# Industrial concurrency specification for C/C++

Mark Batty
University of Kent

# It is time for mechanised industrial standards

Specifications are written in English prose: this is insufficient

Write mechanised specs instead (formal, machine-readable, executable)

Designers can scrutinise, research questions can be identified

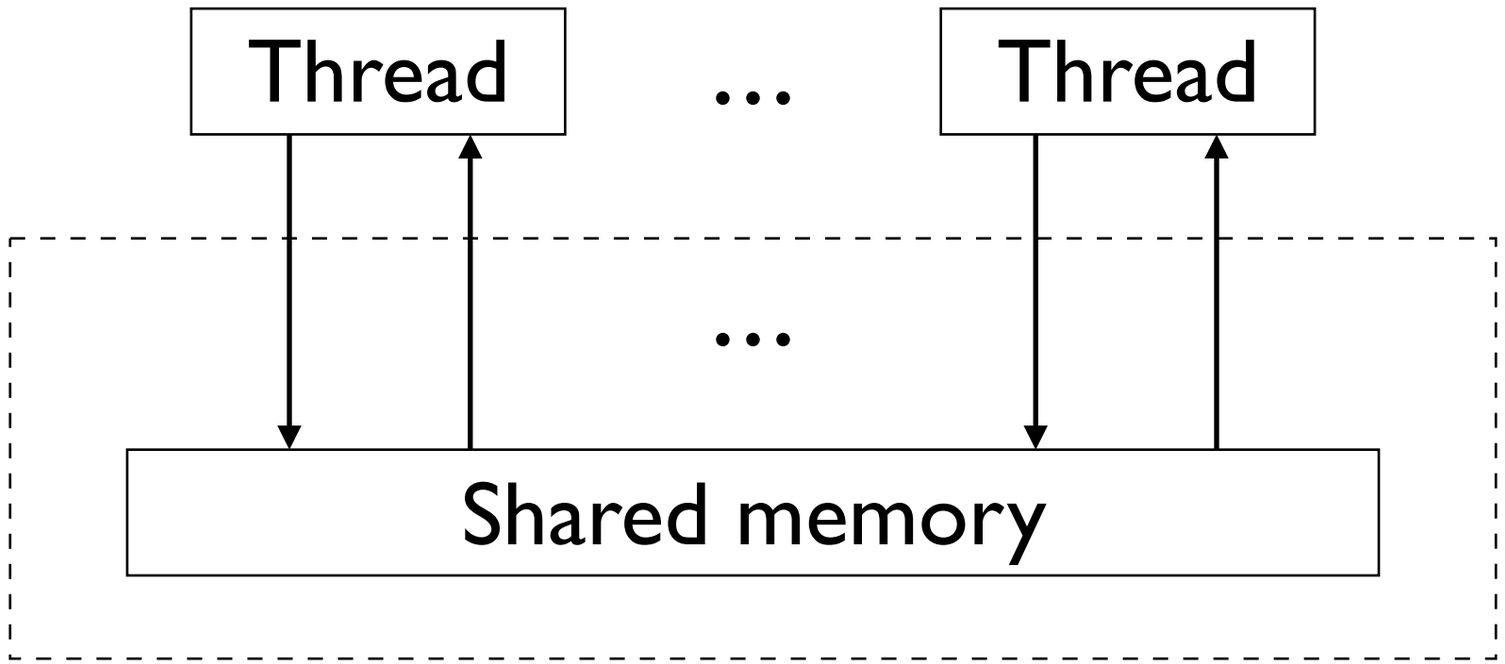
Mechanised specs enable verification for secure systems

