Relaxed Memory:  
The Specification Design Space  

Mark Batty  
University of Cambridge  

Fortran meeting, Delft, 25 June 2013
An ideal specification

Unambiguous

Easy to understand

Sound w.r.t. experimentally observable behaviour

Supportive of desirable programming idioms

Efficiently implementable now and in the future
An ideal specification

Unambiguous
Easy to understand
Sound w.r.t. experimentally observable behaviour
Supportive of desirable programming idioms
Efficiently implementable now and in the future

Testable?
How to define a memory model

Separate *instruction semantics* and the *memory model*

There are two prevailing styles of MM:

Operational models (x86, Power/ARM):
• Run stepwise through an execution
• Maintain a machine state (buffers, lists)

Axiomatic models (C/C++11, Java):
• Enumerate whole program executions
• The model decides which ones are allowed
X86 abstract machine

A very simple operational memory model

Nearly sequentially consistent
X86 abstract machine

Thread

Thread

Write Buffer

Write Buffer

Lock

Shared Memory
X86 abstract machine

<table>
<thead>
<tr>
<th>SB</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( x = 1 )</td>
<td>( y = 1 )</td>
</tr>
<tr>
<td>( r_1 = y ) //reads 0</td>
<td>( r_2 = x ) //reads 0</td>
<td></td>
</tr>
</tbody>
</table>
X86 abstract machine

<table>
<thead>
<tr>
<th>SB</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x=1</td>
<td>y=1</td>
</tr>
<tr>
<td></td>
<td>r1=y //reads 0</td>
<td>r2=x //reads 0</td>
</tr>
</tbody>
</table>

Test SB
Aside: a litmus test

<table>
<thead>
<tr>
<th>SB</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>x=1</td>
<td>y=1</td>
</tr>
<tr>
<td></td>
<td>r1=y //reads 0</td>
<td>r2=x //reads 0</td>
</tr>
</tbody>
</table>

A minimal example showcasing some relaxed behaviour

Can be used to compare very different models
(observable in 11G/167G runs on Power)

SB is one of the most tame relaxed behaviours
Restoring SC with Fences

SB+MFENCE

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=1</td>
<td>y=1</td>
</tr>
<tr>
<td>MFENCE</td>
<td>MFENCE</td>
</tr>
<tr>
<td>r1=y</td>
<td>r2=x</td>
</tr>
</tbody>
</table>

Cannot read 0,0
IBM Power and ARM

Power and ARM share their memory model

Much more relaxed than x86

The programmer can observe many relaxed behaviours
IBM Power is more relaxed

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
</table>
| \( x = 1 \) | \( \text{while (y==0) \{}; \) \| \( r = x \)   
| \( y = 1 \)   | \( \)  |

Observable behaviour: \( r = 0 \)

Message passing (MP)
IBM Power is more relaxed

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>while (y==0) { }</td>
</tr>
<tr>
<td>y = 1</td>
<td>r = x</td>
</tr>
</tbody>
</table>

Observable behaviour: r = 0

Forbidden on SC and x86-TSO
Allowed and observed on Power

1.7G/167G runs
IBM Power is more relaxed

Three possible reasons (at least) for $y==1$ and $x==0$

1. the two writes are performed in opposite order  
   reordering store buffers

2. the two reads are performed in opposite order  
   load reorder buffers / speculation

3. writes are propagated between threads out of order  
   interconnects partitioned by address (cache lines)
IBM Power is more relaxed

Three possible reasons (at least) for $y==1$ and $x==0$

1. the two writes are performed in opposite order
   reordering store buffers

2. the two reads are performed in opposite order
   load reorder buffers / speculation

3. writes are propagated between threads out of order
   interconnects partitioned by address (cache lines)
Power and ARM programming

Visible behaviour much weaker and subtle than x86.

