Funding. This work is supported by the Royal Academy of Engineering and the Lloyd's Register Foundation.

References

1. Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson.
   GPU concurrency: Weak behaviours and programming assumptions.
   In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 577–591, 2015.
2. Jade Alglave, Anthony C. J. Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli.
   The semantics of power and ARM multiprocessor machine code.
   In *Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009*, pages 13–24, 2009.
3. Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter 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.
4. Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell.
   Litmus: Running tests against hardware.
   In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 41–44, 2011.
5. Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter 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.
6. Jade Alglave, Luc Maranget, and Michael Tautschnig.
   Herding cats: Modelling, simulation, testing, and data mining for weak memory.
   ACM Trans. Program. Lang. Syst. (TOPLAS), 36(2):7:1–7:74, 2014.
7. Jade Alglave, Luc Maranget, and Michael Tautschnig.
   Herding cats: modelling, simulation, testing, and data-mining for weak memory.
   TOPLAS, 2014.
8. Mark Batty.
   The C11 and C++11 Concurrency Model.
   PhD thesis, University of Cambridge, 2014.
9. Mark Batty, Mike Dodds, and Alexey Gotsman.
   Library abstraction for C/C++ concurrency.
   In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 235–248, New York, NY, USA, 2013. ACM.
10. Mark Batty, Alastair F. Donaldson, and John Wickerson.
    Overhauling SC atomics in c11 and opencl.
    In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 634–648, 2016.
11. Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell.
    The problem of programming language concurrency semantics.
    In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 283–307, 2015.
12. Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell.
    Clarifying and compiling C/C++ concurrency: from C++11 to POWER.
    In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 509–520, 2012.
13. Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber.
    Mathematizing C++ concurrency.
    In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 55–66, 2011.

14. Mike Dodds, Mark Batty, and Alexey Gotsman.
Compositional verification of relaxed-memory program transformations.
2017.
under submission.
15. Marko Doko and Viktor Vafeiadis.
A program logic for C11 memory fences.
In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, pages 413–430, New York, NY, USA, 2016. Springer-Verlag New York, Inc.
16. Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell.
Modelling the armv8 architecture, operationally: concurrency and ISA.
In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 608–621, 2016.
17. Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell.
Mixed-size concurrency: Arm, power, c/c++11, and SC.
In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 429–442, 2017.
18. ISO/IEC.
Programming Languages – C++, 14882:2011.
19. Alan Jeffrey and James Riely.
On thin air reads towards an event structures model of relaxed memory.
In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 759–767, 2016.
20. Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer.
A promising semantics for relaxed-memory concurrency.
In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 175–189, 2017.
21. Ori Lahav and Viktor Vafeiadis.
Owicki-gries reasoning for weak memory models.
In *Proceedings, Part II, of the 42Nd International Colloquium on Automata, Languages, and Programming - Volume 9135*, ICALP 2015, pages 311–323, New York, NY, USA, 2015. Springer-Verlag New York, Inc.
22. Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer.
Repairing sequential consistency in C/C++11.
In *PLDI '17: Proceedings of the 2017 ACM SIGPLAN conference on Programming language design and implementation*, 2017.
23. Leslie Lamport.
How to make a correct multiprocess program execute correctly on a multiprocessor,.
Computers, IEEE Transactions on, 46(7):779–782, 1997.
24. Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin.
Generating litmus tests for contrasting memory consistency models.
In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 273–287, 2010.
25. Robin Morisset, Pankaj Pawan, and Francesco 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.
26. Scott Owens, Susmit Sarkar, and Peter Sewell.
A better x86 memory model: x86-tso.
In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 391–407, 2009.
27. Jean Pichon-Pharabod and Peter Sewell.
A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions.
In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL '16, pages 622–633, New York, NY, USA, 2016. ACM.

28. Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski.
Operational aspects of C/C++ concurrency.
CoRR, abs/1606.01400, 2016.
29. Azalea Raad, Jules Villard, and Philippa Gardner.
CoLoSL: Concurrent Local Subjective Logic.
In *Proceedings of the 24th European Symposium on Programming (ESOP)*, pages 710–735, 2015.
30. Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams.
Synchronising C/C++ and POWER.
In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 311–322, 2012.
31. Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek 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.
32. Jaroslav Ševčík and David Aspinall.
On validity of program transformations in the java memory model.
In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, pages 27–51, 2008.
33. Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen.
X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors.
CACM, pages 89–97, 2010.
34. Aaron Turon, Viktor Vafeiadis, and Derek Dreyer.
GPS: Navigating weak memory with ghosts, protocols, and separation.
pages 691–707, 2014.
35. Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli.
Common compiler optimisations are invalid in the C11 memory model and what we can do about it.
In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 209–220, 2015.
36. Viktor Vafeiadis and Chinmay 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 &#38; Applications*, OOPSLA '13, pages 867–884, New York, NY, USA, 2013. ACM.
37. John Wickerson, Mark Batty, Bradford M. Beckmann, and Alastair F. Donaldson.
Remote-scope promotion: clarified, rectified, and verified.
In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 731–747, 2015.