Writing mechanised specifications is practical now

# A case study:
# industrial concurrency specification

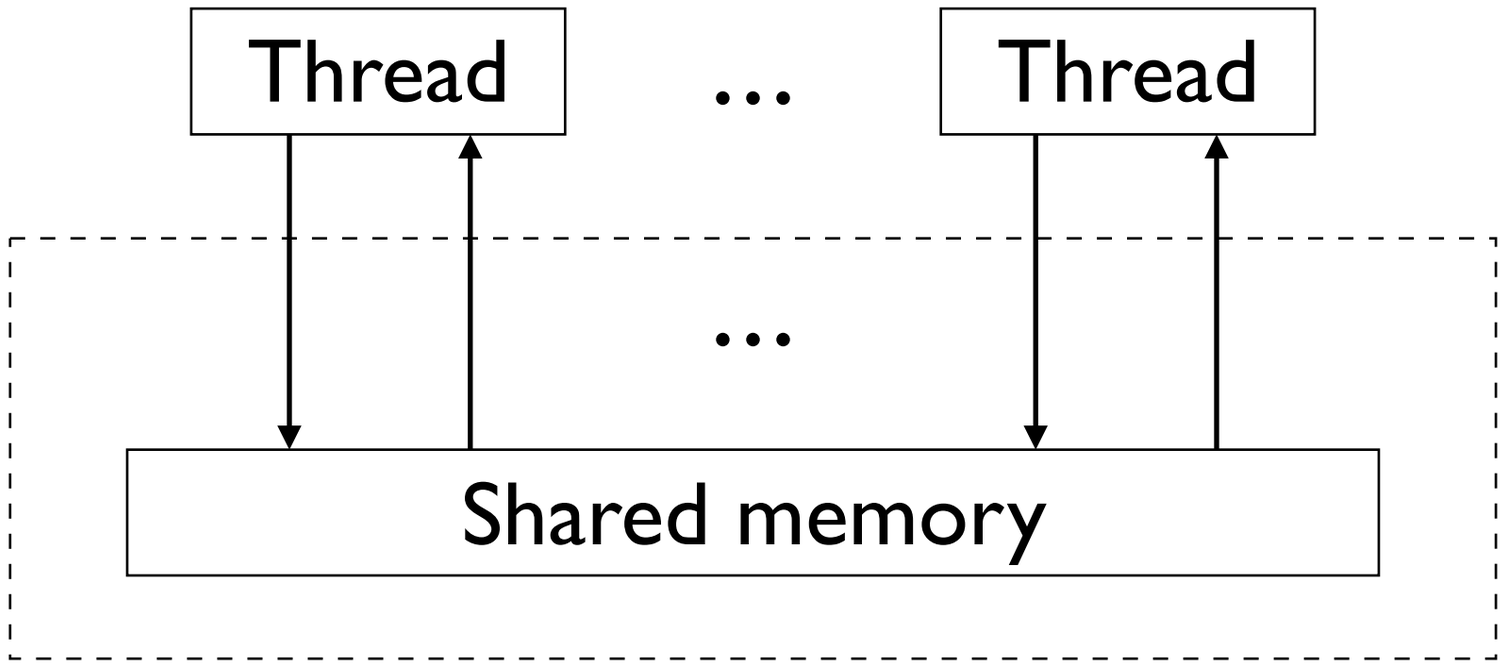
# Shared memory concurrency

Multiple threads communicate through a shared memory



# Shared memory concurrency

Multiple threads communicate through a shared memory




Most systems use a form of shared memory concurrency:

