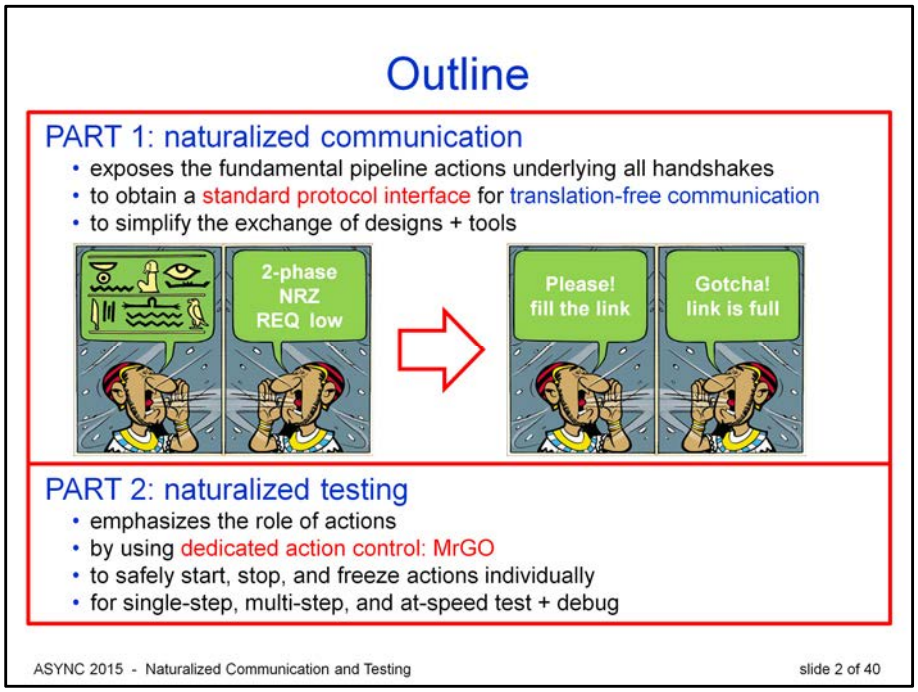
Naturalized communication and testing has been an amazing team builder.
The ideas came out of the research work by the first three authors.
and have been tested on silicon by the other three authors.
You may have seen the chip demos yesterday afternoon.



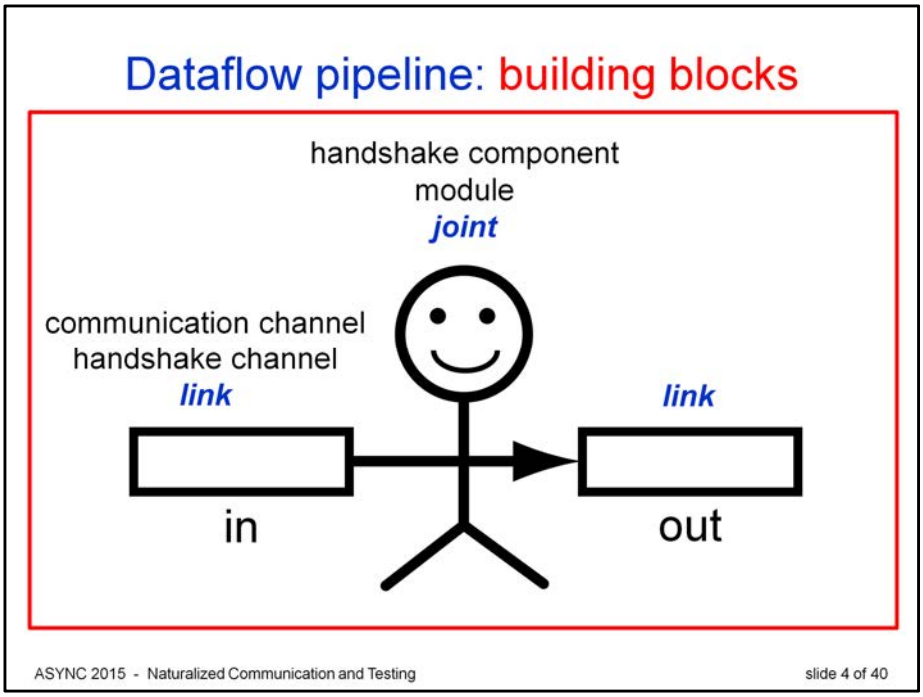
This talk has two parts.

Part 1 is about naturalized communication.
which is about exposing the fundamental pipeline actions
hidden in each and every handshake protocol.
If we use *these actions* instead of the raw handshakes
we can create a **standard protocol interface**
and talk to each other directly - without protocol converters:
<CLICK>
Translation-free communication greatly simplifies
the exchange of designs and tools between different self-timed circuit families.

<CLICK>
Part 2 is about naturalized testing.
Have you ever wondered why your test solution doesn't work for debug?
It's because you and I are so used to seeing everything as state
that we've forgotten the role that actions play.
Traditional test methods manage state in great detail,
but they manage actions very poorly.
The secret to debug is to manage the actions.
I will show you a circuit that allows you to do that - it's called MrGO.
MrGO gives you single-step, multi-step, and at-speed test and debug.

PS Action control also solves the initialization problem presented by
Norman Kluge and Ralf Wollowski in one of the ASYNC 2014 fresh ideas sessions.

We start with Part 1: naturalized communication.

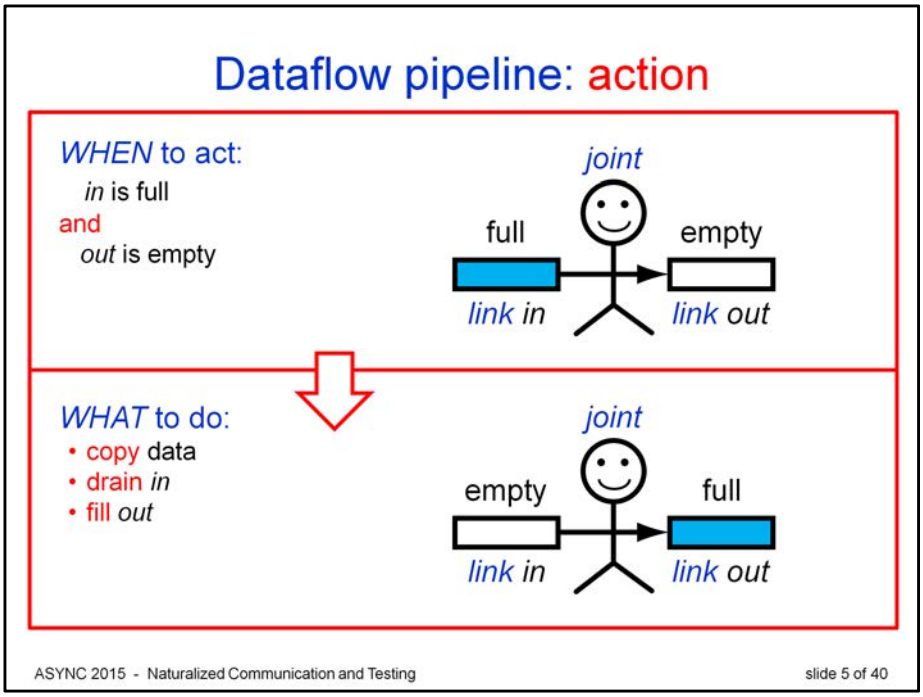


I use the following icons to represent the building blocks of a self-timed system.

The stick-figure in the middle is a handshake component or module.
We call it a **joint**.
Data flow from left to right, in the direction of the arrow.

Each box left and right of the stick-figure is a communication channel,
or handshake channel.
We call it a **link**.
These two links have names *in* and *out.*

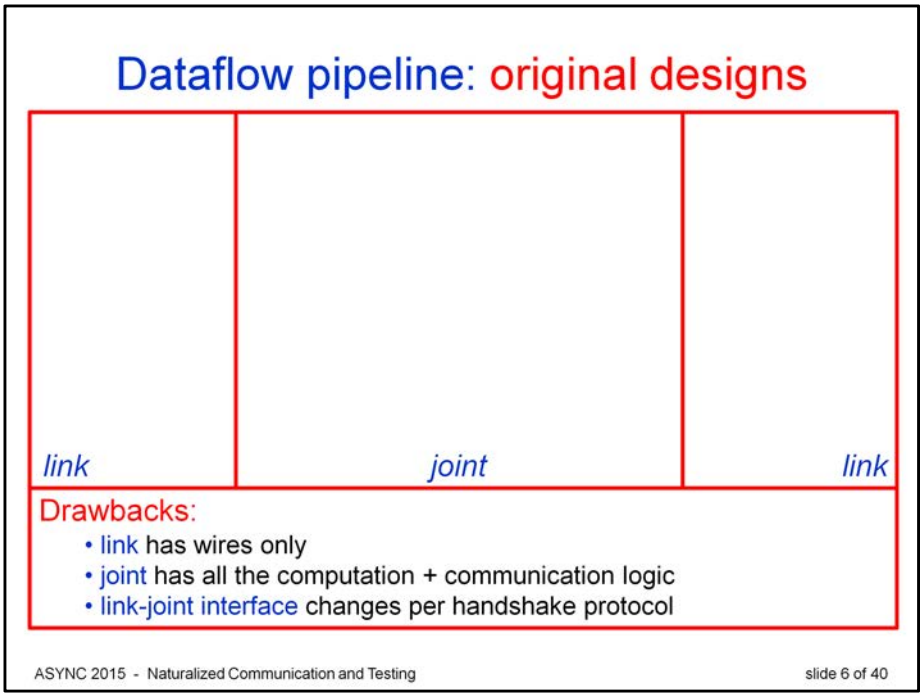
All the systems in this talk are FIFOs, to keep things simple.



The FIFO can either do nothing or act.

It acts **when** link *in* is full and link *out* is empty.
Like this:
<CLICK>
We use blue for full and white for empty.

When it acts it
copies the data, drains link *in*, and fills link *out*
like this:
<CLICK>
Link *in* is now empty
and link *out* is now full.



There are multiple ways to design links and joints.

The next few slides show FIFO designs in four self-timed circuit families.

Even though the designs are different,
they all share the same drawbacks.