Basically, program order is not preserved unless:

• writes to the same memory location (coherence)

• there is an address dependency between two loads
data-flow path through registers and arith/logical operations from the value of the first load to the address of the second

• there is an address or data or control dependency between a load and a store
  as above, or to the value store, or data flow to the test of an intermediate conditional branch

• you use a synchronisation instruction (ptesync, hwsync, lwsync, eieio, mbar, isync).
Programming language memory models

This is a harder problem. The model must:

• be ordered enough to program atop
• provide clear intuitions to programmers

• be efficiently implementable on all targets
• allow for compiler optimisations
Programming language memory models

This is a harder problem. The model must:

• be ordered enough to program atop
• provide clear intuitions to programmers

• be efficiently implementable on all targets
• allow for compiler optimisations

Strong vs Relaxed
Programming language memory models

This is a harder problem. The model must:

• be ordered enough to program atop
• provide clear intuitions to programmers
• be efficiently implementable on all targets
• allow for compiler optimisations

Strong vs Relaxed

Optimisations cause big problems in C/C++11!
C/C++11 concurrency

Formal model developed in discussion with WG21
An axiomatic data-race free model
Semantics defined by sets of execution graphs
More relaxed than Power
A simple program

```c
int r;
int x=0;
int y=0;
x = 1;  // r = y;
y = 1;  // r = x;
```
Memory model

The semantics of a program is a set of execution graphs, here is one:

```c
int r;
int x=0;
int y=0;
x = 1;  r = y;
y = 1;  r = x;
```

```
W x=0
W y=0
W x=1
W y=1
R y=0
R x=0
```
Memory model

The semantics of a program is a set of execution graphs, here is one:

```plaintext
int x=0;

int y=0;

x = 1; r = y;

y = 1; r = x;
```
Happens-before

Happens-before is partial.

```c
int r;
int x=0;
int y=0;
x = 1;  \parallel r = y;
y = 1;  \parallel r = x;
```
Oh no! A data race

hb unordered reads and writes are a race.

```c
int r;
int x=0;
int y=0;
x = 1; // r = y;
y = 1; // r = x;
```
Oh no! A data race

Racy programs have undefined behaviour.

```
int r;
int x=0;
int y=0;
x = 1;  r = y;
y = 1;  r = x;
```
Oh no! A data race

Racy programs have undefined behaviour.
Oh no! A data race

Racy programs have undefined behaviour.

The programmer is required to avoid races
Atomic accesses

Use atomic accesses to write racy code.

```c
int r;
atomic int x=0;
atomic int y=0;
store_{RLX}(&x,1);
r=load_{RLX}(&y);
store_{RLX}(&y,1);
r=load_{RLX}(&x);
```

Atomic accesses do not race.
Atomic accesses

Beware! Not sequentially consistent.

```c
int r;
atomic int x=0;
atomic int y=0;
store_RLX(&x, 1);
store_RLX(&y, 1);
r=load_RLX(&y);
r=load_RLX(&x);
```

![Diagram showing the atomic accesses and their order]

Wednesday, 17 July 13
Atomic accesses

Beware! Not sequentially consistent.

int r;
atomic int x=0;
atomic int y=0;
store_{RLX}(&x,1); r=load_{RLX}(&y);
store_{RLX}(&y,1); r=load_{RLX}(&x);

Allowed by Power, ARM, or compiler optimisations
Message passing

Use memory order annotations to synchronise.

```c
int r;
atomic int x=0;
atomic int y=0;
store_{RLX}(\&x, 1);
store_{REL}(\&y, 1);
r=load_{ACQ}(\&y);
r=load_{RLX}(\&x);
```

relaxed, release/acquire, seq_cst
Message passing

Use memory order annotations to synchronise.

```c
int r;
atomic int x=0;
atomic int y=0;
store_{RLX}(&x,1);  // r=load_{ACQ}(&y);
store_{REL}(&y,1);  // r=load_{RLX}(&x);
```

`rf` from `REL` to `ACQ` becomes an `hb` edge
Message passing

Use memory order annotations to synchronise.

```c
int r;
atomic int x=0;
atomic int y=0;
store_{RLX}(&x,1); r=load_{ACQ}(&y);
store_{REL}(&y,1); r=load_{RLX}(&x);
```

**Cannot read a stale write in** _hb_
Message passing

Use memory order annotations to synchronise.

```c
int r;
atomic int x=0;
atomic int y=0;
store_{RLX}(&x,1); \parallel r=load_{ACQ}(&y);
store_{REL}(&y,1); \parallel r=load_{RLX}(&x);
```

Cannot read a stale write in hb

Wednesday, 17 July 13
Use memory order annotations to synchronise.

```c
int r;
atOMIC int x=0;
atOMIC int y=0;
store_{RLX}(&x,1); | r=load_{ACQ}(&y);
store_{REL}(&y,1); | r=load_{RLX}(&x);
```
Coherence