# An example programming idiom

`data`, `flag`, `r` initially zero

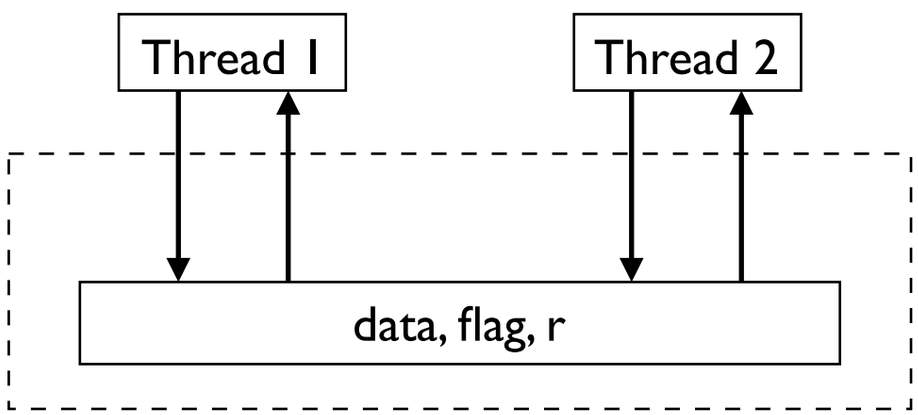
Thread 1:

```
data = 1;
flag = 1;
```

Thread 2:

```
while (flag==0)
   {};
r = data;
```

In the end `r==1`




Sequential consistency: simple interleaving of concurrent accesses

Reality: more complex

# An example programming idiom

`data`, `flag`, `r` initially zero

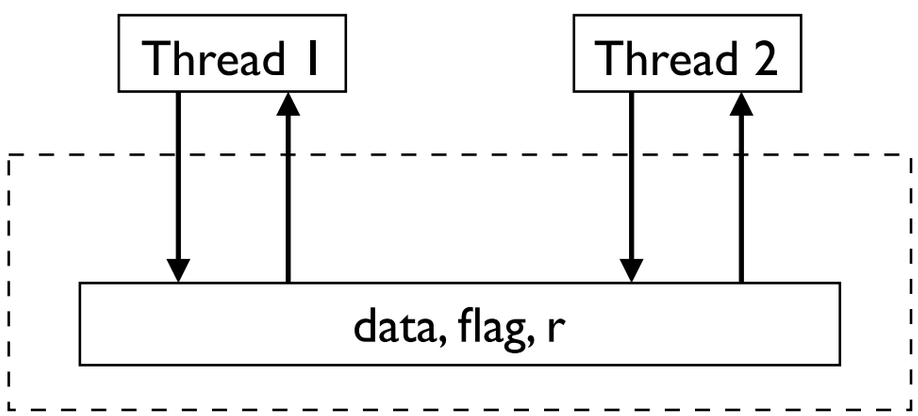
Thread 1:

```
data = 1;
flag = 1;
```

Thread 2:

```
while (flag==0)
   {};
r = data;
```

In the end `r==1`




Sequential consistency: simple interleaving of concurrent accesses

Reality: more complex

# Relaxed concurrency

Memory is slow, so it is optimised (buffers, caches, reordering…)

e.g. IBM's machines allow reordering of unrelated writes

(so do compilers, ARM, Nvidia…)

`data`, `flag`, `r` initially zero

Thread 1:

```
data = 1;
flag = 1;
```

Thread 2:

```
while (flag==0)
  {};
r = data;
```

In the end `r==1`

Sometimes, in the end `r==0`, a relaxed behaviour

Many other behaviours like this, some far more subtle, leading to trouble

# Relaxed concurrency

Memory is slow, so it is optimised (buffers, caches, reordering…)

e.g. IBM's machines allow reordering of unrelated writes

(so do compilers, ARM, Nvidia…)

`data`, `flag`, `r` initially zero

| Thread 1: | Thread 2: |
|---|---|
| `flag = 1;` | `while (flag==0)` |
| `data = 1;` | `  {};` |
| | `r = data;` |

In the end `r==1`

Sometimes, in the end `r==0`, a relaxed behaviour

Many other behaviours like this, some far more subtle, leading to trouble

# Relaxed behaviour leads to problems

Bugs in deployed processors

Many bugs in compilers

Bugs in language specifications

Bugs in operating systems

Power/ARM processors:
unintended relaxed behaviour
observable on shipped machines

[AMSS10]

# Relaxed behaviour leads to problems

Bugs in deployed processors

Many bugs in compilers

Bugs in language specifications

Bugs in operating systems

Errors in key compilers (GCC, LLVM): compiled programs could behave outside of spec.

[MPZN13, CV16]

# Relaxed behaviour leads to problems

Bugs in deployed processors

Many bugs in compilers

Bugs in language specifications

Bugs in operating systems

The C and C++ standards had bugs that made unintended behaviour allowed.

More on this later.

[BOS+11, BMN+15]

# Relaxed behaviour leads to problems

Bugs in deployed processors

Many bugs in compilers

Bugs in language specifications

Bugs in operating systems

Confusion among operating system engineers leads to bugs in the Linux kernel

[McK11, SMO+12]

# Relaxed behaviour leads to problems

Bugs in deployed processors

Many bugs in compilers

Bugs in language specifications

Bugs in operating systems

Current engineering practice is severely lacking!

# Vague specifications are at fault

Relaxed behaviours are subtle, difficult to test for and often unexpected, yet allowed for performance

Specifications try to define what is allowed, but English prose is untestable, ambiguous, and hides errors

# A diverse and continuing effort

Modelling of hardware and languages

Simulation tools and reasoning principles

Empirical testing of current hardware

Verification of language design goals

Test and verify compilers

Feedback to industry: specs and test suites

Build mechanised executable formal models of specifications

[AFI+09,BOS+11,BDW16]
[FGP+16,LDGK08,OSP09]
[FSP+17]

# A diverse and continuing effort

Modelling of hardware and languages

Simulation tools and reasoning principles

Empirical testing of current hardware

Verification of language design goals

Test and verify compilers

Feedback to industry: specs and test suites

Provide tools to simulate the formal models, to explain their behaviours to non-experts