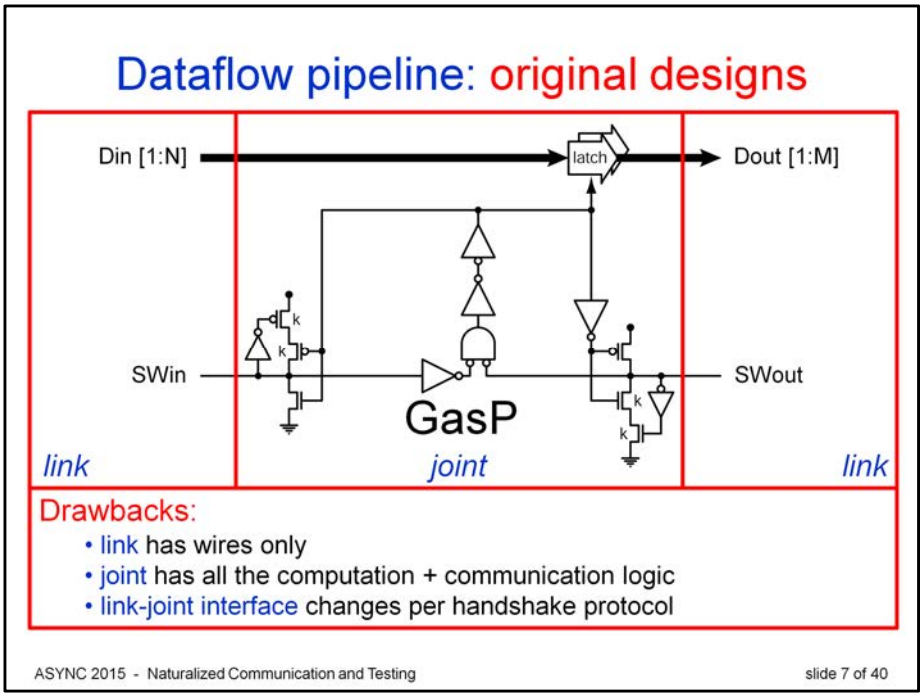
Drawback 1: each link has wires only.

Drawback 2: the joint gets all the logic
the flow control logic,
the datapath logic,
as well the communication logic.

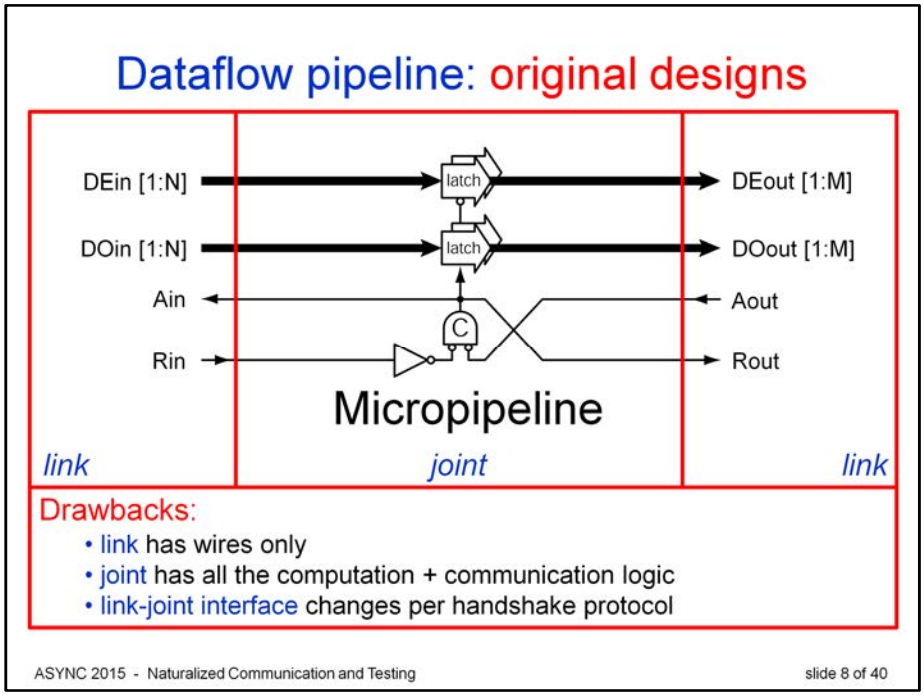
As a result, the link-joint interface changes
whenever the handshake protocol changes.
That's the biggie.
That's the real drawback.

The easiest way to recognize these drawbacks in the next few slides
is to focus on the two links.
If the two links have wires only,
then the drawbacks are there.

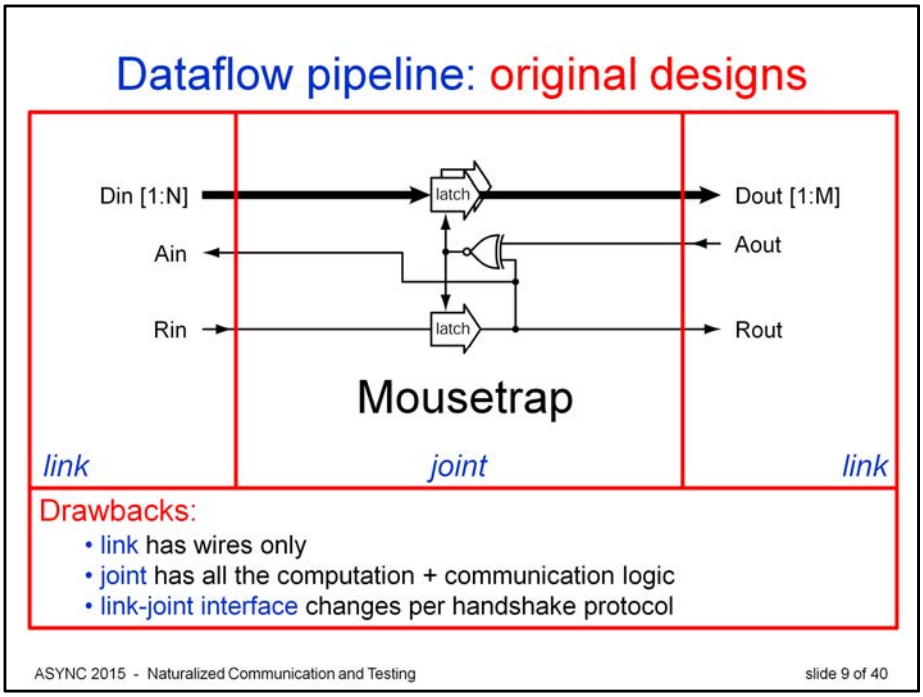
Here we go...



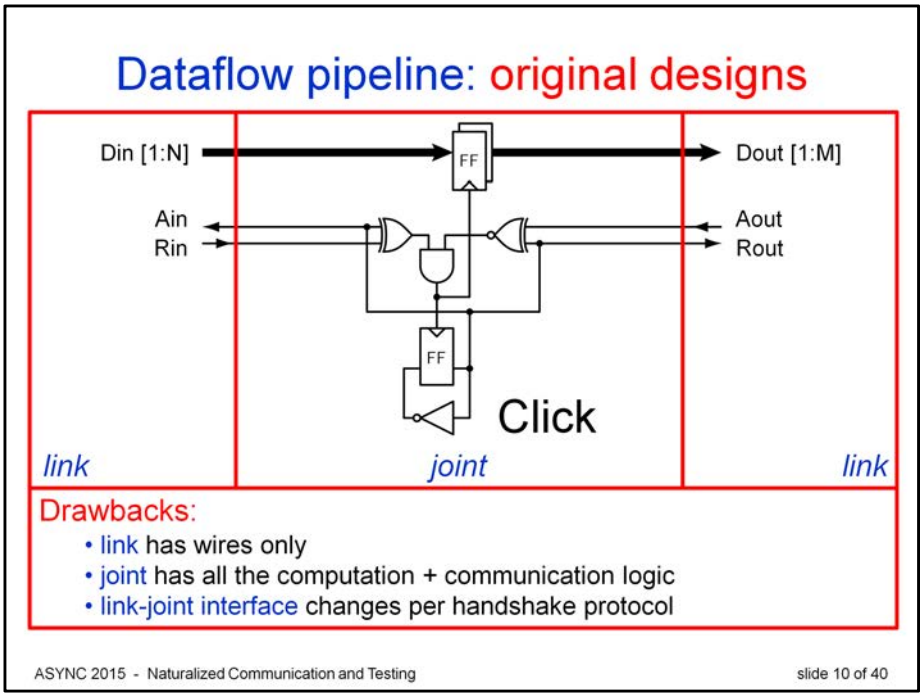
Gasp.
Note that both links have wires only.



Micropipeline.
Both links have wires only.

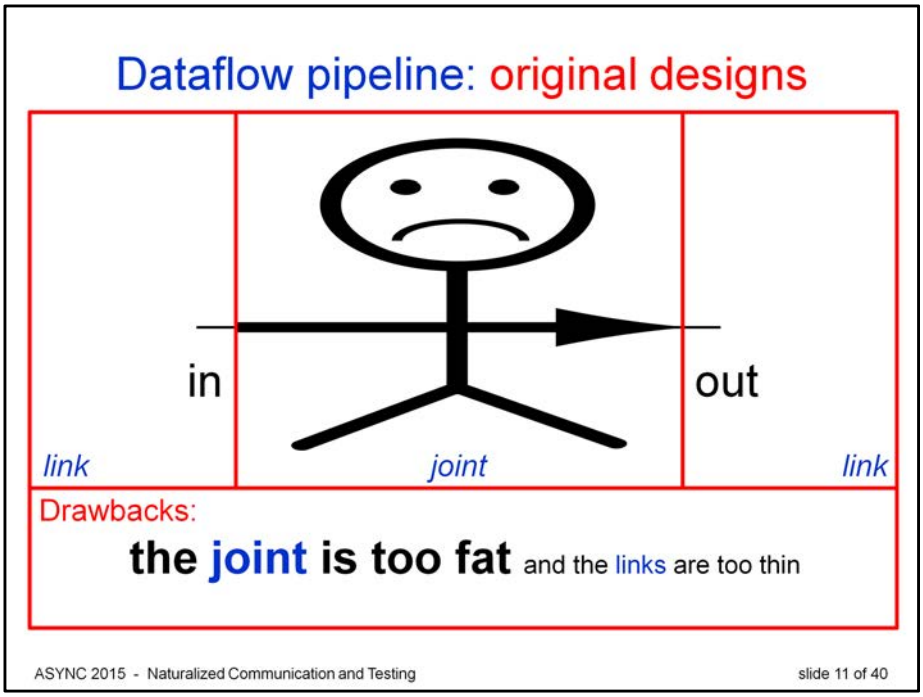


Mousetrap.
Likewise: the left and right links have wires only.