Modification order is a per-location total order over atomic writes

Modification order must agree with happens before

Coherence is defined in terms of this order

<table>
<thead>
<tr>
<th>CoRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
</tr>
<tr>
<td>x=1</td>
</tr>
<tr>
<td>x=2</td>
</tr>
</tbody>
</table>

W x=1
W x=2
R x=2
R x=1

Wednesday, 17 July 13
Coherence

**Modification order** is a per-location total order over atomic writes

Modification order must agree with *happens before*

Coherence is defined in terms of this order

<table>
<thead>
<tr>
<th>CoRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
</tr>
<tr>
<td>x=1</td>
</tr>
<tr>
<td>x=2</td>
</tr>
</tbody>
</table>

W x=1
mo
W x=2
R x=2

R x=1
hb

Wednesday, 17 July 13
Coherence

Modification order is a per-location total order over atomic writes.

Modification order must agree with happens before.

Coherence is defined in terms of this order.

<table>
<thead>
<tr>
<th>CoRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
</tr>
<tr>
<td>x=1</td>
</tr>
<tr>
<td>x=2</td>
</tr>
</tbody>
</table>

Wednesday, 17 July 13
Coherence

**Modification order** is a per-location total order over atomic writes

Modification order must agree with **happens before**

Coherence is defined in terms of this order

---

<table>
<thead>
<tr>
<th>CoRR</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x=1</td>
<td>r1=x //reads 2</td>
</tr>
<tr>
<td></td>
<td>x=2</td>
<td>r2=x //reads 1</td>
</tr>
</tbody>
</table>

W x=1
W x=2

mo

R x=2
R x=1

hb
Coherence

All forbidden in C/C++11!

CoRR

\begin{align*}
W x=2 &\quad \rightarrow \quad R x=2 \\
W x=1 &\quad \neg\rightarrow \quad R x=1 \\
\text{mo} &\quad \uparrow \\
W x=1 &\quad \rightarrow \quad R x=2 \\
\text{rf} &\quad \rightarrow \\
W x=2 &\quad \rightarrow \quad R x=1
\end{align*}

CoWR

\begin{align*}
W x=2 &\quad \rightarrow \quad R x=1 \\
W x=1 &\quad \rightarrow \quad R x=1 \\
\text{hb} &\quad \rightarrow \\
W x=2 &\quad \rightarrow \quad R x=2
\end{align*}

CoWW

\begin{align*}
W x=2 &\quad \rightarrow \quad R x=1 \\
W x=1 &\quad \rightarrow \quad R x=1 \\
\text{mo} &\quad \rightarrow \\
W x=1 &\quad \rightarrow \quad R x=2 \\
\text{rf} &\quad \rightarrow \\
W x=2 &\quad \rightarrow \quad R x=1
\end{align*}

CoRW

\begin{align*}
W x=2 &\quad \rightarrow \quad R x=2 \\
W x=2 &\quad \rightarrow \quad R x=1 \\
\text{hb} &\quad \rightarrow \\
W x=2 &\quad \rightarrow \quad R x=1
\end{align*}
Atomicity

Modification order also defines atomicity of CAS-like features

A successful CAS produces a read-modify-write (RMW) access (A failed CAS is an atomic read. In Fortran?)

A RMW reads the maximal preceding write in \textit{mo}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Thread 0 & Thread 1 \\
\hline
x=1 & \\
x=2 & CAS (&x,2,3) \\
x=4 & \\
\hline
\end{tabular}
\end{table}
**Atomicity**

**Modification order** also defines atomicity of CAS-like features

A successful CAS produces a read-modify-write (RMW) access (A failed CAS is an atomic read. In Fortran?)

A RMW reads the maximal preceding write in \textit{mo}

**Table: RMW**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=1</td>
<td></td>
</tr>
<tr>
<td>x=2</td>
<td>CAS (&amp;x,2,3)</td>
</tr>
<tr>
<td>x=4</td>
<td></td>
</tr>
</tbody>
</table>

\textbf{mo} \downarrow

\textbf{W x}=1

\textbf{W x}=2

\textbf{CAS (\&x,2,3)}

\textbf{MO} \downarrow

\textbf{W x}=4

\textbf{W x}=2

\textbf{RMW x}=2,3
Atomicity

Modification order also defines atomicity of CAS-like features

A successful CAS produces a read-modify-write (RMW) access (A failed CAS is an atomic read. In Fortran?)