Provide reasoning principles to help in the verification of code

[BOS+11,SSP+,BDG13]

# A diverse and continuing effort

Modelling of hardware and languages

Simulation tools and reasoning principles

Empirical testing of current hardware

Verification of language design goals

Test and verify compilers

Feedback to industry: specs and test suites

Run a battery of tests to understand the observable behaviour of the system and check it against the model

[AMSS'11]

# A diverse and continuing effort

Modelling of hardware and languages

Simulation tools and reasoning principles

Empirical testing of current hardware

Verification of language design goals

Test and verify compilers

Feedback to industry: specs and test suites

Explicitly stated design goals should be proved to hold

[BMN+15]

# A diverse and continuing effort

Modelling of hardware and languages

Simulation tools and reasoning principles

Empirical testing of current hardware

Verification of language design goals

Test and verify compilers

Feedback to industry: specs and test suites

Test to find the relaxed behaviours introduced by compilers and verify that optimisations are correct

[MPZN13, CV16]

# A diverse and continuing effort

Modelling of hardware and languages

Simulation tools and reasoning principles

Empirical testing of current hardware

Verification of language design goals

Test and verify compilers

Feedback to industry: specs and test suites

Specifications should be fixed when problems are found

Test suites can ensure conformance to formal models

[B11]

# A diverse and continuing effort

Modelling of hardware and languages

Simulation tools and reasoning principles

Empirical testing of current hardware

Verification of language design goals

Test and verify compilers

Feedback to industry: specs and test suites

## I will describe my part:

# The C and C++ memory model

# Acknowledgements

M. Dodds

A. Gotsman

K. Memarian

K. Nienhuis

S. Owens

J. Pichon-Pharabod

S. Sarkar

P. Sewell

T. Weber

# C and C++

**The** medium for system implementation

Defined by WG14 and WG21 of the International Standards Organisation

The ’11, ’14 and ‘17 revisions define relaxed memory behaviour


# C and C++

**The** medium for system implementation

Defined by WG14 and WG21 of the International Standards Organisation

The '11, '14 and '17 revisions define relaxed memory behaviour

We worked with the ISO, formalising and improving their concurrency design


# C++11 concurrency design

A contract with the programmer: they must avoid **data races**,
two threads competing for simultaneous access to a single variable

`data` initially zero

| Thread 1: | Thread 2: |
| --- | --- |
| `data = 1;` | `r = data;` |

**Beware:**
Violate the contract and the compiler is free to allow anything: catch fire!

# C++11 concurrency design

A contract with the programmer: they must avoid **data races**,
two threads competing for simultaneous access to a single variable

`data` initially zero

| Thread 1: | Thread 2: |
|---|---|
| `data = 1;` | `r = data;` |

**Beware:**
Violate the contract and the compiler is free to allow anything: catch fire!

# C++11 concurrency design

A contract with the programmer: they must avoid **data races**,
two threads competing for simultaneous access to a single variable

`data` initially zero

| Thread 1: | Thread 2: |
| --- | --- |
| `data = 1;` | `r = data;` |

**Beware:**
Violate the contract and the compiler is free to allow anything: catch fire!

Atomics are excluded from the requirement, and can order non-atomics,
preventing simultaneous access and races

# C++11 concurrency design

A contract with the programmer: they must avoid **data races**,
two threads competing for simultaneous access to a single variable

`data`, `r`, `atomic flag`, initially zero

Thread 1:

```
data = 1;
flag = 1;
```

Thread 2:

```
while (flag==0)
  {};
r = data;
```

**Beware:**
Violate the contract and the compiler is free to allow anything: catch fire!

Atomics are excluded from the requirement, and can order non-atomics,
preventing simultaneous access and races

# Design goals in the standard

The design is complex but the standard claims a powerful simplification:

C++11/14: §1.10p21

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 [according to] "sequential consistency".

