Modeling of Architectures

Choose your own adventure in herding cats

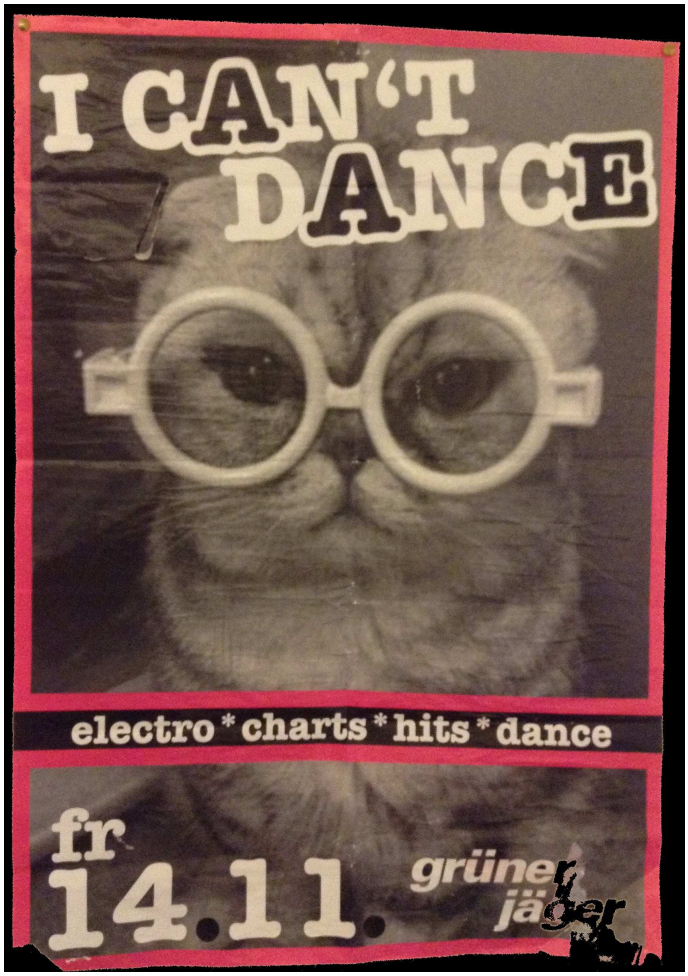
Jade Alglave

Microsoft Research Cambridge and University College London

Abstract

Concurrent programming is known to be quite hard. It is made even harder by the fact that, very often, the execution models of the machines we run our software on are not precisely defined.

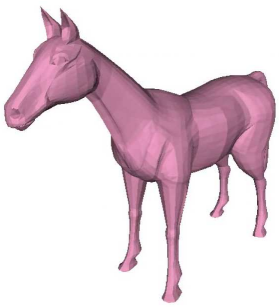
This document is a tutorial on the herd tool and the `cat` language, in which one can define consistency models.



1 Why herd cats anyway?

Concurrent programming is known to be quite hard.

Look at this pink pony:

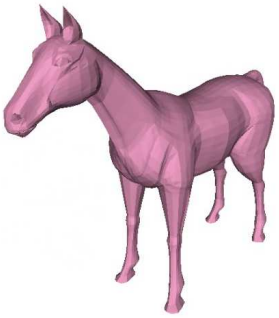


It's been computed on an Intel x86 CPU, with sequential code. It's healthy!

Now look at this one:

It's been computed on an Nvidia GPU, with concurrent code. It's all broken; this is because multicore machines sometimes do not behave quite as we'd expect, and sensible looking code can end up producing unexpected results.

Now, there are ways to repair the concurrent pony, look; this pony was also computed on an Nvidia GPU, under the same conditions as the previous one:



One can repair our pony by placing special instructions in the code used to compute it. However the semantics of such special instructions is very often poorly documented.

In this lecture we present a language, called `cat`, that can help us tackle this issue by allowing us to precisely describe the semantics of these special instructions. More precisely, our `cat` language gives us means to describe *consistency models*, i.e. execution models of concurrent and distributed systems.

By the end of this lecture, you should have been able to build several models:

- Sequential Consistency [8] (see Sec. 3)
- Total Store Order [10] (see Sec. 3),
- a model similar in spirit to IBM Power or ARM [5] (see Sec. 4),
- a model similar in spirit to Nvidia GPUs [3] (see Sec. 5),
- a model inspired by C++ [14] (see Sec. 6).

Reading notes Most of the lecture is going to be interactive, using the herd tool. For this tutorial to go smoothly, I would suggest to:

- have internet access, to be able to go to the herd web interface: virginia.cs.ucl.ac.uk/herd-web/?book=tutorial;
- open the appendix, where I give the final models that the tutorial should help you build; these can act as guidelines along the way.

2 First steps in herding cats

Here we're going to learn about the essential concepts to follow this lecture.

Litmus tests are small snippets of assembly or pseudo-assembly code that allow us to examine the behaviour of a chip (see e.g. [5]), or a model like we're doing here.

Below is our first litmus test. In this test, called `MP` (short for *message passing*), two processors `P0` and `P1` communicate via two shared memory locations `x` and `y`, both initialised to 0:

```
Bell MP
{
x = 0;
y = 0;
}
 P0       | P1         ;
 w[] x 1  | r[] r1 y   ;
 w[] y 1  | r[] r2 x   ;
exists (1:r1 = 1 /\ 1:r2 = 0)
```

On the left, the thread `P0` writes 1 to memory location `x`, and 1 to memory location `y`. On the right, the thread `P1` reads from `y` and places the result into register `r1`, and reads from `x` and places the result into register `r2`. The registers `r1` and `r2` are private to `P1`.

Essentially, `P0` writes a message in `x`, then sets up a flag in `y`, so that when `P1` sees the flag (via its read from `y`), it can read the message in `x`.

At the bottom of the test, we ask "is there an execution of this test such that register `r1` contains the value 1 and register `r2` contains the value 0?".

Exercise: what do you think? Do you think such an execution is possible?

The herd tool Now, let's try it out! The herd tool lets us simulate a model, and run litmus tests against that model, to determine which executions are allowed by this model. So let's go to virginia.cs.ucl.ac.uk/herd-web/?book=tutorial,. In the "litmus test" box, find the file `mp.litmus` in the drop box. Let's select "all executions" for the display, and then click on the pink pony.

Histograms In the "histogram" box, we see the following result:

```
Test MP Allowed
States 4
1:r1=0; 1:r2=0;
1:r1=0; 1:r2=1;
1:r1=1; 1:r2=0;
1:r1=1; 1:r2=1;
Ok
```

Note that we see the result `1:r1=1; 1:r2=0;`, which we asked about in our test. Thus this result is reachable by our test, and this is why the tool says `Ok`; otherwise it would say `No`.

Executions In the "executions" box, we see that this test can have four different executions. In all of them we see four *events*: writes to `x` or `y`, of the shape `W() x=32`, which means that value 32 was written to the memory location `x`, and reads from `x` or `y`, of the shape `R() y=52`, meaning that memory location `y` was read, and the value read was 52. We also see two types of *relations* over these events: po arrows and rf arrows.

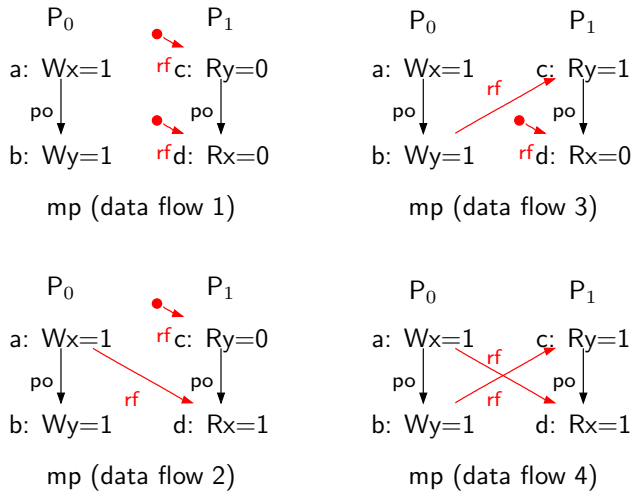


Figure 1: Four executions of the `MP` test

Program order The po arrows represent the *program order*. The program order intuitively corresponds to the order in which the events occur on a thread.

For example in our test, on P0, the write of x is appears in program order before the write of y. Therefore the two corresponding events in Fig. 1 are related by po. Similarly the read of y on P1 appears in our test in program order before the read of x; thus the two corresponding events in Fig. 1 are related by po.

Note also that po is transitive. In Fig. 1, suppose there was an extra event e occuring on P1 with a po arrow from d to e. In this case there would also be a po arrow from c to e. That is, e is after c in the program order.

Read-from The rf (*read-from*) arrows depict who reads from where; more precisely, for each read of a certain location, it finds a unique write of the same location and value.

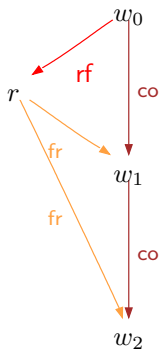
Let's go through each execution one by one. In the first one, the two reads on P1 read from the initial state, which we depict with rf arrows with no sources. Our test has initialised x and y to 0, thus the read events in the first execution have the value 0. In the second execution, the read of x reads from the update of x by P0, hence has the value 1; however the read of y reads from the initial state still, hence takes the value 0. The third execution is the one our test was asking about: the read of y reads from the update by P0, whereas the read of x takes its value from the initial state, i.e. 0. In the last execution, both reads take their values from the updates by P0, i.e they both read the value 1.

Now we're going to write a model that will forbid the result `1:r1=1; 1:r2=0;`, which we asked about in our test. It won't be a minimal model forbidding this result though (by which I mean that the model will forbid much more behaviours than this particular one), but that's okay.

To do so, we need to learn a few more concepts.

Coherence order In most models I know of, there's a notion of a *coherence order*. Intuitively it's a history of all writes to a given memory location x, that represents the order in which writes to x hit x. If you were sitting in memory location x, looking at writes falling down onto you, and recording their values as they come by, you would get the coherence order. In this lecture, the coherence order co is a total order over writes to a given memory location.

From-read Using the read-from and coherence relations, we can build a relation called *from-read*:



Intuitively, a from-read arrow starts from a read of a given location `x`, for example the read r just above, and points to all the writes that overwrite the value this read has taken. In the drawing above, the read r takes its value from the write w_0, as shown by the rf arrow between them. The write w_0 is then overwritten by the write w_1, as shown by the co arrow between them. Hence there is an fr arrow between r and w_1, as w_1 overwrites the write w_0 from which r reads. The write w_1 is then overwritten by the write w_2, as shown by the co arrow between them. Thus there is an fr arrow between r and w_2 too.