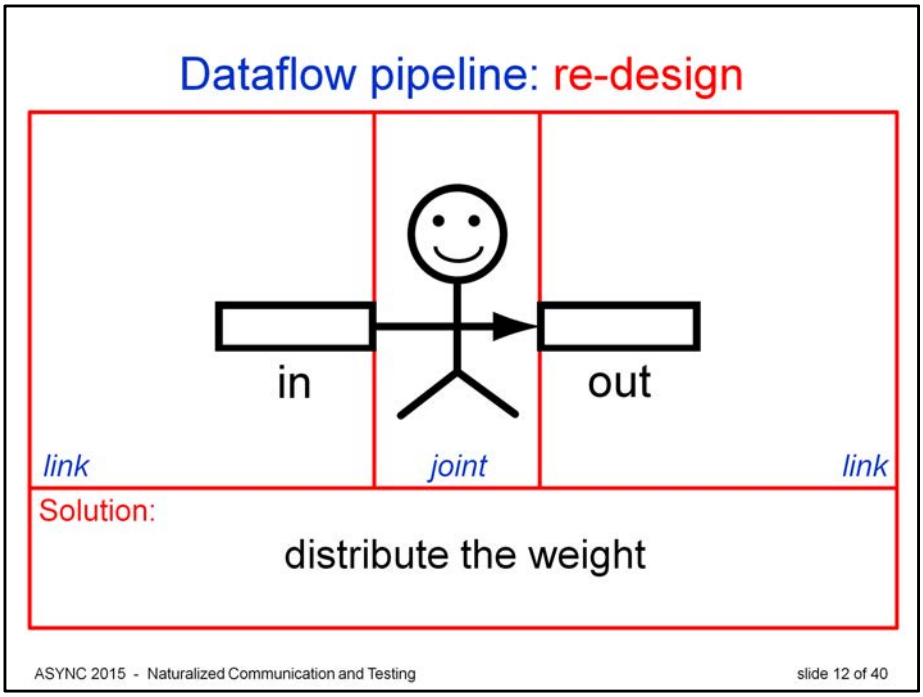
Click.
Same story: both links have wires only.

<NEXT CLICK IS DIFFERENT !!!>



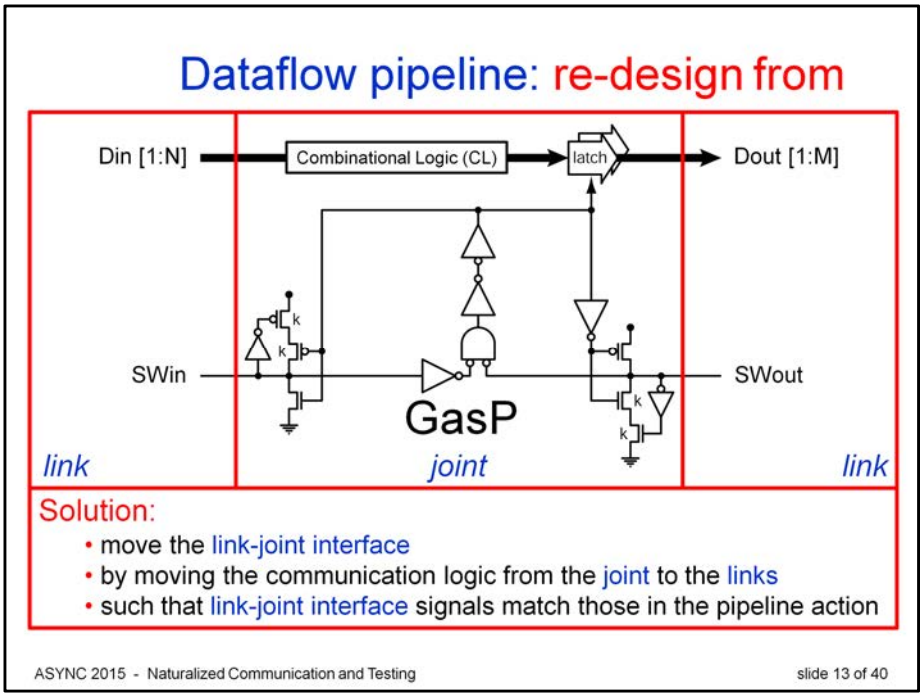
All four families
have over-designed the joint
and under-designed the links.

In everyday English this means
that **the joint is too fat**
and the links are too thin.



An obvious way to solve this
is to distribute the weight.

I will show how to do this for GasP.

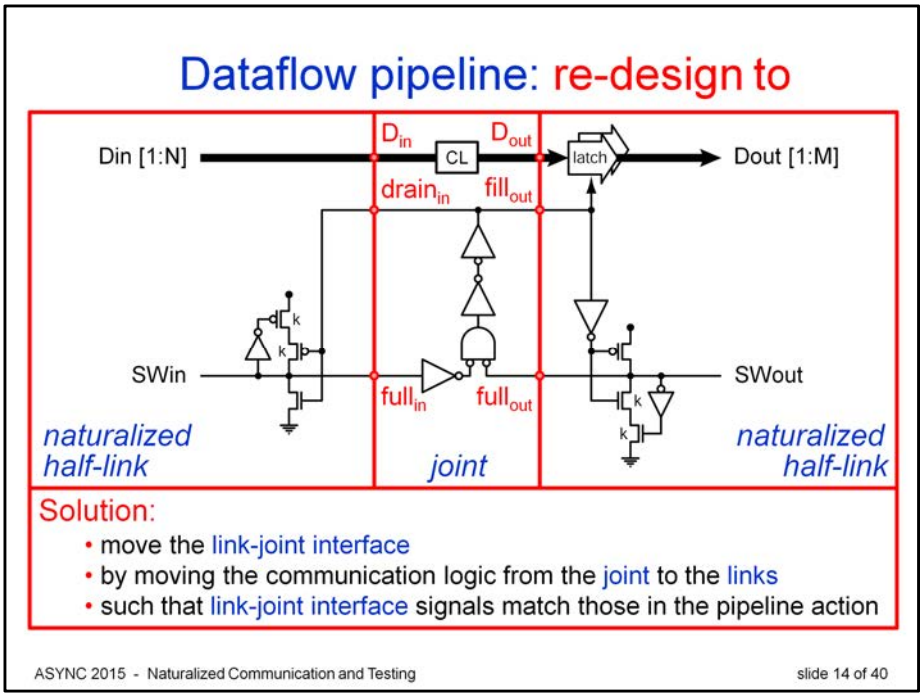


Here is the GasP design again.

Note that I have added combinational logic in the datapath.
<CLICK>
This enables the joint to not just copy
but to also do interesting computations from Data in to Data out.

Remember that we want to distribute the weight
because the joint is over-designed – it's too fat,
and the links are under-designed – they're too thin.

We will distribute the weight by
moving the *communication logic* from the joint to the links
like this:
<CLICK TO NEXT SLIDE>



Let's do that again <GO BACK TO PREVIOUS SLIDE>
We move the link-joint interface from here to <CLICK> here.
<AGAIN> from here to here <CLICK>.

The key observation is that the control signals at the new link-joint interface are an exact match to the signals that we used earlier
to describe the action of a FIFO:
<FROM BOTTOM TO TOP>

- full and empty, where empty is just the negation of full.
- drain
- fill
- and data

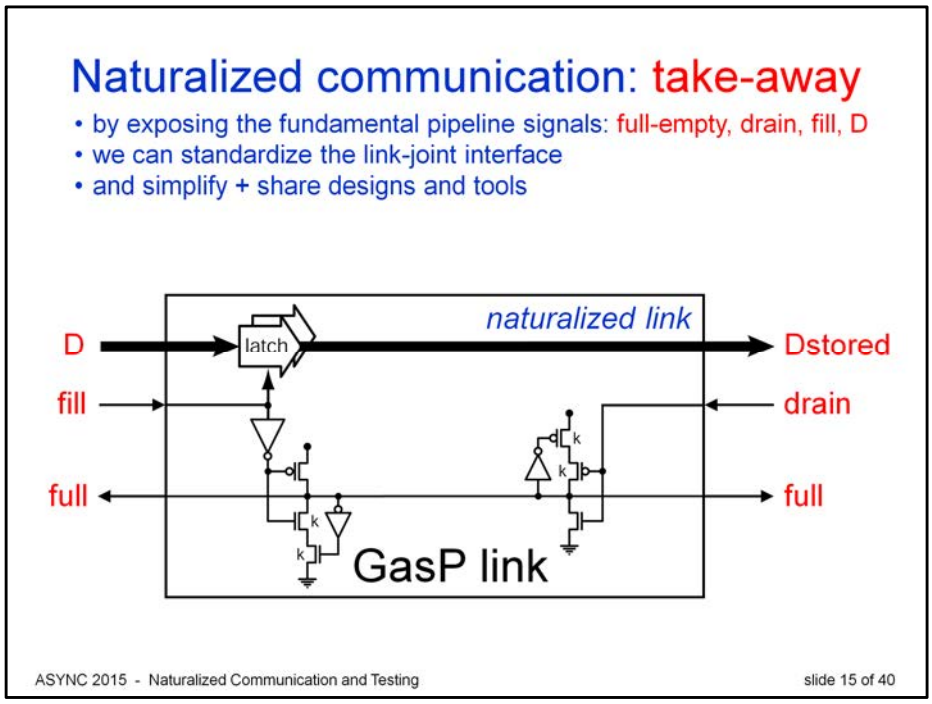
These are the fundamental pipeline protocol signals !
Both you and I use these, but we conceal them
beneath our handshake implementations.

By moving the handshakes further into the links
we expose **the real protocol that we all share**.

There are three more observations for this slide.

1. (ONE) : the data latches have moved to link *out*.
2. (TWO) : the combinational logic remains in the joint.
3. (THREE) : we see only half of each link.