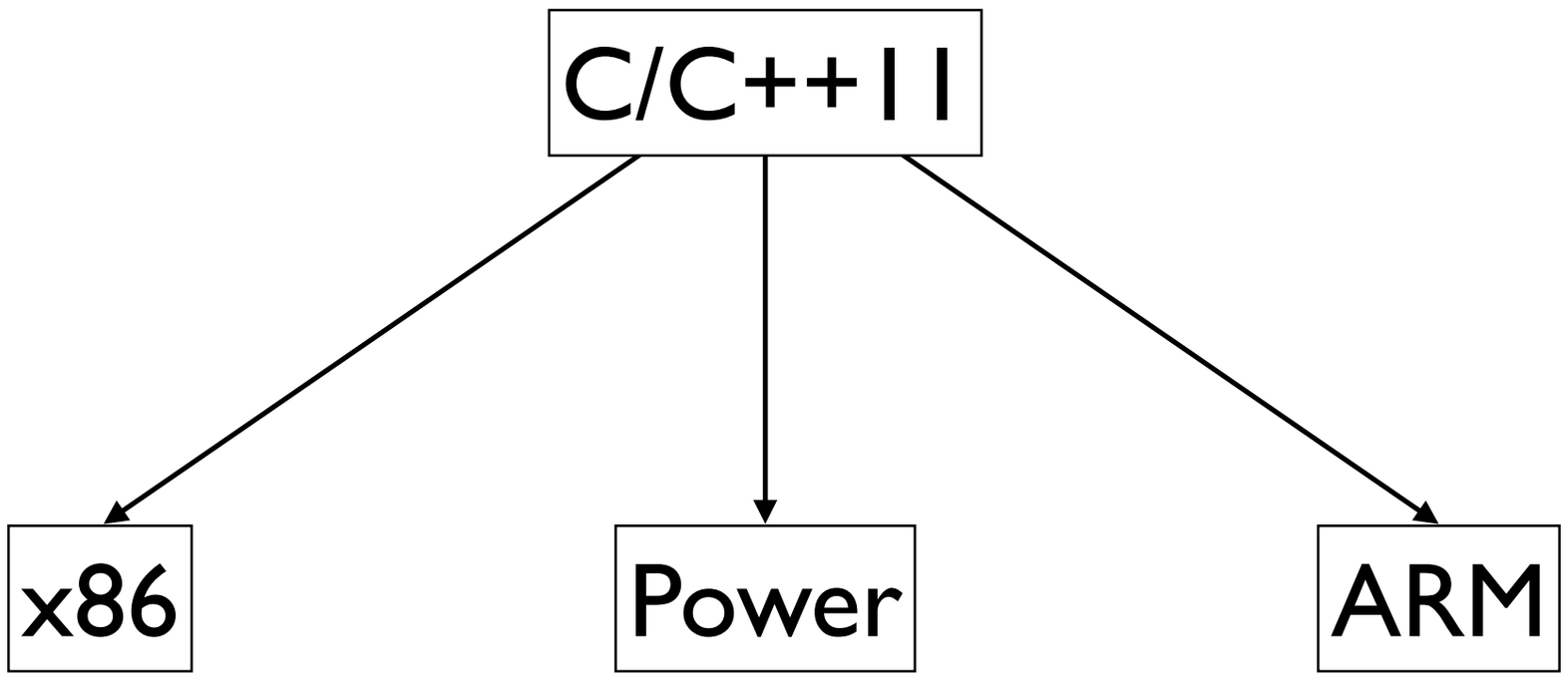
This is the central design goal of the model, called DRF-SC

# Implicit design goals

Compilers like GCC, LLVM map C/C++ to pieces of machine code

| C/C++ | Power | ARM | x86 |
|---|---|---|---|
| Load `acquire` | ld; cmp; bc; isync | ldr; dmb | MOV (from memory) |