A good place to build fr is our `cat` file: go to the "model" box, and find the "toggle cat" button. The interface should put you in front of the `cat` file called `tutorial.cat`. If you click on "make custom cat", you should be able to edit it, and write the line above underneath the title and preamble of the cat file, i.e. underneath:

```
"I can't dance"

include "tutorial.cat"
```

In cat speak, this is how we can build the from-read fr (you can put this line in your `cat` file):

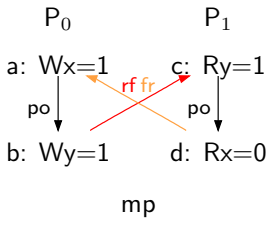
```
let fr = rf^-1;co
```

which means that two events a and b are related by `fr` if it's possible to follow an `rf` arrow backwards from a to some event, then a `co` arrow forwards from there, arriving at b. Also let's add

```
show fr
```

to the `cat` file, to make sure that the tool displays the new fr arrows.

Now let's run herd on `MP` again (don't forget to click on the pink pony!). In the third execution, observe the fr arrow between the read of `x` by `P1` and the update of `x` by `P0`:



Let's look at this execution a bit more: the read of `y` by `P1` reads the update by `P0`, whereas the read of `x` by `P1` reads the initial state, i.e. ignores the update of `x` by `P0`. Note that this is an execution that leads to the result `1:r1=1; 1:r2=0;` that we asked about in our litmus test.

Also, note that such an execution compromises the implementation of message passing idioms: intuitively, we'd like `P1` to access the update of the data `x` after it has read the update of the flag `y`.

In our models, these kind of violations will occur as cycles. Notice, in the execution above, how we can follow the arrows to find a cycle: from a to b to c to e and back to a.

3 Let's herd two kittens [12]

There are several models that would forbid this compromising execution. We'll build two: Sequential Consistency (SC) [8] and Total Store Order (TSO), which is the model of Sparc TSO [11] and Intel x86 [10]. In fact we won't build exactly TSO below, only a fragment of it, but you'll get to build TSO properly in an exercise later on.

Sequential Consistency is such that executions of a program are interleavings of the instructions appearing on the different threads of the program. One can show (see e.g. [1, 2]) that this corresponds to an *axiomatic* model, i.e. model phrased in terms of events, program order, read-from, coherence and from-read (such as the ones we are manipulating here), where the program order and what I call the *communication relations* (i.e. the union of read-from, coherence and from-read) are compatible, i.e. there cannot be any cycle in their union. Let's first build the communication relations in cat speak (to put in your `cat` file):

```
let com = rf | co | fr
```

Here we're gathering the read-from `rf`, coherence order `co` and the from-read `fr` in a relation called `com` (for communications). In `cat` speak, the symbol `|` is the union of arrows, i.e. of relations over events.

Now we can build a *procedure* that will implement Sequential Consistency (to put in your `cat` file):

```
procedure sc() =
  let sc-order = (po | com)+
  acyclic sc-order
end
```

This procedure we call `sc`, and its enforces the acyclicity (through the keyword `acyclic`) of the relation `sc-order`. This relation is defined as the union of the program order `po` and the communication relation `com` that we built earlier. We need to call this procedure later on in the `cat` file for it to have an effect:

```
call sc()
```

Now let's run our `MP` example under that new `cat` file. Let's have a look at the histogram:

```
Test MP Allowed
States 3
1:r1=0; 1:r2=0;
1:r1=0; 1:r2=1;
1:r1=1; 1:r2=1;
No
```

Observe that there are now only 3 possible states, and the compromising result `1:r1=1; 1:r2=0;` has disappeared. If you look at the executions, you'll see that we do not have the compromising one anymore. That's because there was a cycle in the union of the program order `po` and the communicaton `com`: from a to b to c to d and back to a.

Total Store Order is such that write-read pairs (whether to the same memory location or not) on the same thread can appear to occur out of order. This may be due to processor caches and buffering (which lead to faster processors, but surprising behaviours).

Consider the following *store buffering* litmus test, where two processors `P0` and `P1` communicate via two shared memory locations `x` and `y`, both initialised to 0:

```
Bell SB
{
x = 0;
y = 0;
}
 P0       | P1        ;
 w[] x 1  | w[] y 1   ;
 r[] r1 y | r[] r2 x  ;
exists (0:r1 = 0 /\ 1:r2 = 0)
```

On the left, the thread `P0` writes `1` to memory location `x`, and reads from `y` and places the result into register `r1`. Symmetrically, the thread `P1` writes `1` to memory location `y`, and reads from `x` and places the result into register `r2`. The register `r1` is private to `P0`, and `r2` is private to `P1`.

At the bottom of the test, we ask the question "is there an execution of this test such that both registers contain the value 0?"

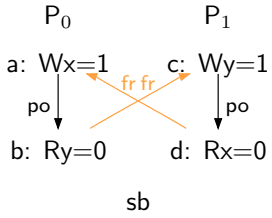
Let's comment out the call to SC because we're trying to build a different model here:

```
(* call sc() *)
```

Let's feed this test to herd; find the file `sb.litmus` in the litmus test drop box. After clicking on the pink pony, it says:

```
Test SB Allowed
States 4
0:r1=0; 1:r2=0;
0:r1=0; 1:r2=1;
0:r1=1; 1:r2=0;
0:r1=1; 1:r2=1;
Ok
```

Note that we see the result `0:r1=0; 1:r2=0;`, which we asked about in our test. This result corresponds to the following execution:



sb

More precisely, the two reads on each thread reads from the initial state (e.g. the read on `P1` reads from the initial state for `x`). Since the initial state is overwritten by the updates to `x` and `y`, we are exactly in the `fr` situation: for example the read of `x` on `P1` is `fr`-before the write of `x` on `P0`.

Such a result can be explained for example by the presence of *store buffers*, one per thread: both writes (the write to `x` on `P0` and the write to `y` on `P1`) can sit in their threads' store buffers for a while, then the reads (from `y` on `P0` and from `x` on `P1`) can take their value from the initial state, i.e. from memory, and finally the writes can hit the memory.

This would be the case on an Intel x86 [10], or a Sparc TSO machine [11].

One can show (see e.g. [10, 1, 2]) that TSO corresponds to an axiomatic model (i.e. phrased in terms of events, program order, read-from, coherence and from-read), where a fragment of the program order and the communication relations are compatible, i.e. there cannot be any cycle in their union. Remember that we've built the communication relations earlier, and placed the definition in our `cat` file.

To model TSO, we also need to build a notion of read-froms between different threads, which we often call *external read-from* (in our `cat` file, next to the definition of `com` for example):

```
let rfe = rf & ext
```

where the primitive `ext` is the relation that gathers pairs of events that belong to different threads, such as the write of `y` by `P0` and the read of `y` by `P1`.

Now, one way to model the store buffering scenarios allowed by TSO is to reorder write-read pairs (whether to the same memory location or not). This means that TSO differs from SC on write-read pairs. Thus we have to exclude all the write-read pairs from `sc-order` to build TSO. We can then require the acyclicity of this new relation `tso-order`. Let's build this as a procedure once again (to put in your `cat` file, just below the `sc` procedure):

```
procedure almost-tso() =
  let ppo = po \ W*R
  let tso-order = ppo | rfe | co | fr
  acyclic tso-order
end
```

Here we're declaring a procedure called `almost-tso`, in which there is a local definition of a relation called `ppo` (for preserved program order). We define `ppo` as the program order `po`, minus the write-read pairs: in cat speak the "setminus" operation is \; the (predefined) set of write events is `W`, the (predefined) set of read events is `R`, thus the set of all write-read pairs is `W*R`.

On the second line of this procedure, we require the acyclicity of the union (remember that the symbol `|` is the union in cat speak) of `ppo` and the communications `com`.

We need to call our `almost-tso` procedure in our `cat` file:

```
call almost-tso()
```

If you run the test now, you should find that the test has still not been forbidden. Now, most architectures provide special instructions called *fences* that prevent certain reorderings. For example, Intel x86 provides `mfence`, which

prevents the reordering of write-read pairs allowed by TSO. Let's build ourselves such a fence; the right place to do so is in the `bell` file. Go to the "model" box, and find the "toggle bell" button; if you click on it, it should put you in front of `tutorial.bell`. Clicking on "make custom bell" should allow you to edit the `bell` file. Do not remove the title line `"I can't dance"`; you could write another title, but not remove the title entirely.

First we declare a possible *annotation* for our fence, for example `'wr` (for write-read):

```
enum Fences = 'wr
```

More precisely, here we declare an enumeration of possible fence annotations under the name `Fences`. For now we only have one annotation, `wr`. This enum will also create a set `Wr` all of the events that bear the annotation `wr`.

Interlude: Enums, tags, and annotations More precisely, in cat speak, one can define enumeration types, as follows. This should go into a `bell` file; if you want to try it out I would suggest opening a fresh `bell` file to not spoil the one we're currently building; don't forget to give a title to your file [16]:

```
"Hey Hey Mama"
enum Led = 'z || 'e || 'p
```

Here we're defining an enumeration type `Led`, which contains three *tags*: `'z`, `'e` and `'p`. The user can then use these tags to specify that certains events can bear eponymous *annotations*; for example (again in the `bell` file):

```
events W[{'z}]
events R[{'e,'p}]
```

specifies that write events (which belong to the predefined set `W`) can bear the annotation `z`, whilst read events (which belong to the predefined set `R`) can bear the annotations `e` or `p`.

The user can then use these annotations in litmus tests, for example (if you want to try it out, find `ledzep.litmus` in the litmus test drop box):

```
Bell BlackDog
{
x = 0;
y = 0;
}
P0         | P1         ;
w[z] x 1   | r[p] r2 x  ;
r[e] r1 y  |            ;
exists(0:r1=0 /\ 1:r2=1)
```

Internally, herd has built one set for each possible tag that was declared in the enumeration type `Led`: the set `Z` gathers all events with annotation `z`, the set `E` all events with annotation `e`, and the set `P` all events with annotation `p`. Thus in the litmus test `LedZep` above, the write event on `P0` will belong to the set `Z`, the read on `P0` to the set `E`, and the read on `P1` to the set `P`.