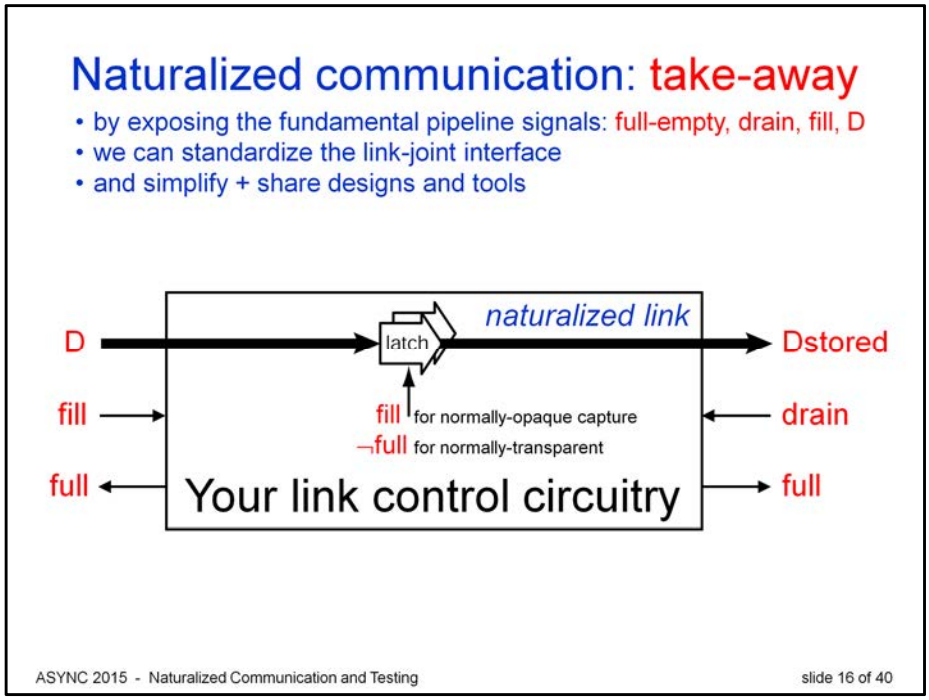
Combine these halves and you get the complete link.
Like this
<CLICK TO NEXT SLIDE>



This is a GasP implementation of a complete link
with the fundamental pipeline signals at its interface.

We call this a *naturalized link*.

Because it's naturalized –
because it uses the fundamental pipeline signals
embedded in each and every handshake protocol –
we can replace the GasP logic
by the link logic of your choice.
<NEXT SLIDE>

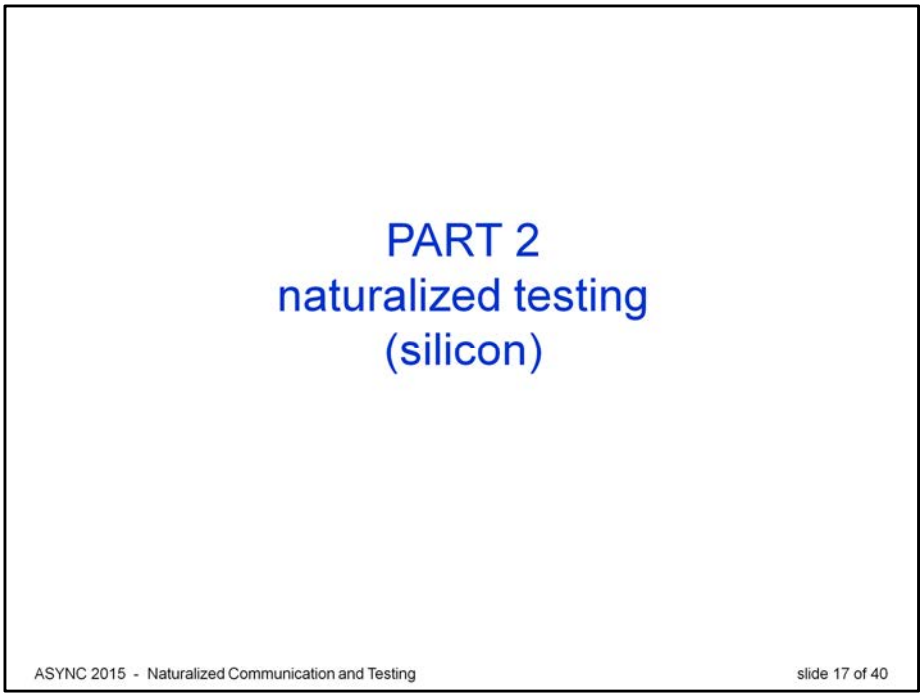


This is the last slide of Part 1.

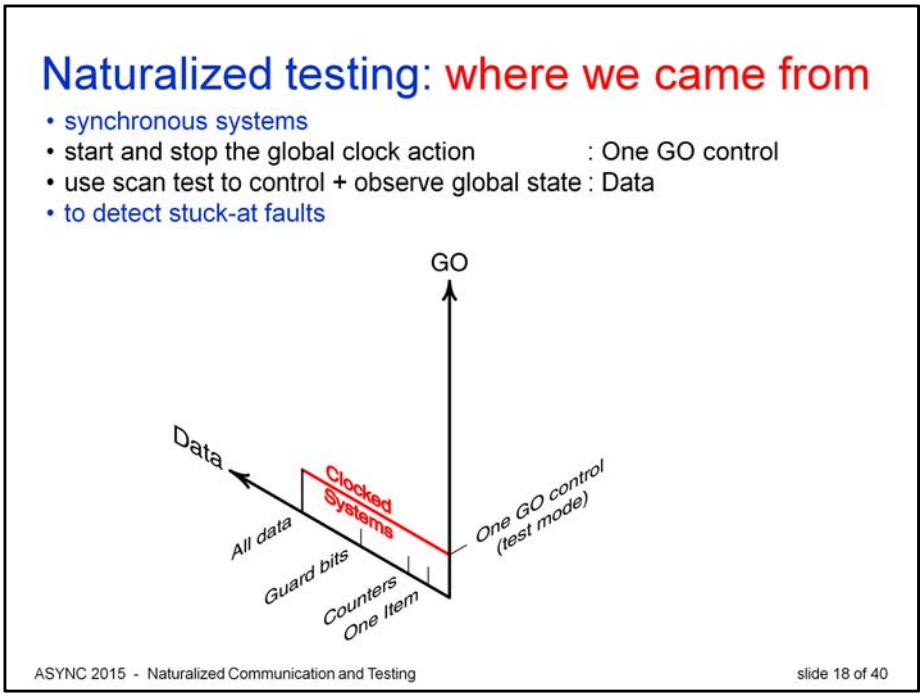
The take-away of Part 1 is that
by exposing the fundamental pipeline signals: full-empty, drain, fill, and data
we can standardize the link-joint interface.

Having a standard interface makes it **very simple**
to exchange and share links, joints, and even design tools.

<NEXT CLICK GOES TO PART 2>



Part 2: naturalized testing



Test solutions for self-timed systems
came from test solutions for synchronous systems.

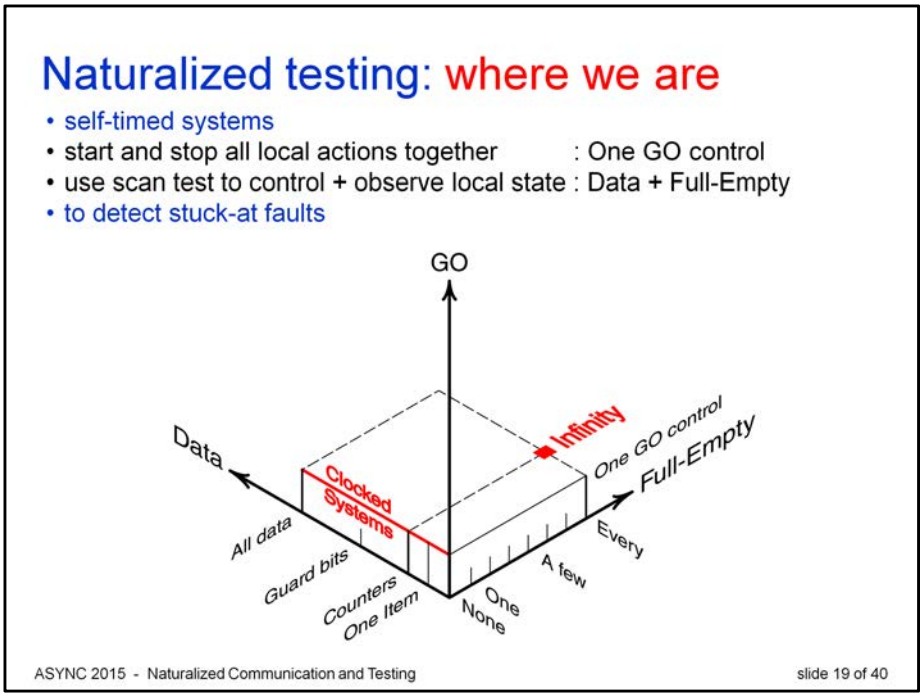
Synchronous systems can start and stop the *global clock action*
and use **scan test** to control and observe *global state.*
They do this primarily **to detect stuck-at faults**

Here is a two-dimensional view of such a test solution.
It has two axes.

The vertical axis controls the clock, by enabling or disabling it.
we call it GO control – you may call it *test mode*.
Traditional clocked systems have **one GO control**.

The horizontal axis labeled **Data** shows
which part of the global state is scanned.
If only *some data* items are scanned, say just the counters,
it's a ***partial*** *scan test* solution.
If all *data* are scanned, it's a ***full*** *scan test* solution.

The test solutions represented by the red line
go from **no scan** to **partial scan** to **full scan**.
Even though we use two axes to describe this red line
the line itself is really one-dimensional.
From our point of view,
traditional scan test is a one-dimensional test method.