Each mapping should preserve the behaviour of the original program



# We formalised a draft of the standard

A mechanised formal model, close to the standard text

## C++11 standard §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.

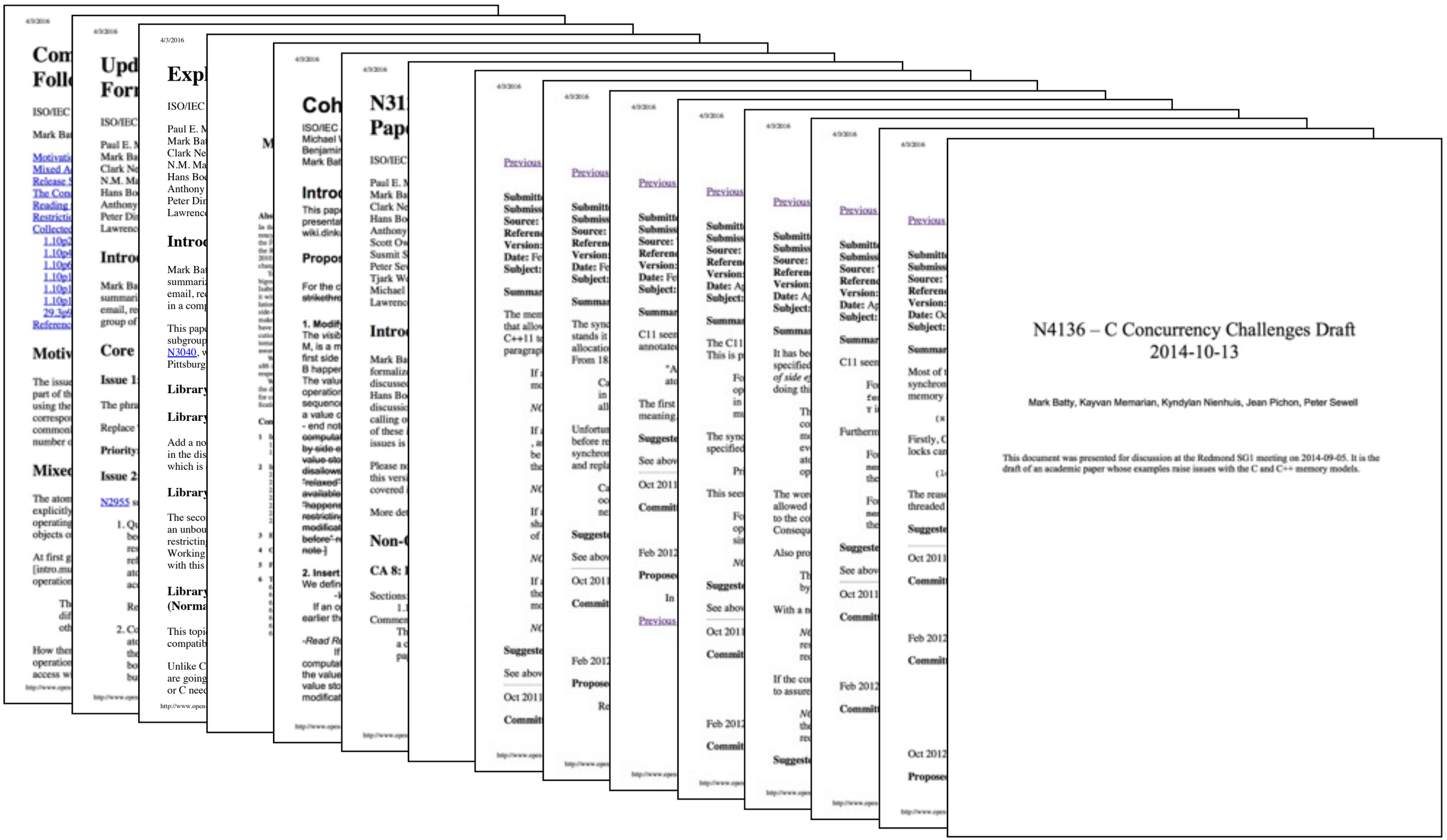
## The corresponding formalisation:

let *happens_before sb ithb* = *sb* ∪ *ithb*

let *consistent_hb hb* =
    isIrreflexive (transitiveClosure *hb*)

# Communication with WG21 and WG14

## Issues were discussed in N-papers and Defect Reports



# Major problems fixed, key properties verified

**DRF-SC:**

The central design goal, was **false**, the standard permitted too much

Fixed the model and then proved (in HOL4) that the goal is now **true**

Fixes were incorporated, pre-ratification, and are in C++11/14

**Compilation mappings:**

Efficient x86, Power mappings are sound [BOS+11,BMO+12,SMO+12]

**Reasoning:**

Developed a reasoning principle for proving programs correct [BDO13]

# Timing was everything

Achieved direct impact on the standard

Making this work was partly a social problem

C++11 was a major revision, so the ISO was receptive to change

But…

# A fundamental problem uncovered

`x`, `y`, `r1`, `r2` initially zero

```
// Thread 1                  // Thread 2
r1 = x;                      r2 = y;
if(r1==1) y = 1;             if(r2==1) x = 1;
```

Can we observe `r1==1`, `r2==1` at the end?

# A fundamental problem uncovered

x, y, r1, r2 initially zero

```
// Thread 1                    // Thread 2
r1 = x;                        r2 = y;
if(r1==1) y = 1;               if(r2==1) x = 1;
```

Can we observe r1==1, r2==1 at the end?