A RMW reads the maximal preceding write in mo

<table>
<thead>
<tr>
<th>RMW</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x=1</td>
<td>CAS (&amp;x,2,3)</td>
</tr>
<tr>
<td></td>
<td>x=2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x=4</td>
<td></td>
</tr>
</tbody>
</table>

Wednesday, 17 July 13
Operational vs Axiomatic models

**Operational models** (x86, Power/ARM):
- Can be similar to micro-architectural implementation
- Provide a decent local intuition
- Good for simulation
- Malleable
- Disguise incidental choices
- Awkward, ad-hoc structure

**Axiomatic models** (C/C++11):
- Mathematically simpler
- Choices are made explicitly
- Simulation is harder
- Intuition is not local, and no machine state
- Out-of-thin-air values an open problem
Operational vs Axiomatic models

Operational models (x86, Power/ARM):
• Can be similar to micro-architectural implementation
• Provide a decent local intuition
• Good for simulation
• Malleable
• Disguise incidental choices
• Awkward, ad-hoc structure

Axiomatic models (C/C++11):
• Mathematically simpler
• Choices are made explicitly
• Simulation is harder
• Intuition is not local, and no machine state
• Out-of-thin-air values an open problem
In C/C++11 out-of-thin-air is ill-defined

C11 abstracts relaxed behaviour introduced by processors and compiler optimisations

Out-of-thin-air values should be forbidden

At the moment this is poorly defined
Example 1: simple thin air

atomic int x=0, y=0;

r1=load_{RLX}(&x);
store_{RLX}(&y,r1);

r2=load_{RLX}(&y);
store_{RLX}(&x,r2);

Can r1 and r2 end up with the value 42?

The intent is to forbid this sort of thing.
Example 2: load buffering (LB)

...but Power and ARM allow load buffering, a similar behaviour without the dependencies,

Power, ARM and C11 allow 1,1 below: (observable on ARM)

```c
atomic int x=0, y=0;
r1 = load_{RLX}(&x);  ||  r2 = load_{RLX}(&y);
store_{RLX}(&y,1);  ||  store_{RLX}(&x,1);
```
Example 3: satisfaction cycles

atomic int x=0, y=0;
if (load_{RLX}(&x)==1) || if (load_{RLX}(&y)==1)
store_{RLX}(&y,1);        store_{RLX}(&x,1);

The C11 model ought to forbid this -- it does not

C11: “implementations should not allow such behaviour”
Forbid dependency cycles?

Define a new edge: dependency

Forbid cycles in dependency union rf

But optimisations remove some dependencies

The spec is then choosing which optimisations to allow
Example 4: dependency tracking?

atomic int x=0, y=0;

r1=load_{RLX}(&x); r3=load_{RLX}(&y);
f(&r1,&r2); f(&r3,&r4);
store_{RLX}(&y,r2); store_{RLX}(&x,r4);

Defining and tracking dependencies might be hard!

Optimisations in non-atomic code could break dependencies.

Would separate compilation provide a dependency summary?
Out-of-thin-air restriction: design space

Just allow this behaviour?

Disallow LB with a stronger model?

If neither, then defining dependency is key

Must allow sequential compiler optimisations in C/C++
Out-of-thin-air restriction: design space

Just allow this behaviour?

Disallow LB with a stronger model?

If neither, then defining dependency is key

Must allow sequential compiler optimisations in C/C++

We don’t know how to fix this in C/C++ right now!
Working towards a model for Fortran

What do implementations allow?

Which programming idioms to allow?
So... what do implementations allow?

Which litmus tests can be observed for atomics?

<table>
<thead>
<tr>
<th>SB</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=1</td>
<td>y=1</td>
<td></td>
</tr>
<tr>
<td>r1=y //reads 0</td>
<td>r2=x //reads 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MP</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=1</td>
<td></td>
<td>r1=y //reads 1</td>
</tr>
<tr>
<td>y=1</td>
<td>r2=x //reads 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LB</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1=x //reads 1</td>
<td>r2=y //reads 1</td>
<td></td>
</tr>
<tr>
<td>y=1</td>
<td>x=1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IRIW</th>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>y=1</td>
<td></td>
<td>x=1</td>
<td>r1=x //reads 1</td>
<td>r3=y //reads 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>r2=y //reads 0</td>
<td>r4=x //reads 0</td>
</tr>
</tbody>
</table>