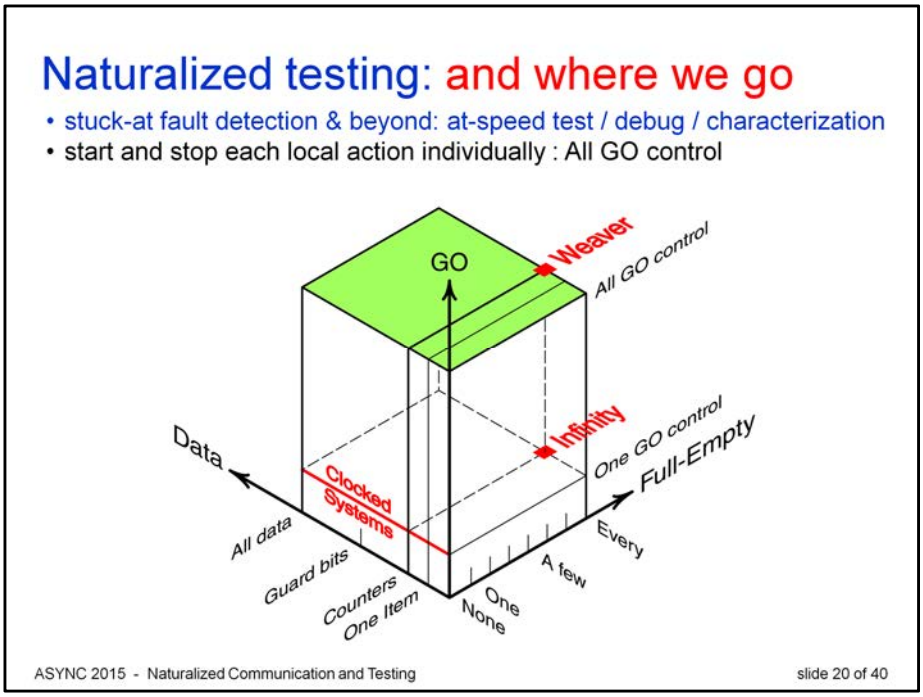
It becomes two-dimensional when we apply it to self-timed systems.

The third axis on the right-hand side of the graph is labeled Full-Empty and represents the control state of the links.

The Infinity chip designed by Sun Microsystems in 2008 lives <HERE>:
it scans every full-empty state,
it scans the counters,
and it can load and unload one Data item.

Because there is exactly one GO control
we have a two-dimensional plane with test solutions.

We went from one-dimensional – the red line – to two-dimensional.



To see the third dimension requires that we recognize
that **a self-timed system isn't about global action.**

The actions of a self-timed system are spontaneous, self-generated,
and widely distributed in both space and time.

It matters to be able to control these distributed actions **separately.**
That's what the GO axis is for:
to control

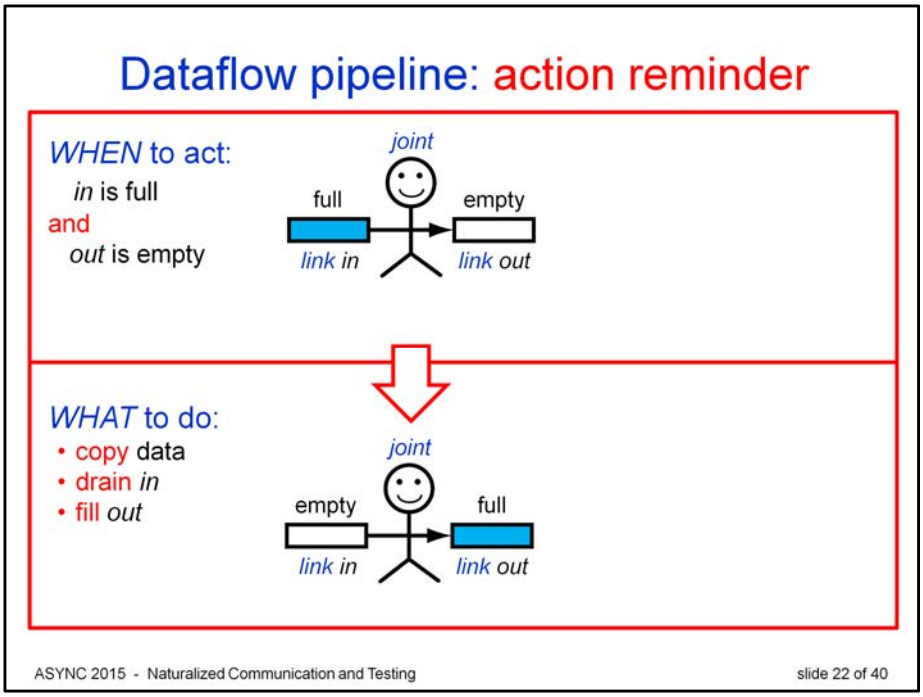
- one action,
- or two
- or three
- ... or all of them.

Our latest chip, the Weaver, lives <HERE>:
it scans all GO control signals,
it scans every full-empty state,
it scans the counters,
and it can load and unload one Data item.

What you get with this **dedicated GO action-control**
goes way beyond stuck-at fault detection.

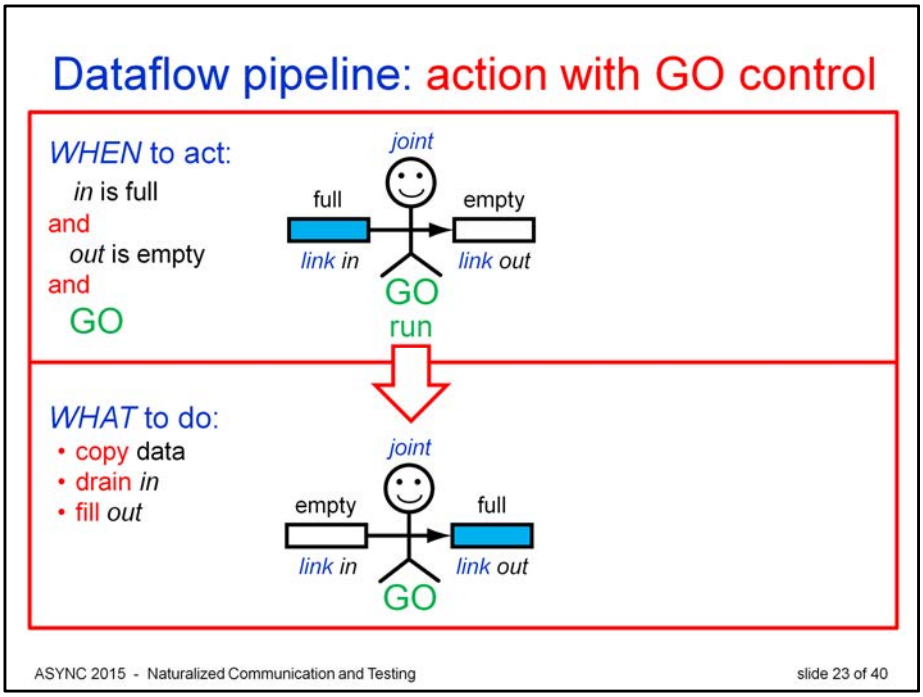
Dedicated action control combined with traditional scan access to state
(Data and Full-Empty)
gives you not only stuck-at fault detection,
but also at-speed test, debug, and characterization.

So, how do we GO there?



The first step is to recognize self-timed actions.
Here is a reminder of what a self-timed action looks like.

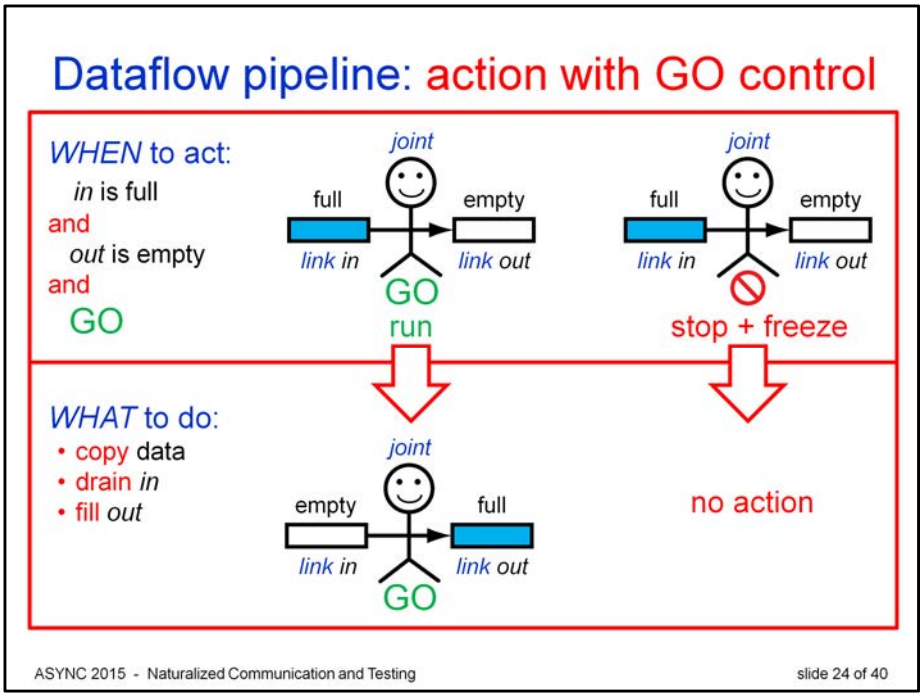
When link *in* is full (blue) **AND** link *out* is empty (white)
copy the data
drain *in*
and fill *out*.



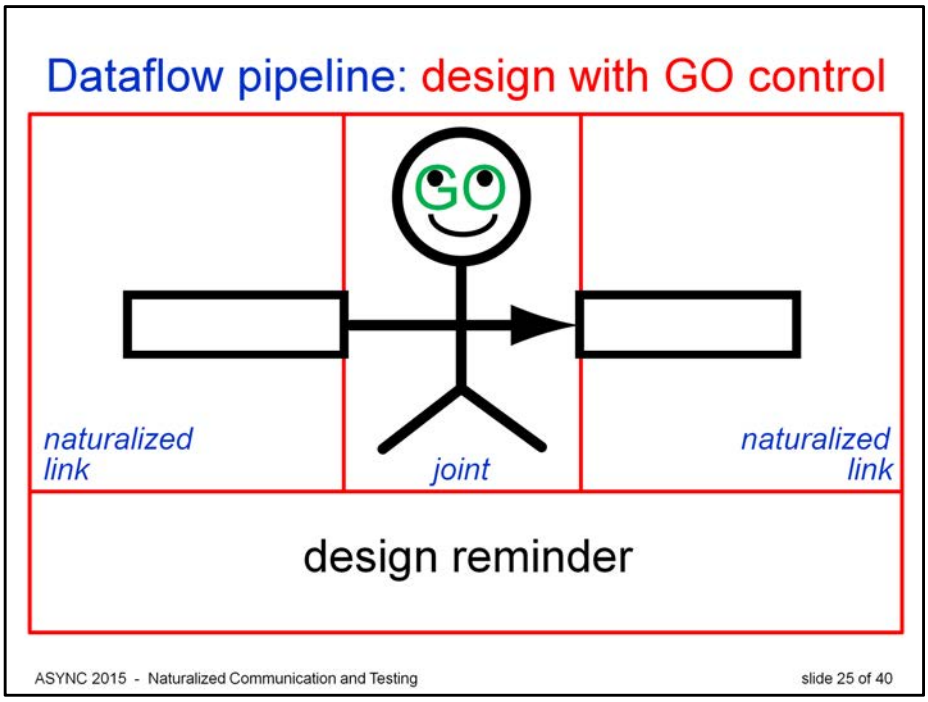
To control this action
we add a GO control signal to the **and-function** in the **when** part of the action.
We leave the **what** part as is.