The user can manipulate these sets in the `bell` file; for example:

```
let ze = Z*E
let ep = E*P
let zep = ze;ep
```

defines three relations `ze`, `ep`, and `zep`, such that `ze` gathers all pairs where the right extremity belongs to `Z` and the left extremity belongs to `E`, `ep` gathers all pairs where the right extremity belongs to `E` and the left extremity belongs to `P`, and `zep` builds the sequence of a step of `ze` and a step of `ep`. We can add

```
show ze, ep, zep
```

if we want to visualise these three relations.

Back to the write-read fence Then we need to say that our fence events (much like read and write events, but to represent fences) can bear the annotations that we've just defined:

```
events F[Fences]
```

More precisely, here we say that our fence events, of the shape `f(...)` can bear the annotation `wr`. Let's go ahead and modify the `SB` example by adding fences to it:

```
Bell SB+fwr+fwr
{
x = 0;
y = 0;
}
 P0       | P1        ;
 w[] x 1  | w[] y 1   ;
 f[wr]    | f[wr]     ;
 r[] r1 y | r[] r2 x  ;
exists (0:r1 = 0 /\ 1:r2 = 0)
```

As you can see, there is a fence instruction `f(wr)` between each write-read pair on `P0` and `P1`. Thus the two corresponding events belong to the set `Wr`.

Now we need to build a *relation* to gather all pairs of memory events (read or write) that are separated by a fence in between them in program order; the standard library has such a primitive, it is called `fencerel`. We can put the following lines in our `bell` file for example (where `F & Wr` is the set of fence events that bear the annotation `wr`):

```
let fwr = fencerel(F & Wr)
show fwr
```

The definition of `fencerel` is in herd's standard library; it is as follows:

```
let fencerel(S) = (po & (_ * S)); po
```

It takes as argument a set `S` of events, and builds the sequence (the symbol `;` designates the sequence of relations in cat speak) of the relation `po & (_ * S)` and the program order `po`. Now let's look at the relation `po & (_ * S)` a bit more. It is built as the intersection (`&` is the intersection in cat speak) of the program order `po`, and the set of pairs `(_ * S)`. These pairs are such that the

domain (the left extremity) can be anything, whether read or write (_ means "anything" in cat speak), and the range (the right extremity) is in the set `S`.

So now when we define `fwr` as `fencerel(F & Wr)` up above, this means that we've built the relation gathering all pairs of events (read or write), such that there is a fence event (i.e. that belong to `F`) in between them in program order, and this fence event bears the annotation `wr` (i.e. belongs to the set `Wr`). This is true, for example, of the write-read pair on `P0` in our example `SB+fwr+fwr`.

Now we can add this `fwr` relation to our `tso` definition; that is, we can update our `almost-tso` procedure as follows:

```
procedure almost-tso() =
  let ppo = po \ W*R
  let tso-order = ppo | fwr | rfe | co | fr
  acyclic tso-order
end
```

Note how we've added the `fwr` relation to the shape of cycles forbidden by our `tso` definition, so that now we can have no cycle that is made of `ppo`, `fwr` or `com` arrows.

Let's feed this test to herd and see what it says (find the file `sb+fwr+fwr.litmus` in the litmus test drop box):

```
Test SB+fwr+fwr Allowed
States 3
0:r1=0; 1:r2=1;
0:r1=1; 1:r2=0;
0:r1=1; 1:r2=1;
No
```

Observe that the final state we were asking about, that can be explained by the store buffer scenario outlined above, has disappeared, thanks to the fences.

Think of saving your `bell` and `cat` files, for example under the names `kittens.bell` and `kittens.cat`.

4 Let's herd our first big cat: a tiger [6]

Today we'll learn how to build a model that is similar in spirit to IBM Power and ARM. These two models revolve around a handful of principles that we'll build one by one.

Start from fresh `bell` and `cat` files, but keep the definitions of `fr`, `rfe`, and `com`. We're going to use the `sc` procedure in a slightly different way, to `flag` the executions that do not satisfy SC; in our `kittens.cat` file we had:

```
procedure sc() =
  let sc-order = (po | com)+
  acyclic sc-order
end

[...]

call sc()
```

to forbid non-SC executions, i.e. executions with a cycle in the union of the program order `po` and the communications `com`.

Now we don't want to forbid SC executions, but just `flag` them. Thus we implement a procedure to flag non-SC executions (in our `cat` file):

```
procedure sc-flag() =
  let sc-order = (po | com)+
  flag ~acyclic sc-order as non-sc
end

call sc-flag()
```

that is we flag, with the name `non-sc`, all the executions where there is a cycle in the union of the program order `po` and the communications `com`. Note that `~` is the negation. Finally, select the "positive executions" display option.

4.1 SC per location

The first principle is called SC PER LOCATION. It means that if you analyse your program through the prism of a sole memory location at a time, everything looks as if on SC. Formally, this also means that non-relational analyses are sound for free under for consistency models that respect SC PER LOCATION [4].

Now, let's look at a bunch of litmus tests.

coWW

```
Bell coWW
{
x = 0;
}
 P0       ;
 w[] x 1  ;
 w[] x 2  ;
exists (x=1)
```

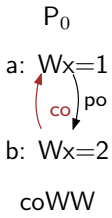
In the `coWW` test, we have only one thread `P0`, which does two writes of memory location `x` in program order. The first write in program order writes the value 1 to `x`; the second writes 2. We're asking at the end if it is possible to have the value 1 in `x` at the end, which means that the write of value 2 has hit the memory before the write of value 1.

Recall that we have defined the coherence order precisely for that purpose: describing the order in which writes to a given memory location hit that location.

Now let's run `herd` on our `cat` file and the test `coWW` (find `coww.litmus` in the litmus test drop box). We get the following histogram:

```
Test coWW Allowed
States 2
x=1;
x=2;
Ok
```

On the execution side, we get the execution that leads to the final state we've asked about in our litmus test; note that it's a non-SC execution (see the line `Flag non-sc` in the histogram above) because of its cycle in the union of `po` and `com`:



If you select the "all executions"option, you'll see two executions; one where the coherence order co follows the program order po, and the one above, where the coherence order co is in the opposite direction as po.

coRW1

```
Bell coRW1
{
x = 0;
}
 P0       ;
 r[] r1 x ;
 w[] x 1  ;
exists (0:r1=1)
```

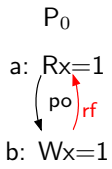
In the `coRW1` test, we have only one thread `P0`, which does a read and a write of memory location `x` in program order. The read access reads `x` and places the result into register `r1`. The write access writes the value 1 to `x`. We're asking at the end if it is possible to have the value 1 in `r1` at the end, which means that the read takes its value from the po-later write.

Recall that we have defined the read-from precisely for that purpose: describing who reads from where; in this case if the read of `x` can take its value from the po-later write of value 1.

Now let's run herd on our `cat` file and the test `coRW1` (find the file `corw1.litmus` in the litmus test dropbox). We get the following histogram:

```
Test coRW1 Allowed
States 2
0:r1=0;
0:r1=1;
Ok
```

On the execution side, we get the execution that leads to the final state we've asked about in our litmus test; note that it's a non-SC execution (see the `Flag non-sc` line just above) because of its cycle in the union of `po` and `com`:



coRW1

If we select the "all executions" display option, we get two executions; one where the read takes its value from the initial state, and the one above, where the read takes its value from the po-later write of value 1.

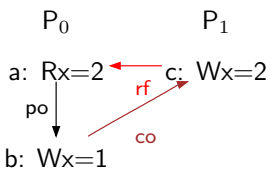
coRW2

```
Bell coRW2
{
x = 0;
}
 P0       | P1      ;
 r[] r1 x | w[] x 2 ;
 w[] x 1  |         ;
exists (0:r1=2 /\ x=2)
```

In the `coRW2` test, we have two threads `P0` and `P1`, communicating via the shared memory location `x` which is initialised to 0. `P0` is the same as in the previous test `coRW1`, i.e. does a read and a write of memory location `x` in program order. The read access reads `x` and places the result into register `r1`. The write access writes the value 1 to `x`. `P1` writes the value 2 into memory location `x`. We're asking at the end if it is possible to have the value 2 in `r1` and in `x` at the end, which means that the read takes its value from the write of `x` on `P1` (`r1=2`) and that the write by `P1` hits the memory after the write by `P0` (`x=2`).

Now let's run herd on our `cat` file and the test `coRW2` (find `corw2.litmus` in the litmus test dropbox).

On the execution side, we get the following execution; note that it's a non-SC execution because of its cycle in the union of `po` and `com`:



coRW2

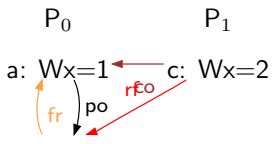
If we select the "all executions" display option, we get six executions, the third one being the one leading to the result we're asking about in the test. In this execution, the read of `x` by `P0` takes its value from the write by `P1` (note the read-from rf arrow between them), and the write by `P0` hits the memory before the write by `P1` (note the coherence arrow co between them).

coWR

```
Bell coWR
{
x = 0;
}
 P0       | P1      ;
 w[] x 1  | w[] x 2 ;
 r[] r1 x |         ;
exists (0:r1=2 /\ x=1)
```

In the `coWR` test, we have two threads `P0` and `P1`, communicating via the shared memory location `x` which is initialised to 0. `P0` writes 1 into memory location `x`, and reads `x`, placing the result into register `r1`. `P1` writes 1 into `x`. At the end we're asking if it's possible to have the value 2 into `r1`, i.e. the read by `P0` reads from the write by `P1`, and to have the value 1 in `x`, i.e. the write of `P0` hits the memory after the write of `P1`.

Now let's run herd on our `cat` file and the test `coWR` (file `cowr.litmus`). On the execution side, we get the following non-SC execution:



coWR

If we select the "all executions" option, we get six executions, the fourth one being the one leading to the result we're asking about in the test. In this execution, the read of `x` by `P0` takes its value from the write by `P1` (note the read-from rf arrow between them), and the write by `P0` hits the memory after the write by `P1` (note the coherence arrow co between them).