There are many more!
With relaxed CAS in C/C++11

SB, MP and LB look the same, but are allowed:

**SB**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAS(&amp;x,1,2)</td>
<td>CAS(&amp;y,1,2)</td>
</tr>
<tr>
<td>r1= CAS(&amp;y,0,1) //reads 0</td>
<td>r2= CAS(&amp;x,0,1) //reads 0</td>
</tr>
</tbody>
</table>

**MP**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAS(&amp;x,1,2)</td>
<td>r1= CAS(&amp;y,1,2) //reads 1</td>
</tr>
<tr>
<td>CAS(&amp;y,0,1)</td>
<td>r2= CAS(&amp;x,0,1) //reads 0</td>
</tr>
</tbody>
</table>

**LB**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1= CAS(&amp;x,1,2) //reads 1</td>
<td>r2= CAS(&amp;y,1,2) //reads 1</td>
</tr>
<tr>
<td>CAS(&amp;y,0,1)</td>
<td>CAS(&amp;x,0,1)</td>
</tr>
</tbody>
</table>
I3-269: A model for Fortran?

Underlying targets use message passing

Optimisation very important

Use C/C++ without relaxed?
I3-269: A model for Fortran?

Underlying targets use message passing
Optimisation very important
Use C/C++ without relaxed?

This is “causal consistency”
All writes are releases, all reads acquires.
All reading produces happens-before.
13-269: A model for Fortran?

Underlying targets use message passing
Optimisation very important
Use C/C++ without relaxed?

This is “causal consistency”
All writes are releases, all reads acquires.
All reading produces happens-before.
13-269: A model for Fortran?

Underlying targets use message passing
Optimisation very important
Use C/C++ without relaxed?

This is “causal consistency”
All writes are releases, all reads acquires.
All reading produces happens-before.
I3-269: A model for Fortran?

Underlying targets use message passing
Optimisation very important
Use C/C++ without relaxed?

Cycles are forbidden in happens-before

This gets rid of load buffering
I3-269: A model for Fortran?

Underlying targets use message passing
Optimisation very important
Use C/C++ without relaxed?

Cycles are forbidden in happens-before

This gets rid of load buffering
I3-269: A model for Fortran?

Underlying targets use message passing
Optimisation very important
Use C/C++ without relaxed?

Cycles are forbidden in happens-before

This gets rid of load buffering
I3-269: A model for Fortran?

Underlying targets use message passing
Optimisation very important
Use C/C++ without relaxed?

Strange CAS behaviour goes away

<table>
<thead>
<tr>
<th>SB</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAS(&amp;x,1,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1 = CAS(&amp;y,0,1) //reads 1</td>
<td>r1 = CAS(&amp;x,0,1) //reads 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MP</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAS(&amp;x,1,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAS(&amp;y,0,1)</td>
<td>r1 = CAS(&amp;x,1,2) //reads 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r2 = CAS(&amp;x,0,1) //reads 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MP</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = CAS(&amp;x,1,2) //reads 1</td>
<td>r1 = CAS(&amp;y,1,2) //reads 1</td>
<td></td>
</tr>
<tr>
<td>CAS(&amp;y,0,1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
13-269: A model for Fortran?

Underlying targets use message passing
Optimisation very important
Use C/C++ without relaxed?

Thin air behaviour
goes away

```c
atomic int x=0, y=0;
    r1=load_{RLX}(&x);
    r2=load_{RLX}(&y);
store_{RLX}(&y,r1);
store_{RLX}(&x,r2);
```
13-269: A model for Fortran?

Underlying targets use message passing
Optimisation very important
Use C/C++ without relaxed?

```c
// sender
x = ...
y.store(1, release);
```

```c
// receiver
while (0 == y.load(acquire));
r = x;
```
13-269: A model for Fortran?
Underlying targets use message passing
Optimisation very important
Use C/C++ without relaxed?

This comes for free on x86.

On Power/ARM this costs something:

Each atomic write needs an lwsync
Reads need an isync, and false dependencies
RMW’s need both
I3-269: A model for Fortran?

Underlying targets use message passing
Optimisation very important
Use C/C++ without relaxed?

```cpp
// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(acquire));
r = x;
```

It would forbid breaking-out of synchronisation:

```cpp
// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```
Which litmus tests can be observed?

Which programming idioms should be supported?

Can the target machine be abstracted?

Which compiler optimisations should be allowed?

What behaviours do intended implementations have?