The write of y is dependent on the read of x

# A fundamental problem uncovered

`x`, `y`, `r1`, `r2` initially zero

```
// Thread 1                  // Thread 2
r1 = x;                      r2 = y;
if(r1==1) y = 1;             if(r2==1) x = 1;
```

Can we observe `r1==1`, `r2==1` at the end?

The write of `y` is dependent on the read of `x`

The write of `x` is dependent on the read of `y`

# A fundamental problem uncovered

`x`, `y`, `r1`, `r2` initially zero

```
// Thread 1                  // Thread 2
r1 = x;                      r2 = y;
if(r1==1) y = 1;             if(r2==1) x = 1;
```

Can we observe `r1==1`, `r2==1` at the end?

The write of `y` is dependent on the read of `x`

The write of `x` is dependent on the read of `y`

`1/1` never occurs in compiled code, and ought to be forbidden

# A fundamental problem uncovered

`x`, `y`, `r1`, `r2` initially zero

```
// Thread 1              // Thread 2
r1 = x;                  r2 = y;
if(r1==1) y = 1;         if(r2==1) x = 1;
```

Can we observe `r1==1`, `r2==1` at the end?

The write of `y` is dependent on the read of `x`

The write of `x` is dependent on the read of `y`

`1/1` never occurs in compiled code, and ought to be forbidden

"[ *Note:* […] However, implementations should not allow such behavior. — *end note* ]"

ISO: notes carry no force, and "should" imposes no constraint, so yes!

# A fundamental problem uncovered

**Why?** Dependencies are ignored to allow dependency-removing optimisations

C++ Should respect the left-over dependencies

We have proved that no fix exists in the structure of the current specification

This identifies a difficult research problem

The writ

The writ

`1/1` ne

*"[ Note: [...] However, implementations should not allow such behavior. — end note ]"*

ISO: notes carry no force, and "should" imposes no constraint, so yes!

# The thin-air problem

# A slightly different program

x, y, r1, r2 initially zero

```
// Thread 1                  // Thread 2
r1 = x;                      r2 = y;
if(r1==1) y = 1;             if(r2==1) {x = 1}
                             else {x = 1}
```

Can we observe r1==1, r2==1 at the end?

In both branches on Thread 2, 1 is written to x

An optimising compiler may perform common subexpression elimination

# A slightly different program

`x`, `y`, `r1`, `r2` initially zero

```
// Thread 1                 // Thread 2
r1 = x;                     r2 = y;
if(r1==1) y = 1;            x = 1;
```

Can we observe `r1==1`, `r2==1` at the end?

In both branches on Thread 2, `1` is written to `x`

An optimising compiler may perform common subexpression elimination

In the altered program, ARM would allow the outcome `1 / 1`

# A slightly different program

`x`, `y`, `r1`, `r2` initially zero

```
// Thread 1                  // Thread 2
r1 = x;                      r2 = y;
if(r1==1) y = 1;             if(r2==1) {x = 1}
                             else {x = 1}
```

Can we observe `r1==1`, `r2==1` at the end?

We need a semantic notion of dependency

Execution depends on more than one control-flow path

C++ spec considers one at a time: fundamental change is needed

# Current work

**Several candidates:**

- The Promising Semantics [KHL+17] — an abstract machine with speculation of writes through *promises*. At each step, promised writes must be sure to execute.

- Jeffrey and Riely [JR16] — based on event structures, executions are built up iteratively, out of order. Add a read only if the write it reads from must be executed.

- Podkopaev, Sergey, and Nanevski [PSN16] — an abstract machine where conditionals can be preemptively explored, and writes that always occur can be promoted.

- Bubbly and Ticky semantics [PPS16] — based on event structures. The event structure is non-deterministically mutated in a transition system that mimics compiler optimisations.

Each relies on a repeated search over multiple control-flow paths

This makes the models more expensive to evaluate than C++ (cannot use SAT)

Which model to choose?