Recall that we have defined the from-read precisely for that purpose: starting from a read such as the one by `P0`, pointing to all the writes (such as the one by `P0`) that overwrite the value read (given by the write on `P1`). Thus we have a from-read arrow fr between the read by `P0` and the po-preceding write by `P0`.

coRR

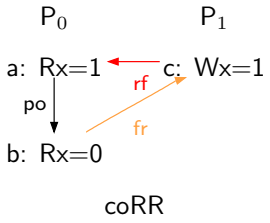
```
Bell coRR
{
x = 0;
}
 P0       | P1      ;
 r[] r1 x | w[] x 1 ;
 r[] r2 x |         ;
exists (0:r1=1 /\ 0:r2=0)
```

In the `coRR` test, we have two threads `P0` and `P1`, communicating via the shared memory location `x` which is initialised to 0. `P0` reads `x` twice, placing the

results into `r1` and `r2`; `P1` writes 1 to `x`. At the end we're asking if it's possible for `r1` to hold the value 1, i.e. after having read from `P1`, and for `r2` to hold the value 0, i.e. after having read from the initial state.

Now let's run herd on our `cat` file and the test `coRR` (file `corr.litmus`).

On the execution side, we get the following execution; note that it's a non-SC execution because of its cycle in program order and communications:



If we select the "all executions" option, we get four executions, the third one being the one leading to the result we're asking about in the test. In this execution, the first read by `P0` reads from the write on `P1` (note the read-from arrow rf between them), and the second read by `P0` reads from the initial state.

Recall that we have defined the from-read precisely for that purpose: starting from a read such as the second one by `P0`, pointing to all the writes (such as the one by `P1`) that overwrite the value read (given by the initial write of `x`). Thus we have a from-read arrow fr between the read by `P0` and the write by `P1`.

Forbidding these idioms Let's look at all the executions that we have flagged to be non-SC: observe that they all have a similar shape, in which the program order contradicts the communication relations. More precisely, because all of our tests use one memory location only, it's the program order restricted to the same location that contradicts the communication relations.

Let's define a new notion `po-loc`, i.e. the program order restricted to both extremities having the same location; in cat speak (a good place to put this definition would be in the `cat` file, just next to the definition of `com`):

```
let po-loc = po & loc
```

where the primitive `loc` gathers all pairs of read and write events that have the same location.

Now, let's require for `po-loc` to not contradict our communication relations (recall we've built this notion before, into the relation `com`), within a procedure `sc-per-location`:

```
procedure sc-per-location() =
  acyclic po-loc | com
end
```

Don't forget to call the procedure `sc-per-location` in your `cat` file:

```
call sc-per-location()
```

Now let's re-run all of our tests, and observe that we do not have the executions that we've flagged anymore!

Exercise: SC per location with load-load hazard Certain architectures, such as Sparc RMO [11] allow what is sometimes called *load-load hazard*, i.e. a situation where the `coRR` test that we've just seen is allowed to yield the result `0:r1=1; 0:r2=0;`.

How do you think we can build a check that forbids all tests `coWW`, `coRW1`, `coRW2` and `coWR`, but allows the test `coRR`?

4.2 No thin air

The second principle is called NO THIN AIR. Intuitively, this principle forbids scenarios where a read can take its value from a write that *depends* on this read. The word "depends" can be interpreted in many different ways; let's make that precise. Consider the following litmus test:

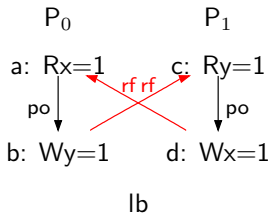
```
Bell LB
{
x = 0;
y = 0;
}
 P0       | P1       ;
 r[] r1 x | r[] r2 y ;
 w[] y 1  | w[] x 1  ;
exists (0:r1 = 1 /\ 1:r2 = 1)
```

In the `LB` test, we have two threads `P0` and `P1`. `P0` reads `x` and places the result into register `r1`, then writes 1 to `y`. `P1` reads `y` and places the result into register `r2`, then writes 1 to `x`. At the end we're asking whether it is possible for both registers to contain the value 1, i.e. if the two reads can read from the po-later writes. This is perfectly well possible on ARM or Nvidia machines for example [5, 3], because the read-write pairs on each thread can be reordered.

Let's run herd on this test with our current `cat` file (find the `lb.litmus` file in the litmus test drop box); we get the following histogram:

```
Test LB Allowed
States 4
0:r1=0; 1:r2=0;
0:r1=0; 1:r2=1;
0:r1=1; 1:r2=0;
0:r1=1; 1:r2=1;
Ok
```

The execution corresponding to the situation we asked about in the test is as follows; note that it's a non-SC execution because of its cycle in union of program order and communications:



lb

Now, we can place something between the read and the write on each thread to prevent their reordering: we can use *dependencies*, typically *address, data*, or *control* dependencies. Note that dependency relations always start with a read.

Interlude: Implementing dependencies In this lecture, we'll abstract away from actual ways of implementing dependencies. But to give an idea of what I mean, here's an example of data dependency. Consider the following variant of `LB`:

```
Bell LB+datas
{
x = 0;
y = 0;
}
P0            | P1            ;
r[] r1 x      | r[] r1 y      ;
xor r2 r1 r1  | xor r2 r1 r1  ;
add r3 r2 1   | add r3 r2 1   ;
w[] y r3      | w[] x r3      ;
exists (0:r1 = 1 /\ 1:r2 = 1)
```

Observe how we use operations on registers between the read and write on each thread. More precisely on `P0`, we read location `x` and place the result into register `r1`. Then we `xor` the value in `r1` with itself, and place the result into register `r2` (of course the result is always `0`, but that's okay). Then we `add` 1 to the value in `r2`, and place the result (i.e. 1) into `r3`. Finally we write the value in `r3` into location `y`. This manipulation of registers is enough to implement a data dependency from the read of `x` to the write of `y`.

Abstracting dependencies For this lecture however, we're going to model dependencies as fences. This means that the dependencies that we are manipulating here are stronger than in the wild. However the definitions and axioms we're defining should hold with a proper notion of dependencies.

Let's do it! Let's open our `bell` file, and create a tag `dep` for dependencies; we can for example declare a `Fences` type:

```
enum Fences = 'dep
```

Now in our litmus test we can use dependencies (note the `f[dep]` instructions in between the read and the write on each thread):

```
Bell LB+dep+dep
{
x = 0;
y = 0;
}
 P0        | P1        ;
 r[] r1 y  | r[] r2 x  ;
 f[dep]    | f[dep]    ;
 w[] x 1   | w[] y 1   ;

exists (0:r1 = 1 /\ 1:r2 = 1)
```

Run it on your current `cat` file and observe that there are non-SC executions.

Finally we have to give a semantics to our dependencies. The right place to do this is your `cat` file. Let's look at the execution of the `LB` litmus test that we give above. We can see a cycle in the union of the program order `po` and the read-froms `rf` between threads. We want to build a check such that having dependency arrows instead of program order arrows forbids this execution.

So we need to implement a few concepts in cat speak; let's build the relations yielded by our special dependency events; recall that `herd` has a standard library in which there is a function called `fencerel`. This function builds, given a set of events (e.g. `F`), the relation gathering all pairs of events in program order that have such an event (e.g. `f(dep)`) between them. Thus using `fencerel` we can build the relation corresponding to dependencies (to put in your `bell` file):

```
let deps = fencerel(F & Dep) & (R * _)
show deps
```

Note how we restrict our dependency relation `deps` to pairs of events that start with a read event (which belongs to the predefined set `R`). Also, we add `show deps` to make sure that `herd` will display this new relation.

Now let's go back to the execution we want to forbid. By the look of it, we want to build a check such that a cycle in the union of the dependencies `deps` and the external read-froms `rfe` is forbidden. Let's call this union *happens-before* and write it `hb` for example. In cat speak (to put in your `cat` file):

```
let hb = (deps | rfe)+
```

Note that we use the transitive closure `()+` to make `hb` a transitive relation. We don't really need to for now, especially since we're going to require `hb` to be acyclic in a minute, but it's going to be important later.

Now let's require, in our `cat` file, for the happens-before relation to be acyclic to forbid the execution above; in cat speak:

```
procedure no-thin-air() =
  acyclic hb
end
```

Don't forget to call this procedure in your `cat` file:

```
call no-thin-air()
```

Now run `lb+dep+dep.litmus` on your current `cat` file and observe that there is no non-SC execution anymore.

Exercise: Implementing address dependencies How do you think we can implement address dependencies? By this I mean that I'd like to see a sequence of instructions which, placed between two reads, implement e.g. an address dependency.

4.3 Propagation

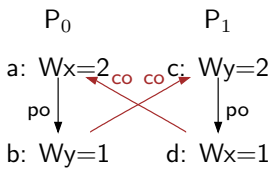
The third principle is called PROPAGATION. Intuitively, this principle gives the semantics of *fences*, which are special instructions that ensure that two writes

of distinct locations separated by a fence have to propagate to other threads in the order in which they appear in the program.

One representative example of this principle is the following:

```
Bell 2+2w
{
x = 0;
y = 0;
}
 P0       | P1      ;
 w[] x 2  | w[] y 2 ;
 w[] y 1  | w[] x 1 ;
exists (x=2 /\ y=2)
```

The `2+2w` test has two threads `P0` and `P1` which both write to the shared memory locations `x` and `y`. `P0` writes 2 to `x` and 1 to `y`, when `P2` writes 2 to `y` and 1 to `x`. In the end we're asking if it's possible to have both locations holding the value 2, which could be explained by the write-write pairs being reordered on both threads. This corresponds to the following execution, which is non-SC:



2+2w

Try running herd on `2+2w.litmus` under our current `cat` file and observe that there are (flagged) non-SC executions.

Now, placing a fence between each write-write pair should forbid this cycle. Let's do it! Let's open our `bell` file, and update our fences. Remember that we had defined an annotation `'dep` for them previously:

```
enum Fences = 'dep
```

Here we additionally give ourselves *lightweight fences* `'lw` (there will be heavyweight ones later on):

```
enum Fences = 'dep || 'lw
```

Now, placing a lightweight fence between each write-write pair of the `2+2w` test (leading to `2+2w+lwfs`) should forbid this cycle.

To do so, we first need to define a relation `flw`, that contains all the possible pairs of events in program order separated by a lightweight fence; recall that we can use the `fencerel` definition. A good place to do so is our `bell` file:

```
let flw = fencerel(F & Lw)
show flw
```

Let's also create a `fences` relation to gather all actual fences (as opposed to dependencies); for now we only have lightweight fences, but we'll have heavyweight fences in a moment:

```
let fences = flw
```

Now in our `cat` file we can define the *propagation order* as the order induced by lightweight fences:

```
let prop = fences
```

In cat speak, we can now forbid the `2+2w` cycle by the following procedure (to put in your `cat` file):

```
procedure propagation() =
  acyclic prop | co
end
```

And don't forget to call the procedure (in your `cat` file):

```
call propagation()
```

Try running `herd` on `2+2w.litmus` on this new `cat` file and observe that there is no non-SC execution anymore.