Now:
When link *in* is full (blue) **AND** link *out* is empty (white) **AND** GO is enabled
copy the data
drain *in*
and fill *out*.

When GO is enabled, the action runs as before.



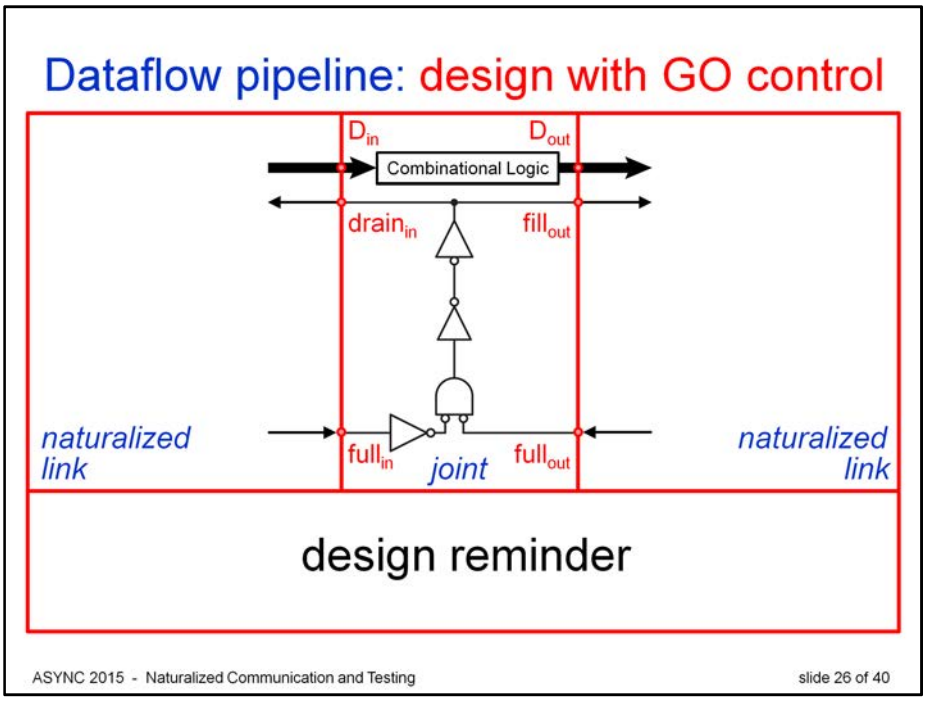
But when GO is **dis-**abled
the action stops and freezes.



Where do we add this GO control?

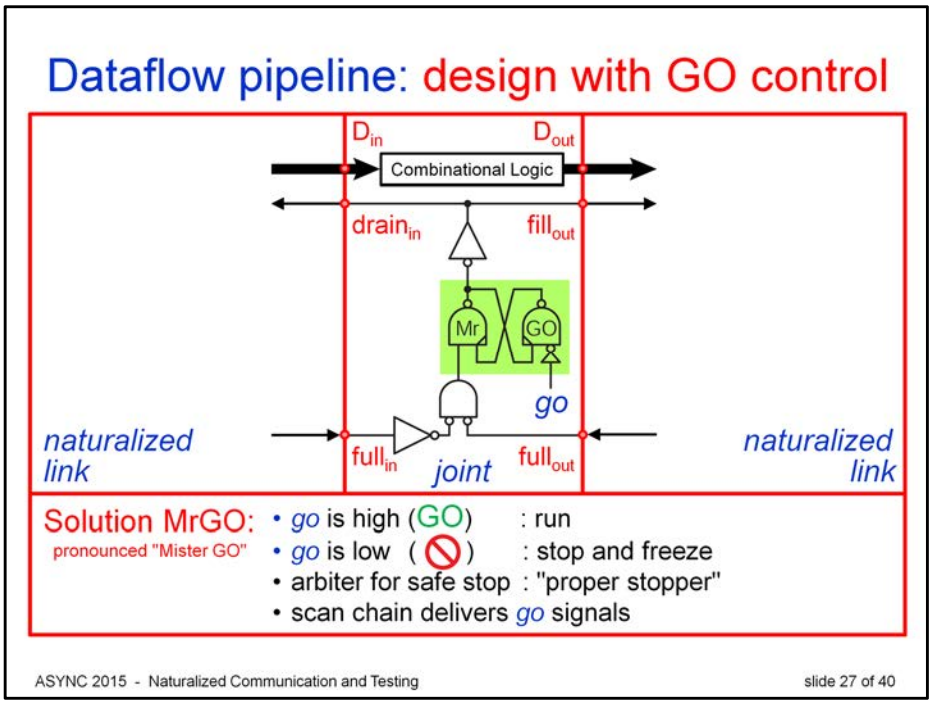
Well ... the and-function is located in the joint
so the joint gets the GO control.

Here is a reminder of what a joint looks like
<CLICK to NEXT SLIDE>



It's an and-function

and it has the combinational logic for the datapath.



We add the GO control signal in the tail of the and-function,
and we call it *go.*

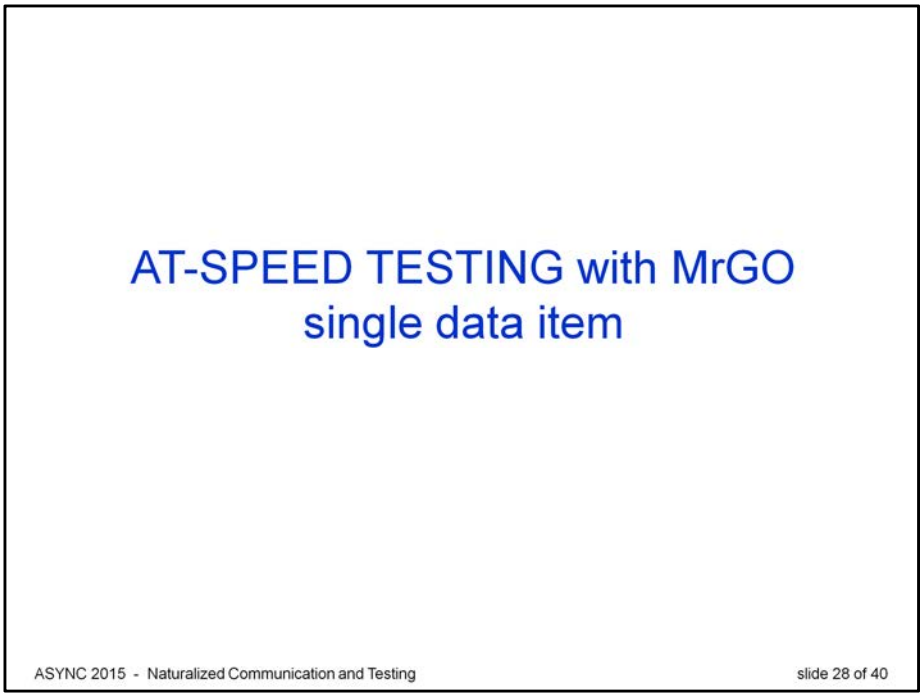
The *go* signal comes with its own arbiter
so we can safely stop self-timed actions in full flight.

The green box with the arbitrated nand-gate and the go signal
is called **"Mister GO" (MrGO).**

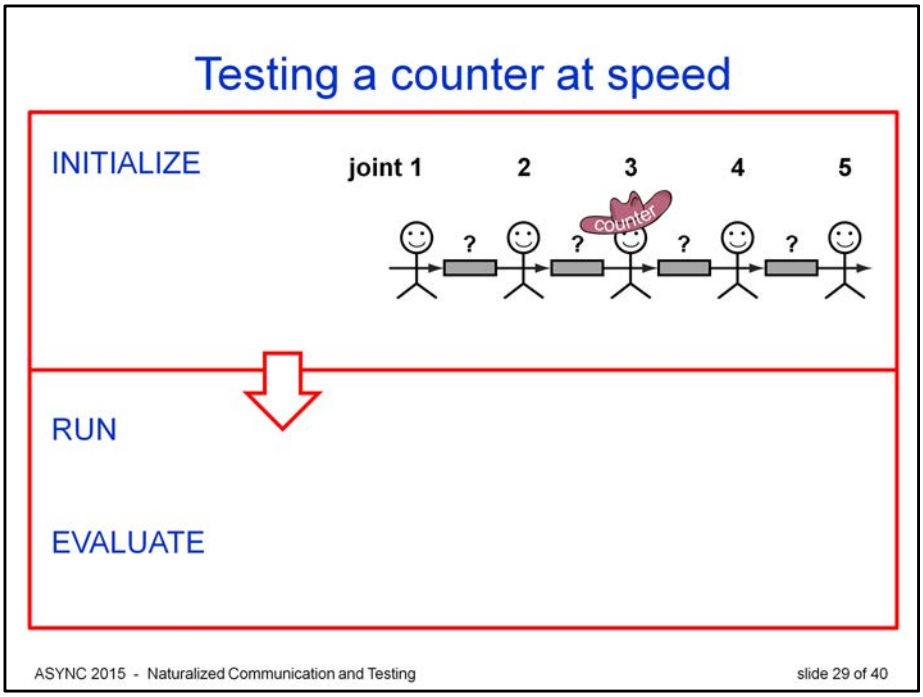
When *go* is high
MrGO unfreezes the joint, and allows it to run as usual.