# Model simulation

Simulators provide lightweight automatic validation of design criteria:
[WBSC17] uses SAT to check DRF-SC, compiler mappings, litmus tests for C++, OpenCL, CPUs, GPUs

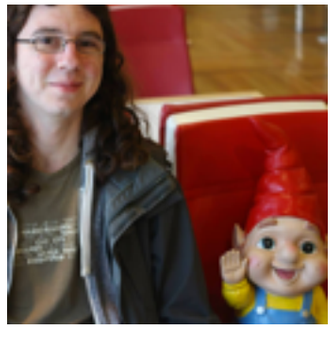
We are building a simulator for thin-air models

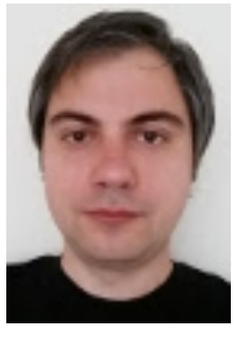
Higher complexity requires advanced Quantified Boolean Formula solvers

A VeTTS grant is paying for the development of a web interface

Our own solution to the thin-air problem is under development:

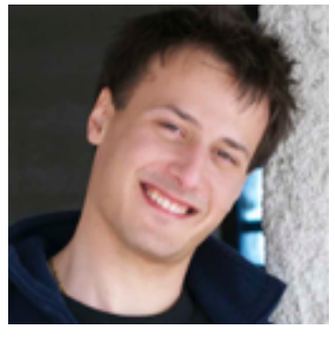
a compositional denotational semantics that looks rather different [B17]


Simon Cooksey


Radu Grigore

University of Kent
Sarah Harris


Scott Owens


Marco Paviotti

# Conclusion

Mechanised industrial specification is practical, with a valuable payoff:

- Improved specs
- Simulators — an executable golden model that matches the spec
- Test suites can be generated
- Design criteria can be validated

It can guide us to future research questions

It is a necessary step in formal verification of security properties

[ABD+15] J. Alglave, M. Batty, A. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, J. Wickerson. GPU concurrency: weak behaviours and programming assumptions. ASPLOS'15

[AFI+09] 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. DAMP'09

[AMSS10] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. CAV'10

[AMSS'11] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. TACAS'11/ETAPS'11

[B17] M. Batty. Compositional relaxed concurrency. Philosophical Transactions A, 2017

[B11] P. Becker, editor. Programming Languages — C++. 2011. ISO/IEC 14882:2011. A non-final version is available at http://www.open-std.org/jtc1/sc22/ wg21/docs/papers/2011/n3242.pdf.

[BDG13] M. Batty, M. Dodds, A. Gotsman. Library Abstraction for C/C++ Concurrency. POPL'13

[BDW16] M. Batty, A. Donaldson, J. Wickerson. Overhauling SC atomics in C11 and OpenCL. POPL'16

[BMN+15] M. Batty, K. Memarian, K. Nienhuis, J. Pichon, P. Sewell. The Problem of Programming Language Concurrency Semantics. ESOP'15

[BMO+12] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++0x to POWER. POPL'12

[BOS+11] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. POPL'11

[CV16] S. Chakraborty, V. Vafeiadis. Validating optimizations of concurrent C/C++ programs. CGO'16

[FGP+16] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, P. Sewell. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. PLDI'16

[FSP+17] S. Flur, S. Sarkar, C. Pulte, K. Nienhuis, L. Maranget, K. E. Gray, A. Sezgin, M. Batty, P. Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. POPL'17

[JR16] A. Jeffrey, J. Riely. On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory. LICS'16

[LDGK08] G. Li, M. Delisi, G. Gopalakrishnan, and R. M. Kirby. Formal specification of the MPI-2.0 standard in TLA+. PPoPP'08

[KHL+17] A Promising Semantics for Relaxed-Memory Concurrency. J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, D. Dreyer. POPL'17

[McK11] P. E. McKenney. [patch rfc tip/core/rcu 0/28] preview of RCU changes for 3.3, November 2011. https://lkml.org/lkml/2011/11/2/363

[MOG+14] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. ICFP '14

[MPZN13] R. Morisset, P. Pawan, F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. PLDI'13

[OSP09] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. TPHOLS'09

[PPS16] J. Pichon-Pharabod and P. Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. POPL'16

[PSN16] A. Podkopaev, I. Sergey, and A. Nanevski. Operational aspects of C/C++ concurrency. CoRR'16.

[SMO+12] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. PLDI'12

[SSP+] S. Sarkar, P. Sewell, P. Pawan, L. Maranget, J. Alglave, D. Williams, F. Zappa Nardelli. The PPCMEM Web Tool. www.cl.cam.ac.uk/~pes20/ppcmem/

[WBDB15] J. Wickerson. M. Batty, B. Beckmann, A. Donaldson. Remote-Scope Promotion: Clarified, Rectified, and Verified. OOPSLA'15

[WBSC17] J. Wickerson, M. Batty, T. Sorensen, G. A. Constantinides. Automatically comparing memory consistency models. POPL'17