4.4 Observation

The fourth principle is called OBSERVATION. Intuitively, this principle means that two reads of distinct locations have to read writes in the order in which these writes propagate. One representative example of this principle is the message passing example we've seen at the beginning:

```
Bell MP
{
x = 0;
y = 0;
}
 P0           | P1            ;
 w() [x], 1   | r() r1, [y]   ;
 w() [y], 1   | r() r2, [x]   ;
exists (1:r1 = 1 /\ 1:r2 = 0)
```

As we've seen before, the test as is can yield the result where `P1` sees the new flag (`r1=1`), but reads the stale data (`r2=0`). There are architectural several reasons for this to happen, amongst which:

1. the two reads on `P1` could be reordered—this could happen for example on ARM, IBM Power, or Nvidia machines [5, 3];
2. the two writes on `P0` could be reordered—this could happen for example on ARM, IBM Power, or Nvidia machines [5, 3];
3. the two writes could swap places on their way to the reading thread `P1`, the write of `y` by `P0` hitting `P1` before the write of `x` does—this could happen for example on ARM or IBM Power machines [5].

We need several devices to protect against each of these items:

1. to protect against the reordering of reads on `P1`, one typically uses dependencies such as the ones we've defined to deal with `LB`;
2. to protect against the writes being reordered on their thread or on their way to the reading thread, we need fences.

Now in our litmus test we can use for example:

1. a data dependency to prevent the reordering of reads on `P1` (note the `f(dep)` events in between the reads in the litmus test below);
2. a lightweight fence to prevent the write scenarios we mentioned earlier (note the `f(lw)` between the writes in the litmus test below).

This corresponds to the following test:

```
Bell MP+lw+dep
{
x = 0;
y = 0;
}
 P0       | P1        ;
 w[] x 1  | r[] r1 y  ;
 f[lw]    | f[dep]    ;
 w[] y 1  | r[] r2 x  ;
exists (1:r1 = 1 /\ 1:r2 = 0)
```

Finally we have to define our OBSERVATION principle. To do so, let's first go back to the `LB` litmus test: to forbid the non-SC execution, we chose earlier to use dependencies between the read and write on each thread. Equally we could have chosen to place a fence on each thread, like so:

```
Bell LB+lws
{
x = 0;
y = 0;
}
 P0       | P1        ;
 r[] r1 x | r[] r2 y  ;
 f[lw]    | f[lw]     ;
 w[] y 1  | w[] x 1   ;
exists (0:r1 = 1 /\ 1:r2 = 1)
```

or to mix and match fences and dependencies:

```
Bell LB+dep+lw
{
x = 0;
y = 0;
}
 P0       | P1       ;
 r[] r1 x | r[] r2 y ;
 f[dep]   | f[lw]    ;
 w[] y 1  | w[] x 1  ;
exists (0:r1 = 1 /\ 1:r2 = 1)
```

This means that, in our `cat` file, we can extend our happens-before relation `hb` to include our fences, as follows; `hb` was:

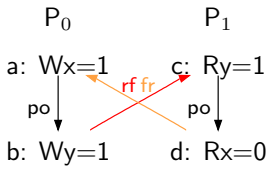
```
let hb = (deps | rfe)+
```

and now becomes:

```
let hb = (deps | fences | rfe)+
```

The `no-thin-air` and `observation` checks can remain unchanged, and now we forbid tests like `LB+lws` or `LB+dep+lw` above.

Now, let's look at the execution of the `MP` litmus test that we want to forbid (note that it is non-SC):



mp

Intuitively, we want to make sure that if the two writes a and b by `P0` are separated by a lightweight fence, they cannot be reordered, and propagate to the reading thread `P1` in the same order as they appear on `P0`. Moreover we want to ensure that the two reads c and d by `P1` cannot be reordered when separated by a dependency.

Reading off the drawing very plainly, we want to forbid executions where we take one step via a lightweight fence, then one step of read-from, then one step of dependency, then one step of from-read and end up where we started.

Now, observe that there is a `hb` path from the write a by `P0` to the read d by `P1`: $(a, b) \in$ `fences`, $(b, c) \in$ `rf` and $(c, d) \in$ `deps`. It is, however, a special path in `hb`, as its first step consists of a fence step. This special subset of `hb` is in fact the propagation order. Remember that we had defined it in our `cat` file, as follows:

```
let prop = fences
```

which is not quite right anymore, in the light of the `MP` example. So let's refine our propagation order as follows (to put in your `cat` file):

```
let prop = fences;hb*
```

Thus the propagation order `prop` starts off with a step of fence, then can continue with a happens-before chain of any length, through external read-froms, dependencies, and other fences. This is what is sometimes called *B-cumulativity* of a fence [5]: when the propagation order induced by a fence carries over to chains of happens-before.

We can now phrase our OBSERVATION axiom in terms of the propagation order (to put in your `cat` file):

```
procedure observation() =
  irreflexive fre;prop
end
```

where `fre` is the external from-read, i.e. a from-read arrow between two events that belong to different threads. In cat speak, we can define it using the predefined `ext` relation that gathers pairs of events that belong to different threads, such as the read d by `P1` and the write a by `P0`. A good place to put the definition of `fre` is in your `cat` file, for example next to the definition of `rfe`:

```
let fre = fr & ext
```

Don't forget to call the procedure `observation` at the end of your `cat` file:

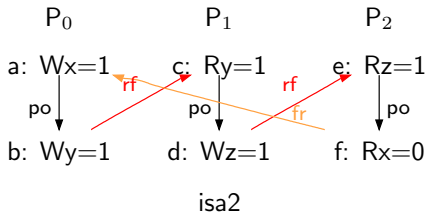
```
call observation()
```

Running this new `cat` file on the tests `mp+lw+dep.litmus`, `lb+lws.litmus` and `lb+dep+lw.litmus` should forbid their non-SC executions.

Observe that the OBSERVATION principle also forbids distributed variants of the message passing example, such as `ISA2+lw+dep+dep`:

```
Bell ISA2+lw+dep+dep
{
x = 0;
y = 0;
}
 P0       | P1       | P2       ;
 w[] x 1  | r[] r1 y | r[] r2 z ;
 f[lw]    | f[dep]   | f[dep]   ;
 w[] y 1  | w[] z 1  | r[] r3 x ;
exists (1:r1 = 1 /\ 2:r2 = 1 /\ 2:r3=0)
```

The test `ISA2+lw+dep+dep` is similar to the message passing one (`MP`), in that we want to ensure that the two writes by the first thread `P0` propagate in the order in which they've been written, to forbid the scenario where, even if the flag `y` has been passed over to `P1`, i.e. `r1=1`, the read of `x` on `P2` reads from the initial state instead of from the update of `x` by `P0`. This corresponds to the following non-SC execution:



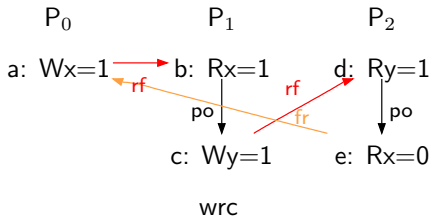
isa2

The difference with `MP` is that the propagation is over several threads: here the write of `y` by `P0` propagates to `P1`, whereas the write of `x` by `P0` propagates to `P2`. Because the threads `P1` and `P2` communicate (via `z`), and because the accesses on both `P1` and `P2` are ordered via dependencies, we have a happens-before chain from the write of `y` by `P0` to the read of `x` by `P2`. This is enough to create a propagation order arrow from the write of `x` by `P0` to the read of `x` by `P2`, which therefore contradicts the execution where the read of `x` would read from the initial state.

The OBSERVATION principle should also forbid `WRC+lw+dep`:

```
Bell WRC+lw+dep
{
x = 0;
y = 0;
}
 P0       | P1        | P2         ;
 w[] x 1  | r[] r1 x  | r[] r2 y   ;
          | f[lw]     | f[dep]     ;
          | w[] y 1   | r[] r3 x   ;
exists (1:r1=1 /\ 2:r2=1 /\ 2:r3=0)
```

The difference with `MP` is that the two writes that we want to ensure propagate in the right order are on different threads: the write of the data `x` is made by `P0`, whereas the write of the flag `y` is made by `P1`. We want to ensure that if the reading thread `P2` takes the flag (so that `r2=1`), then the update of the data `x` has reached `P2`, so that the read of `x` on `P2` cannot read from the initial state anymore. This would correspond to the following non-SC execution:



Our definition of propagation order does not forbid this yet; we have to add a new notion to our `cat` file, namely *A-cumulativity*. We say that a fence is A-cumulative when it orders two writes on different threads `P0` and `P1` (just like the write a of `x` by `P0` and the write c of `y` by `P1`), such that `P1` reads the write by `P0` then does the fence, then does the second write. Looking at the execution above, it means that an A-cumulative fence placed between b and c should create an arrow between a and c. In cat speak:

```
let A-cumul = rfe;fences
```

Now let's update our propagation order to include A-cumulativity (to put in your `cat` file):

```
let prop = (fences | A-cumul);hb*
```

Observe that `WRC+lw+dep` is now forbidden.

Exercise: Distributed `2+2w` Consider the following litmus test, which is essentially a distributed variant of `2+2w`:

```
Bell w+rw+ww
{
x = 0;
y = 0;
}
 P0       | P1       | P2       ;
 w[] x 2  | r[] r1 x | w[] y 2  ;
          | w[] y 1  | w[] x 1  ;
exists (1:r1=2 /\ x=2 /\ y=2)
```

Which fences should we use to forbid the final state? Why? Where should we put them?

4.5 Restoring SC with heavyweight fences

The fifth principle explains how to regain SC. To do so, we need to use *heavyweight fences*. We can extend our fences as follows (in our `bell` file):

```
enum Fences = 'dep || 'lw || 'hw
```

Now consider the store buffering litmus test that we've seen earlier:

```
Bell SB
{
x = 0;
y = 0;
}
 P0       | P1       ;
 w[] x 1  | w[] y 1  ;
 r[] r1 y | r[] r2 x ;
exists (0:r1 = 0 /\ 1:r2 = 0)
```

Putting a heavyweight fence between the write-read pairs on each thread should forbid the scenario where both reads take their value from the initial state. More generally, putting a heavyweight fence between any pair of events in program order should restore SC.