When *go* is low
MrGO acts like a **proper stopper** and will stop and freeze the joint action.

We use the scan chain to deliver the individual *go* signals.



The proof of the pudding is in the eating.
So, let's do a test.

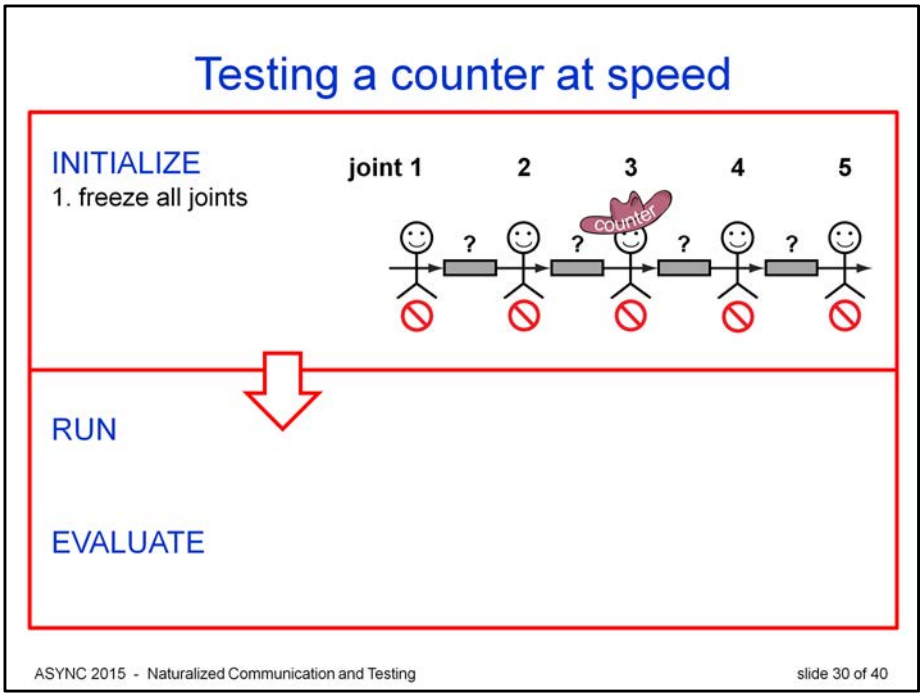


Here is a FIFO with 5 joints.
Joint number 3 has a cowboy hat - that's a counter.

Your task should you decide to accept it
is to test the counter at speed.

You can do this in three steps.
Initialize, run, and evaluate.

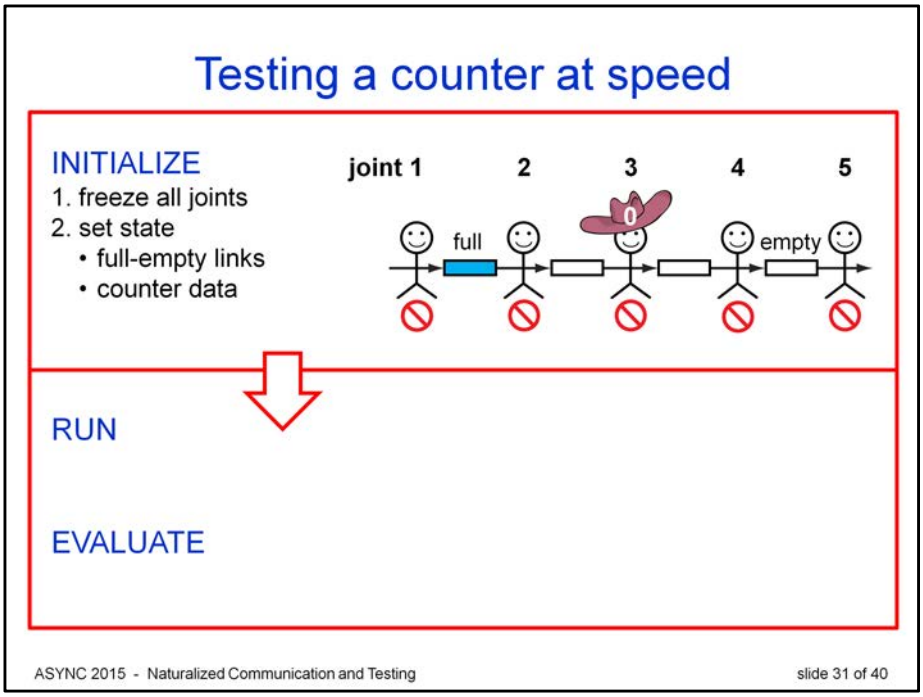
NOTE:
for proper alignment, the text has been hidden by making it white,
and uncovered in the next few slides by making it black.



To initialize the system, you first freeze all the joints.

You do this by making all the *go* signals low
as indicated by the red stop signs.

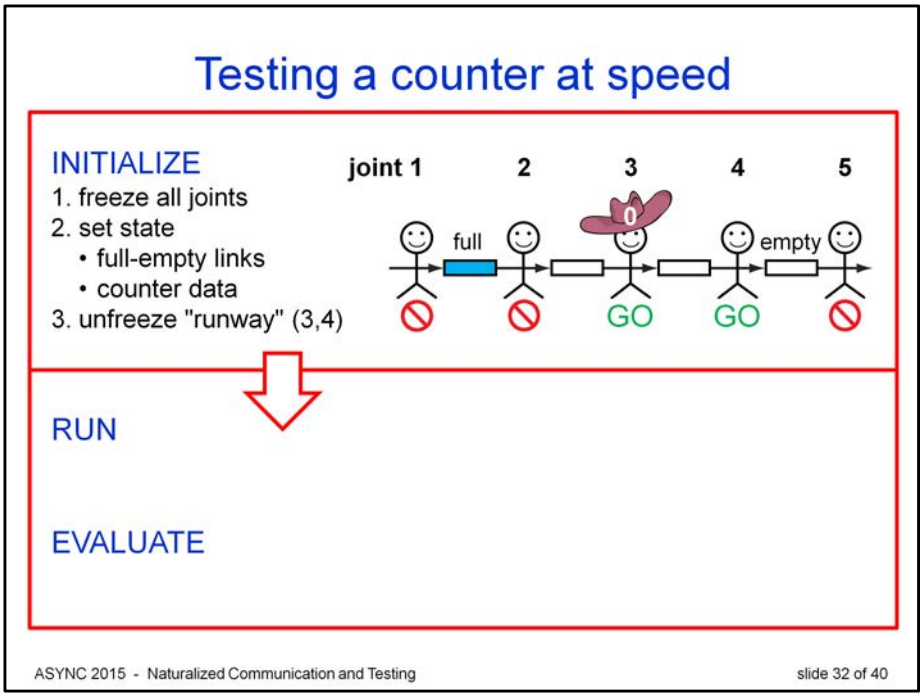
You have now stopped every action in this FIFO.



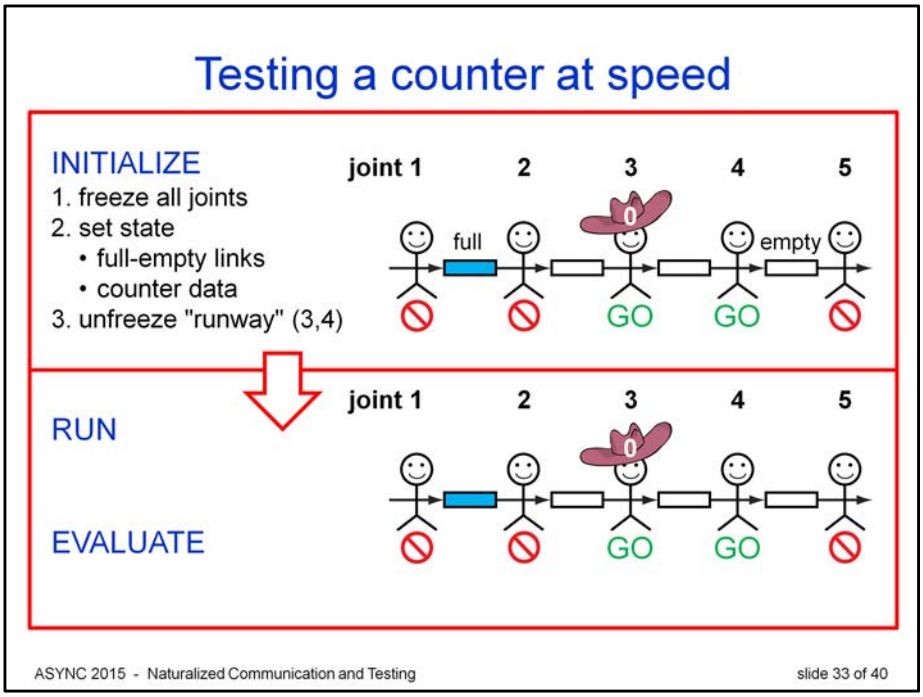
Next, you set the state.

To run one Data item through this FIFO,
you make the first link full
and the other links empty.

You set the counter value to let's say zero.



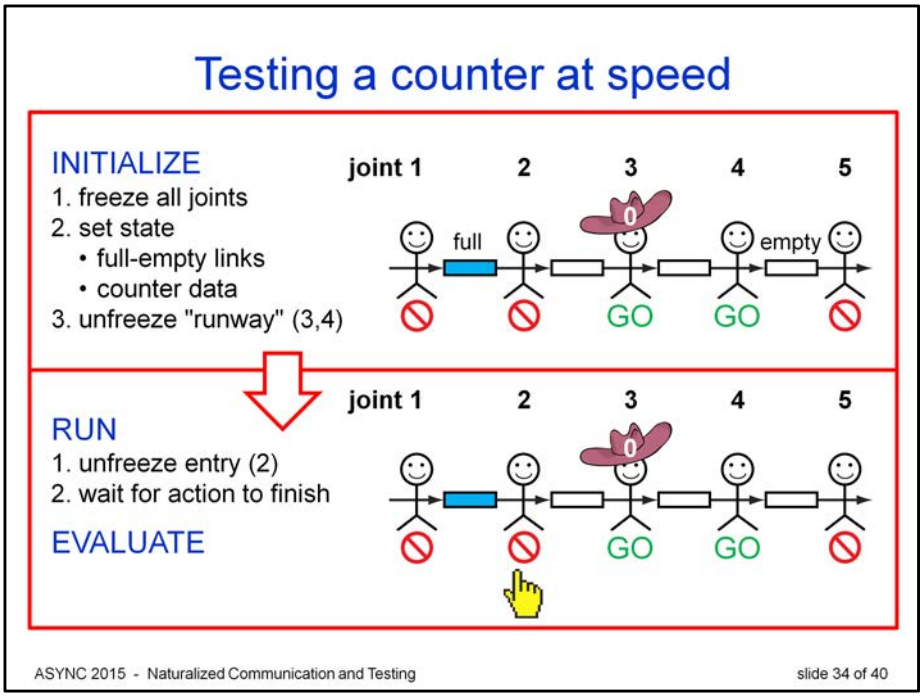
Then you clear the runway by unfreezing joints 3 and 4.