Recall that SC can be defined as the acyclicity of the union of program order `po` and the communication relations `com`. Thus to restore SC with heavyweight fences, we shouldn't allow any cycle in the union of the relation `fhw`, induced by the heavyweight fences, and the communications.

Let's define `fhw` in our `bell` file:

```
let fhw = fencerel(F & Hw)
show fhw
```

And in our `cat` file, let's implement our fifth principle:

```
procedure restoring-sc() =
  acyclic fhw | com
end

call restoring-sc()
```

Observe that `SB+hws` is now forbidden; note however that `SB+lws` is still allowed, as one really needs heavyweight fences to restore SC. The lightweight fences only contribute to building the propagation order.

Now, heavyweight fences should also forbid the non-SC executions of `LB` and `MP`. This means that we should add them to our `fences` relation, so that they naturally get included in the definitions of `hb` and `prop`, thus contribute to the NO-THIN-AIR and OBSERVATION checks. In our `cat` file, we had:

```
let fences = flw
```

and now we should have:

```
let fences = flw | fhw
```

Think of saving these `bell` and `cat` files, for example under the names `tiger.bell` and `tiger.cat`.

Exercise: Independent Reads of Independent Writes Consider the following litmus test, known as `IRIW`:

```
Bell IRIW
{
x = 0;
y = 0;
}
 P0       | P1       | P2      | P3       ;
 w[] x 1  | r[] r1 x | w[] y 1 | r[] r3 y ;
          | r[] r2 y |         | r[] r4 x ;
exists (1:r1 = 1 /\ 1:r2 = 0 /\ 3:r3=1 /\ 3:r4=0)
```

Which fences should we use to forbid the final state? Why? Where should we put them?

Exercise: SC and TSO Re-Make Re-Model [9] Remember that we've defined SC and (not quite) TSO earlier in our `cat` file, as follows:

```
procedure sc() =
  let sc-order = (po | com)+
  acyclic sc-order
end

procedure almost-tso() =
  let ppo = po \ W*R
  let tso-order = ppo | rfe | co | fr
  acyclic tso-order
end
```

Try to reformulate both models in terms of the five principles we've just learnt: SC PER LOCATION, NO THIN AIR, PROPAGATION, OBSERVATION and RESTORING SC. The SC model you'll end up with will be equivalent to the one above. The TSO model you'll end up with will be TSO proper!

5 Let's herd our second big cat: a jaguar [13]

Today we'll learn how to build a model that is similar in spirit to Nvidia GPUs. This model differs from the previous one because GPUs have *scopes*. Start with fresh `bell` and `cat` files (keep the title and include).

Intuitively, a scope is a set of threads. Here we'll consider three different scopes: `cta`, `gpu` and `system`.

5.1 Scopes

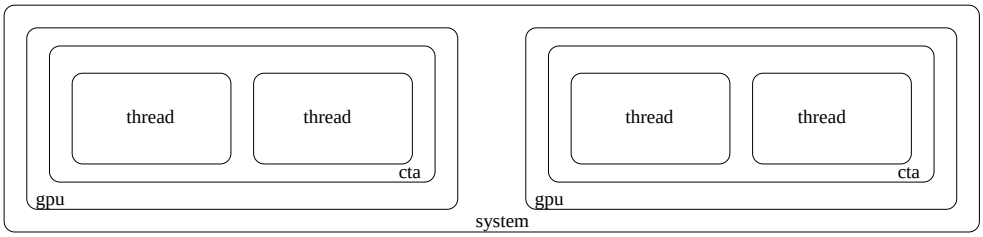


Figure 2: Concurrency hierarchy

Scopes are organised hierarchically: `cta` is *narrower* than `gpu`, and `gpu` is narrower than `system`, as shown in Fig. 2.

Building the concurrency hierarchy To build this concurrency hierarchy, the ideal place is our `bell` file; we can simply enumerate these three scopes as follows:

```
enum Scopes = 'cta || 'gpu || 'system
```

We also need to specify the hierarchy, with a function called `narrower` (to put in your `bell` file):

```
let narrower(s) = match s with
  || 'system -> 'gpu
  || 'gpu -> 'cta
end
```

The dual function is called `wider` (to put in your `bell` file):

```
let wider(s) = match s with
  || 'gpu -> 'system
  || 'cta -> 'gpu
end
```

Scope tree Litmus tests also need changing; in particular we need to say which scope a given thread belongs to. Let's go back to the message passing example:

```
Bell MP
{
x = 0;
y = 0;
}
 P0       | P1        ;
 w[] x 1  | r[] r1 y  ;
 w[] y 1  | r[] r2 x  ;
exists (1:r1 = 1 /\ 1:r2 = 0)
```

Let's implement a variant where `P0` and `P1` are each on a different `cta`, but in the same `gpu`, hence in the same `system`:

```
Bell MP-mit-scopes
{
x = 0;
y = 0;
}
 P0       | P1        ;
 w[] x 1  | r[] r1 y  ;
 w[] y 1  | r[] r2 x  ;
scopes: (system (gpu (cta P0) (cta P1)))

exists (1:r1 = 1 /\ 1:r2 = 0)
```

Note the addition of the *scope tree* `scopes: (system (gpu (cta P0) (cta P1)))` that specifies where the threads `P0` and `P1` are.

Scope annotations Now, we need to say that our instructions can bear these scope annotations, indicating at which level of the concurrency hierarchy they can operate.

In this model, we'd like for our fences to have different effects depending on which scope they apply to. To implement that, we can add scope annotations to our fences events (that belong to the predefined set `F`). The right place to do so is in our `bell` file:

```
events F[Scopes]
```

We will need to be able to say, given an instruction or an event of a litmus test, which scope it belongs to. There are two different notions of scoping: *syntactic* and *execution* scopes (sometimes called *static* and *dynamic* [7, 3]).

Syntactic scope The herd tool provides the primitive `tag2events` which, given a tag such as the scope ones `'cta`, `'gpu` or `'system`, returns all the events that bear this tag. Thus `tag2events('gpu)` returns the set of all events annotated with `'gpu`.

Execution scope The herd tool also provides the primitive `tag2scope` which, given a tag, returns the relation that links events that are executed within the same scope level as `tag`. Consider our `MP` example, and let's build `tag2scope('cta)`:

for this particular instance of `MP`, it will contain eight pairs. The first pair gathers the two writes on `P0`, because the scope tree `(system (gpu (cta P0) (cta P1)))` specifies that `P0` is in its own `cta`. Similarly, the second pair will gather the two reads on `P1`. The set `tag2scope('cta)` also contains the symmetric pairs to the ones we've just studied, that leads to 4 pairs, to which it adds the four identity pairs, where each event is paired with itself. Note that not all pairs are shown in diagrams, as most relations undergo a transitivity removal procedure before being printed.

Building `tag2scope('gpu)` will gather all the possible pairs of events, because the scope tree specifies that `P0` and `P1` belong to the same `gpu`; idem for `tag2scope('system)`.

5.2 RMO per scope

So now, experimentally Nvidia GPUs implement *RMO per scope* [3]. Let's study RMO first, then the scopes.

Relaxed Memory Order, or RMO, is a Sparc model that allows the reordering of any pair of read or write events in program order [11, 1, 2]. One can restore these orderings using dependencies or fences. We've defined the dependencies `deps` earlier. We have defined fences as well, but for today we'll define our fences locally.

Let's write an `rmo` procedure in our `cat` file (note that you need the definitions of `rfe` and `fr`):

```
procedure rmo() =
  let rmo-fences = fencerel(F)
  let rmo-order = deps | rmo-fences | rfe | co | fr
  acyclic rmo-order
end
```

What we do here is the following: we define a relation `fence` using our `fencerel` primitive applied to the set of fence events `F`. Then we define a relation that we call `rmo-order`, which is the union of the dependencies `deps`, the fence relation `rmo-fences`, the external read-from `rfe`, the coherence order `co`, and the from-read `fr`

One can show (see e.g. [1, 2]) that requiring the acyclicity of this relation `rmo-order` is enough to implement RMO.

Exercise: difference between RMO and Power or ARM What's a test that distinguishes RMO from Power or ARM (as we've defined them previously)? More precisely, what's a test that's forbidden on RMO but allowed on Power or ARM?

Scope hierarchy Now, we need to express the fact that each level of our scope hierarchy (`cta`, `gpu`, `system`) will behave like RMO. To do so, we can modify our `rmo` procedure to take scopes into account (to put in your `cat` file):

```
procedure rmo() =
  let rmo-fences(t) = fencerel(F & tag2events(t))
  let rmo-order(t) = (fence(t) | rfe | co | fr) & tag2scope(t)
  forall t in Scopes do
    acyclic rmo-order(t)
  end
end
```

By contrast to our scope-less `rmo` procedure, here our local relation `fence` takes a scope annotation `t` as an argument, and builds the relation induced by fences (which belong to `F`) that bear the syntactic annotation `t`. Then the relation `rmo-order` uses this scoped fence relation; additionally, we impose that the extremities of the pairs gathered in `rmo-order` belong to the same scope instance of level `t`. Finally we require the acyclicity of the relation `rmo-order` for each level of the concurrency hierarchy, i.e. for each `t` in the set `Scopes`.

Don't forget to call the procedure `rmo` from your cat file:

```
call rmo()
```

Now try to run `mp-mit-scopes.litmus` and observe that it is allowed; try again with the following test:

```
Bell MP-mit-scopes+fgpus
{
x = 0;
y = 0;
}
 P0       | P1        ;
 w[] x 1  | r[] r1 y  ;
 f[gpu]   | f[gpu]    ;
 w[] y 1  | r[] r2 x  ;
scopes: (system (gpu (cta P0) (cta P1)))

exists (1:r1 = 1 /\ 1:r2 = 0)
```

and observe that it is forbidden.

Now think of saving your `bell` and `cat` files, for example under the names `jaguar.bell` and `jaguar.cat`.

Exercise: Implementing scope inclusion How would you implement an RMO per scope model as above, but where the fences have an effect not only at their scopes, but also at narrower scopes? That is, the fence for `system` also has an effect at `gpu` and `cta` level for example? This procedure should forbid `mp-mit-scopes+fgpu+fsys`:

```
Bell MP-mit-scopes+fgpu+fsys
{
x = 0;
y = 0;
}
 P0       | P1        ;
 w[] x 1  | r[] r1 y  ;
 f[gpu]   | f[sys]    ;
 w[] y 1  | r[] r2 x  ;
scopes: (system (gpu (cta P0) (cta P1)))

exists (1:r1 = 1 /\ 1:r2 = 0)
```

but not `mp-mit-scopes+fcta+fgpu`:

```
Bell MP-mit-scopes+fcta+fgpu
{
x = 0;
y = 0;
}
 P0       | P1        ;
 w[] x 1  | r[] r1 y  ;
 f[gpu]   | f[sys]    ;
 w[] y 1  | r[] r2 x  ;
scopes: (system (gpu (cta P0) (cta P1)))

exists (1:r1 = 1 /\ 1:r2 = 0)
```

6 Let's herd our third big cat: a panther [15]

Today we'll learn how to build a model that is inspired by C++. Start with fresh `bell` and `cat` files (keep the title and include).

This model is different from the previous ones, in particular because it doesn't simply reject executions based on the presence of certain cycles. It also looks for *data races*, and declares an execution that has a data race to be *undefined*.

6.1 Plain and special events

Let's first build our `bell` file. We're going to have two different flavours of events: *plain* ones and *special* ones:

```
enum Flavours = 'plain || 'special

events R[Flavours]
events W[Flavours]
```

6.2 Release-acquire semantics

Now let's focus on our special events. The model we're building is such that synchronisation happens through special events, more precisely, when two threads

communicate (i.e. one writes to a location that is read by the other).

Thus the read-from relation over special events is quite central to this model; let's add this notion to our `cat` file, as a relation `special-rf` (for `rf` over special events):

```
let special-rf = rfe & (Special * Special)
```

Now we need to implement the notion that when two threads communicate in an atomic way, they synchronise. Let's build a *happens before* relation `hb`:

```
let hb = (po | special-rf)+
```

Here we say that an event e_1 happens before another one e_2 (i.e. $(e_1, e2) \in$ `hb`) when e_1 is in program order before e_2, or e_2 reads from e_1 and they're both special, or any chain of such steps (note how we use the transitive closure).

Note that we didn't say anything specific about fences in our `bell` file; that's because fences won't play much of a role in this model so that we can focus on the special accesses instead. Therefore today in our happens before relation, we're taking all of the program order where we had previously taken only dependencies and fences. On the other hand we're only using the special read-from, whereas previously we used all of `rf`.

Now to make our happens-before relation an order that we can build on, we should implement a NO THIN AIR check. We can put the following procedure in our `cat` file:

```
procedure no-thin-air() =
  acyclic hb
end
```

Now using this happens-before relation we can forbid message passing scenarios from going wrong; recall `MP`:

```
Bell MP
{
x = 0;
y = 0;
}
 P0       | P1        ;
 w[] x 1  | r[] r1 y  ;
 w[] y 1  | r[] r2 x  ;
exists (1:r1 = 1 /\ 1:r2 = 0)
```

For this test to make sense in our current setup, where reads and writes can be plain or special, we need to annotate our events as being `plain`

```
Bell MP-plain
{
x = 0;
y = 0;
}
 P0            | P1             ;
 w[plain] x 1  | r[plain] r1 y  ;
 w[plain] y 1  | r[plain] r2 x  ;
exists (1:r1 = 1 /\ 1:r2 = 0)
```

If we make the communication over the flag `y` special, like so:

```
Bell MP-special
{
x = 0;
y = 0;
}
 P0              | P1              ;
 w[plain] x 1    | r[special] r1 y ;
 w[special] y 1  | r[plain] r2 x   ;
exists (1:r1 = 1 /\ 1:r2 = 0)
```

then we create a happens-before order from the update of the data `x` (i.e. the write of `x` on `P0`) and the read of `x` on `P1`.

Now, to forbid the final state of this variant of `MP`, we need to build an OBSERVATION check (to put in your `cat` file):

```
procedure observation() =
  irreflexive fre;hb
end
```

6.3 Validity

Now let's gather our checks into a single procedure (to put in your `cat` file). Note that we added a call to `sc-per-location` `cat` files, just because:

```
procedure valid() =
  call sc-per-location()
  call no-thin-air()
  call observation()
end
```

Of course for this to work you need to add the definition of the `sc-per-location` procedure, which you can copy from your previous `cat` files.

Don't forget to call your `valid` procedure (you need `fre` as before):

```
call valid()
```

Now try to run `mp-plain.litmus` under your new `cat` file and observe it is allowed; try it out on `mp-special.litmus` and observe it is forbidden.

6.4 Data races and undefined executions

Now we want to be able to distinguish executions that have data races, and flag them as being undefined. Let's define data races (in our `cat` file):

```
let at-least-one k = (k * _ | _ * k)
let conflict = at-least-one(W) & loc & ext
let race =
  let r = conflict & ~(hb | hb^-1)
  in r \ ((I * M) | (M * I) | (Special * Special))
show race
```

Thus we define a race as a pair of accesses that:

- `conflict` (which implies that the accesses are distinct, as they must be on different threads when they conflict), and
- are not ordered by `hb` or `hb^-1` (i.e. take a step of `hb` backwards), and
- not one of them is an initialisation write, and
- are not both special.

A conflict is pair of accesses, such that at least one is a write (i.e. belongs to `W`), both accesses are relative to the same memory location (i.e. they belong to `loc`), and they belong to different threads (i.e. they belong to `ext`).

Now we can flag racy executions as undefined:

```
procedure race-free() =
  flag ~empty race as undefined
end
```

and define our executions to be both valid and race-free:

```
procedure execution() =
  call valid()
  call race-free()
end
```

Don't forget to call this procedure in your `cat` file:

```
call execution()
```

and to comment out our previous standalone call to the `valid` procedure because it appears in our `execution` procedure now.

Now try out our new `cat` file on `MP-special`, and observe that the tool finds it is racy. To fix this issue, we can modify our example like so (`beq` is a branch instruction; below it branches to `END` if `r1` is equal to 0):

```
Bell MP-special+branch
{
x = 0;
y = 0;
}
 P0              | P1              ;
 w[plain] x 1    | r[special] r1 y ;
                 | beq r1, 0, END  ;
 w[special] y 1  | r[plain] r2 x   ;
                 | END:            ;
exists (1:r1 = 1 /\ 1:r2 = 0)
```

This is because the branch on `P1` here ensures that the read of `x` on `P1` takes its value only after the flag `y` has been read by `P1`. The fact that you need a branch also shows `po` is a dynamic notion, otherwise `hb = (po | rf-special)+` would catch the test without the branch, and not find it racy.

Exercise: Release Sequence In C++ there's a notion of *release sequence*, which says that synchronisation does not just happen via `special-rf`, i.e. from the special write of a special read-from to the corresponding special read. Rather, it can happen from any write of the same location and on the same thread that precedes (in program order) a special write.

How would you modify our model to include this notion?

7 Credits

The `cat` language is mostly Luc Maranget's and my work, the five principles as well [5]. The shiny web interface is thanks to Tyler Sorensen. The `bell` subset has benefited from Tyler's contribution. Tyler has computed the three pink ponies at the beginning of this document.

For more ponies: https://youtu.be/3-Y8xLsqywY.

For more `bell` and `cat` files: virginia.cs.ucl.ac.uk/herd-web. In particular:

- IBM Power: virginia.cs.ucl.ac.uk/herd-web/?book=herding-cats&language=ppc&cat=ppc
- ARM: virginia.cs.ucl.ac.uk/herd-web/?book=herding-cats&language=arm&cat=arm
- C++: virginia.cs.ucl.ac.uk/herd-web/?book=c11popl15
- Nvidia PTX: virginia.cs.ucl.ac.uk/herd-web/?book=ptx&language=ptx&cat=ptx.

With thanks to the supersonic beta-testers: Patrick Cousot, Matthew Hague, Luc Maranget, Tyler Sorensen, Michael Tautschnig and Jules Villard.

References

[1] Jade Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris 7, 2010.

[2] Jade Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2):178–210, 2012.

[3] 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 *ASPLOS 2015*.

[4] Jade Alglave, Daniel Kroening, John Lugton, Vincent Nimal, and Michael Tautschnig. Soundness of data flow analyses for weak memory models. In *APLAS*, pages 272–288. Springer, 2011.

[5] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. *TOPLAS*, 36(2).

[6] The Beatles. The Continuing Story of Bungalow Bill. In *White Album*, 1968.

[7] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous-race-free memory models. In *ASPLOS 14*.

[8] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[9] Roxy Music. Re-Make Re-Model. In *Roxy Music*, 1972.

[10] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, pages 391–407. Springer, 2009.

[11] SPARC International Inc. *The SPARC Architecture Manual Version 9*, 1994.

[12] Sparks. Here Kitty. In *Hello Young Lovers*, 2006.

[13] T-Rex. Jeepster. In *Electric Warrior*, 1971.

[14] 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 *POPL 2015*.

[15] Sonic Youth. Kool Thing. In *Goo*, 1990.

[16] Led Zeppelin. Black Dog. In *IV*, 1971.

A Answers to exercises

A.1 First step in herding cats

Exercise: what do you think? Do you think such an execution is possible?
Yes! On IBM Power or ARM machines for example [5].

A.2 First big cat: tiger

Exercise: SC per location with load-load hazard Certain architectures, such as Sparc RMO [11] allow what is sometimes called *load-load hazard*, i.e. a situation where the `coRR` test that we've just seen is allowed to yield the result `0:r1=1; 0:r2=0;`. How do you think we can build a check that forbids all tests `coWW`, `coRW1`, `coRW2` and `coWR`, but allows the test `coRR`?

In cat speak:

```
let po-loc-llh = po-loc \ R*R
acyclic po-loc-llh | com as sc-per-location-llh
```

Intuitively what we're doing here is removing the read-read pairs (`R*R`) from `po-loc` to build the `po-loc-llh` relation, then enforcing that this new relation is compatible with the communication relations `com`.

Let's run herd on `coWW, coRW1, coRW2`, and `coWR`, and observe that they are still forbidden. Now let's run it on `coRR` and observe that it is allowed. We've built a check that enforces SC PER LOCATION but allows for load-load hazards!

Exercise: Implementing address dependencies How do you think we can implement address dependencies? By this I mean that I'd like to see a sequence of instructions which, placed between two reads, implement e.g. an address dependency.

Consider the following variant of `MP`; the reading thread `P1` has an address dependencies between its two reads:

```
Bell MP+lw+addr
{
x = 0;
y = 0;
}
P0        | P1           ;
w[] x 1   | r[] r1 y     ;
f(lw)     | xor r3 r1 r1 ;
w[] y 1   | add r4 x r3  ;
          | r[] r2 r4    ;
exists (1:r1 = 1 /\ 1:r2 = 0)
```

This dependency, in conjunction with the lightweight fence on `P0`, should be enough to forbid the non-SC execution of `MP`.

Exercise: Distributed `2+2w` Consider the following litmus test, which is essentially a distributed variant of `2+2w`:

```
Bell w+rw+ww
{
x = 0;
y = 0;
}
 P0       | P1       | P2       ;
 w[] x 2  | r[] r1 x | w[] y 2  ;
          | w[] y 1  | w[] x 1  ;
exists (1:r1=2 /\ x=2 /\ y=2)
```

Which fences should we use to forbid the final state? Why? Where should we put them?

We should put a lightweight fence between the read-write pair on `P1`, and a lightweight fence between the write-write pair on `P2`, like so:

```
Bell w+rw+ww+lws
{
x = 0;
y = 0;
}
 P0       | P1       | P2       ;
 w[] x 2  | r[] r1 x | w[] y 2  ;
          | f[lw]   | f[lw]    ;
          | w[] y 1  | w[] x 1  ;
exists (1:r1=2 /\ x=2 /\ y=2)
```

This is because the A-cumulativity of the lightweight fence on `P1` will impose an ordering between the write of `x` on `P0` and the write of `y` on `P1`.

Exercise: Independent Reads of Independent Writes Consider the following litmus test, known as `IRIW`:

```
Bell IRIW
{
x = 0;
y = 0;
}
 P0       | P1       | P2      | P3       ;
 w[] x 1  | r[] r1 x | w[] y 1 | r[] r3 y ;
          | r[] r2 y |         | r[] r4 x ;
exists (1:r1 = 1 /\ 1:r2 = 0 /\ 3:r3=1 /\ 3:r4=0)
```

Which fences should we use to forbid the final state? Why? Where should we put them?

We should put a heavyweight fence between the read-read pairs on `P1` and `P3`, like so:

```
Bell IRIW+hws
{
x = 0;
y = 0;
}
 P0       | P1       | P2      | P3       ;
 w[] x 1  | r[] r1 x | w[] y 1 | r[] r3 y ;
          | f[hw]    |         | f[hw]    ;
          | r[] r2 y |         | r[] r4 x ;
exists (1:r1 = 1 /\ 1:r2 = 0 /\ 3:r3=1 /\ 3:r4=0)
```

This is because `IRIW` essentially is a distributed variant of the store buffering example, therefore reacts to fences in much the same way as `SB`, just like `ISA2` or `WRC` react to fences (lightweight in their case) in much the same way as `MP`.

Thus the A-cumulativity of the heavyweight fence will impose an ordering between the write of `x` on `P0` and the read of `y` on `P1` (idem for the write of `y` on `P2` and the read of `x` on `P3`).

Exercise: SC and TSO Re-Make Re-Model [9] Remember that we've defined SC and (not quite) TSO earlier. Try to reformulate both models in terms of the five principles we've just learnt: SC PER LOCATION, NO THIN AIR, PROPAGATION, OBSERVATION and RESTORING SC. The SC model you'll end up with will be equivalent to the one above. The TSO model you'll end up with will be TSO proper!

Take the tiger `cat` file, and use the following `bell` files. For SC:

```
"Re-Make"
let deps = po
let fhw = po
let fences = fhw
```

and for TSO:

```
"Re-Model"

enum Fences = 'wr
events F[Fences]
let fwr = fencerel(F & Wr)
let deps = po
let fhw = fwr | po \ (W*R)
let fences = fhw
show fwr, fhw
```

A.3 Second big cat: jaguar

Exercise: difference between RMO and Power or ARM What's a test that distinguishes RMO from Power or ARM (as we've defined them previously)? More precisely, what's a test that's forbidden on RMO but allowed on Power or ARM?

`IRIW+deps` distinguishes RMO from Power or ARM:

```
Bell IRIW+deps
{
x = 0;
y = 0;
}
 P0       | P1       | P2      | P3       ;
 w[] x 1  | r[] r1 x | w[] y 1 | r[] r3 y ;
          | f[dep]   |         | f[dep]   ;
          | r[] r2 y |         | r[] r4 x ;
exists (1:r1 = 1 /\ 1:r2 = 0 /\ 3:r3=1 /\ 3:r4=0)
```

because it is forbidden on RMO [1, 2], but allowed on Power and ARM [5].

Exercise: Implementing scope inclusion How would you implement an RMO per scope model as above, but where the fences have an effect not only at their scopes, but also at narrower scopes? That is, the fence for `system` also has an effect at `gpu` and `cta` level for example? This procedure should forbid `mp-mit-scopes+fgpu+fsys`, but not `mp-mit-scopes+fcta+fgpu`.

We need to invent a recursive notion of `wider`, that will return, given a scope level `s`, *all* scope levels wider than `s`, not just the immediately wider one:

```
let rec wider2(s) = match s with
  || 'gpu -> 'system
  || 'cta -> 'gpu | 'system
end
```

We can then use this new notion in the way we define our `rmo-fences`:

```
procedure rmo() =
  let rmo-fences(t) = fencerel(F & wider2(t))
  let rmo-order(t) = (fence(t) | rfe | co | fr) & tag2scope(t)
  forall t in Scopes do
    acyclic rmo-order(t)
  end
end
```

Note how we replace the call to `tag2events` by a call to `wider2` in the definition of `rmo-fences`.

Exercise: Release Sequence In C++ there's a notion of *release sequence*, which says that synchronisation does not just happen via `special-rf`, i.e. from the special write of a special read-from to the corresponding special read. Rather, it can happen from any write of the same location and on the same thread that precedes (in program order) a special write.

How would you modify our model to include this notion?

Like so:

```
let hb = (po | co?;special-rf)+
```

B Complete `bell` and `cat` files

B.1 Kittens

Here's `kittens.bell`:

```
"I'm at your service ma'am, and here's your kitty back"

enum Fences = 'wr
events F[{'wr}]

let fwr = fencerel(F & Wr)
show fwr
```

Here's `kittens.cat`:

```
"How can I thank you for your bringing kitty back?"

include "tutorial.cat"

let fr = rf^-1;co
show fr

let rfe = rf & ext

let com = rf | co | fr

(* SC *)

procedure sc() =
  let sc-order = (po | com)+
  acyclic sc-order
end

(* call sc() *)

(* Almost TSO *)

procedure almost-tso() =
  let ppo = po \ W*R
  let tso-order = ppo | fwr | rfe | co | fr
  acyclic tso-order
end

call almost-tso()
```

B.2 Tiger

Here's `tiger.bell`**:**

```
"He went out tiger hunting with his elephant and gun"

enum Fences = 'dep || 'lw || 'hw
events F[Fences]

let deps = fencerel(F & Dep) & (R * _)
show deps

let flw = fencerel(F & Lw)
show flw

let fhw = fencerel(F & Hw)
show fhw

let fences = flw | fhw
```

Here's `tiger.cat`**:**

```
"In case of accidents he always took his mum"

include "tutorial.cat"

let fr = rf^-1;co
show fr
let fre = fr & ext
let rfe = rf & ext
let com = rf | co | fr

(* Flag non-SC executions *)

procedure sc-flag() =
  let sc-order = (po | com)+
  flag ~acyclic sc-order as non-sc
end

call sc-flag()

(* SC per location *)

let po-loc = po & loc

procedure sc-per-location() =
  acyclic po-loc | com
end

call sc-per-location()
```

```
(* No thin air *)

let hb = (deps | fences | rfe)+

procedure no-thin-air() =
  acyclic hb
end

call no-thin-air()

(* Propagation *)

let A-cumul = rfe;fences
let prop = (fences | A-cumul);hb*

procedure propagation() =
  acyclic prop | co
end

call propagation()

procedure observation() =
  irreflexive fre;prop
end

call observation()

procedure restoring-sc() =
  acyclic fhw | com
end

call restoring-sc()
```

B.3 Jaguar

Here's `jaguar.bell`**:**

```
"I'll call you Jaguar"

enum Scopes = 'cta || 'gpu || 'system

let narrower(s) = match s with
  || 'system -> 'gpu
  || 'gpu -> 'cta
end

let wider(s) = match s with
  || 'gpu -> 'system
  || 'cta -> 'gpu
end

events F[Scopes]
```

Here's `jaguar.cat`**:**

```
"If I may be so bold"

include "tutorial.cat"

let fr = rf^-1;co
let rfe = rf & ext

procedure rmo() =
  let fence(t) = fencerel(F & tag2events(t))
  let rmo-order(t) = (fence(t) | rfe | co | fr) & tag2scope(t)
  forall t in Scopes do
    acyclic rmo-order(t)
  end
end

call rmo()
```

B.4 Panther

Here's `panther.bell`**:**

```
"Kool thing walkin' like a panther"

enum Flavours = 'plain || 'special

events R[Flavours]
events W[Flavours]
```

Here's `panther.cat`**:**

```
"Come on and give me an answer"

include "tutorial.cat"

(* SC per location *)

let po-loc = po & loc

let fr = rf^-1;co
let fre = fr & ext
show fr

let com = rf | co | fr

procedure sc-per-location() =
  acyclic po-loc | com
end

(* No thin air *)

let special-rf = rfe & (Special * Special)

let hb = (po | special-rf)+

procedure no-thin-air() =
  acyclic hb
end

(* Observation *)

procedure observation() =
  irreflexive fre;hb
end
```

```
(* Valid executions *)

procedure valid() =
  call sc-per-location()
  call no-thin-air()
  call observation()
end

(* call valid() *)

(* Races *)

let at-least-one k = (k * _ | _ * k)
let conflict = at-least-one(W) & loc & ext
let race =
  let r = conflict & ~(hb | hb^-1)
  in r \ (id | (I * M) | (M * I) | (Special * Special))
show race

procedure race-free() =
  flag ~empty race as undefined
end

procedure execution() =
  call valid()
  call race-free()
end

call execution()
```