And that's your initial state.

You've been very careful
and kept the first and last joints frozen,
to keep other test inputs out
and to keep your test results in.

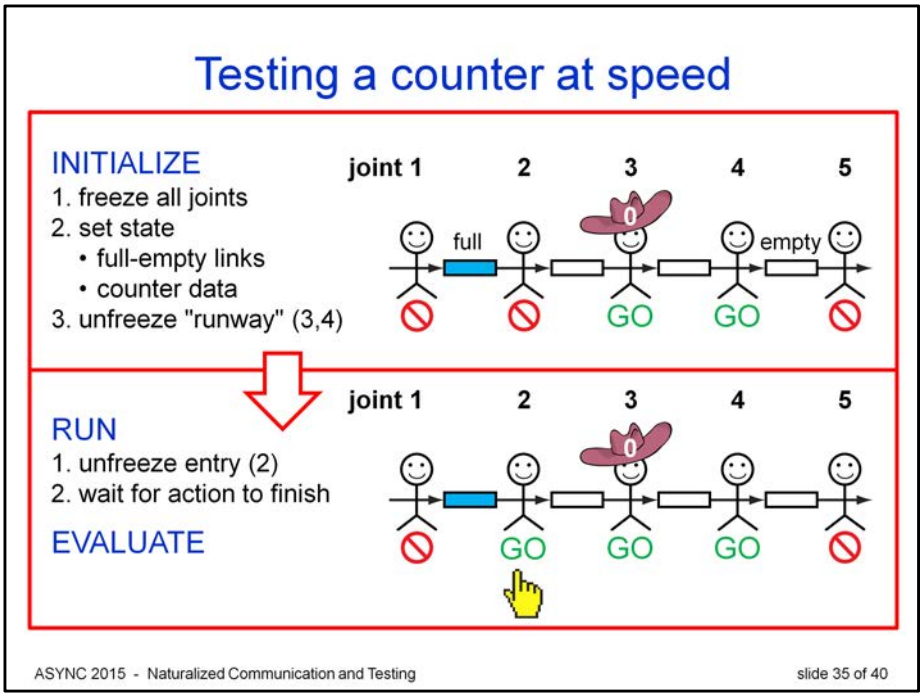
Goodonya!



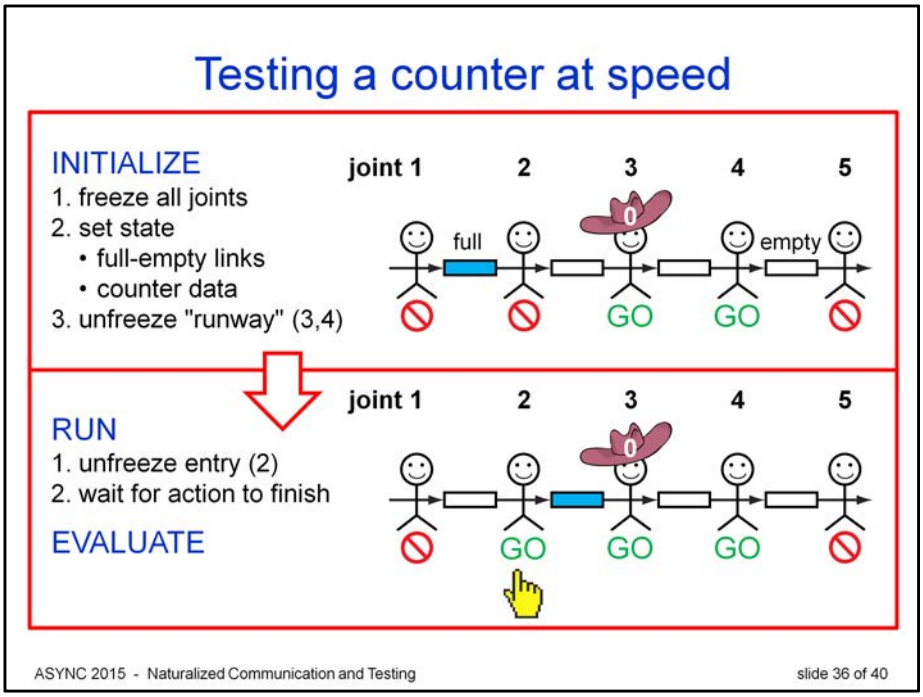
You're about to unfreeze the entry to the runway,
by making the *go* signal of joint number 2 high.
That's the *go* signal with the hand-cursor.

You know things will happen fast when you do that.
So, you're gonna pay close attention
to make sure you see the **blue Data** move from left to right
and to see it update the counter.

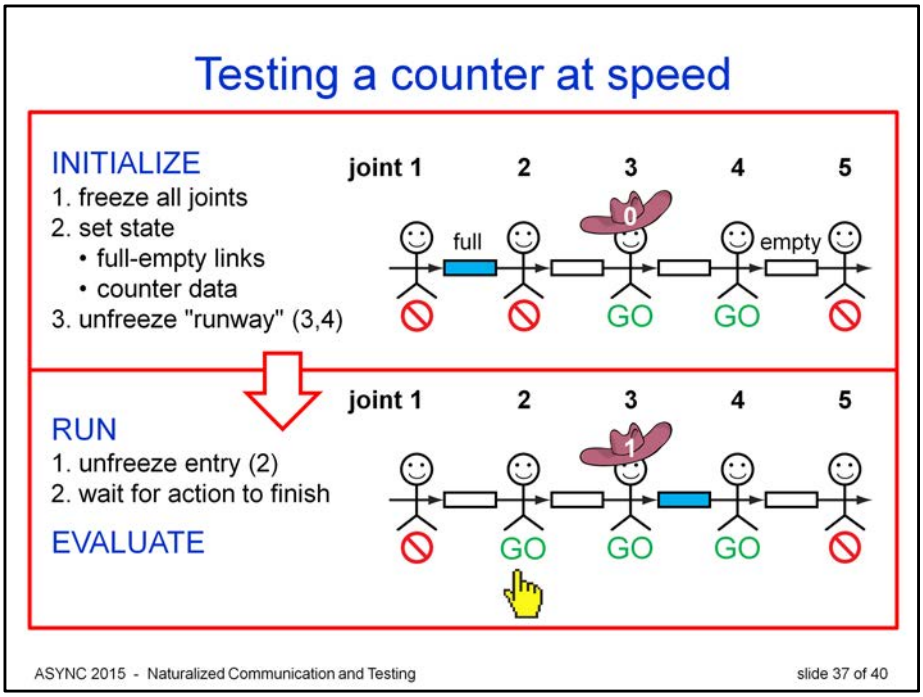
GO!


Testing a counter at speed
INITIALIZE
1. freeze all joints
2. set state
• full-empty links
• counter data
3. unfreeze "runway" (3,4)
joint 1
2
3
4
5
full
0
empty
GO
GO
RUN
1. unfreeze entry (2)
2. wait for action to finish
EVALUATE
joint 1
2
3
4
5
0
GO
GO
GO
ASYNC 2015 - Naturalized Communication and Testing
slide 35 of 40

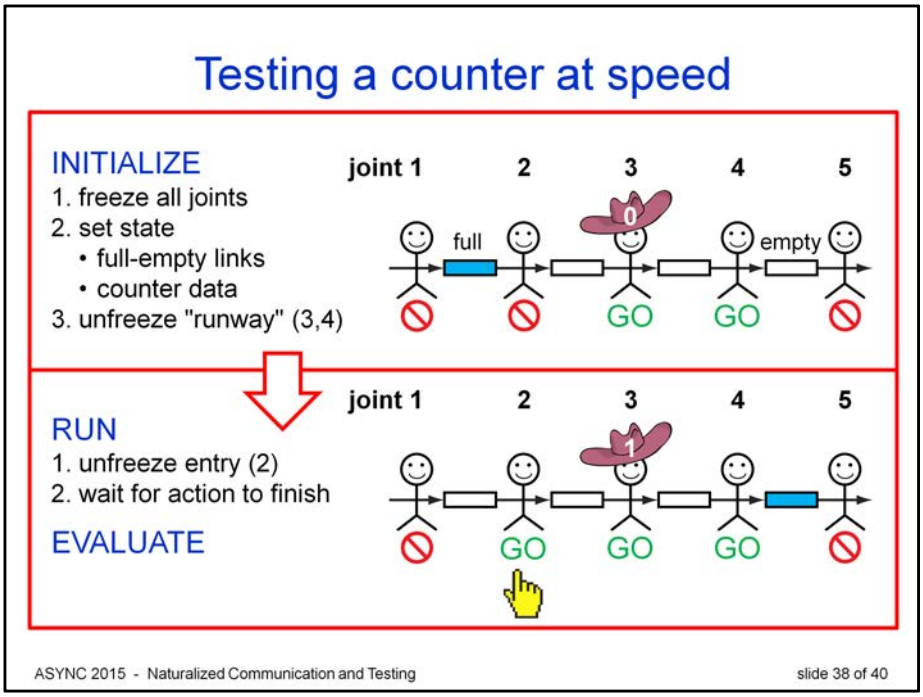
<SELF-TIMED>


Testing a counter at speed
INITIALIZE
1. freeze all joints
2. set state
• full-empty links
• counter data
3. unfreeze "runway" (3,4)
joint 1
2
3
4
5
full
empty
0
GO
GO
RUN
1. unfreeze entry (2)
2. wait for action to finish
EVALUATE
joint 1
2
3
4
5
0
GO
GO
GO
ASYNC 2015 - Naturalized Communication and Testing
slide 36 of 40

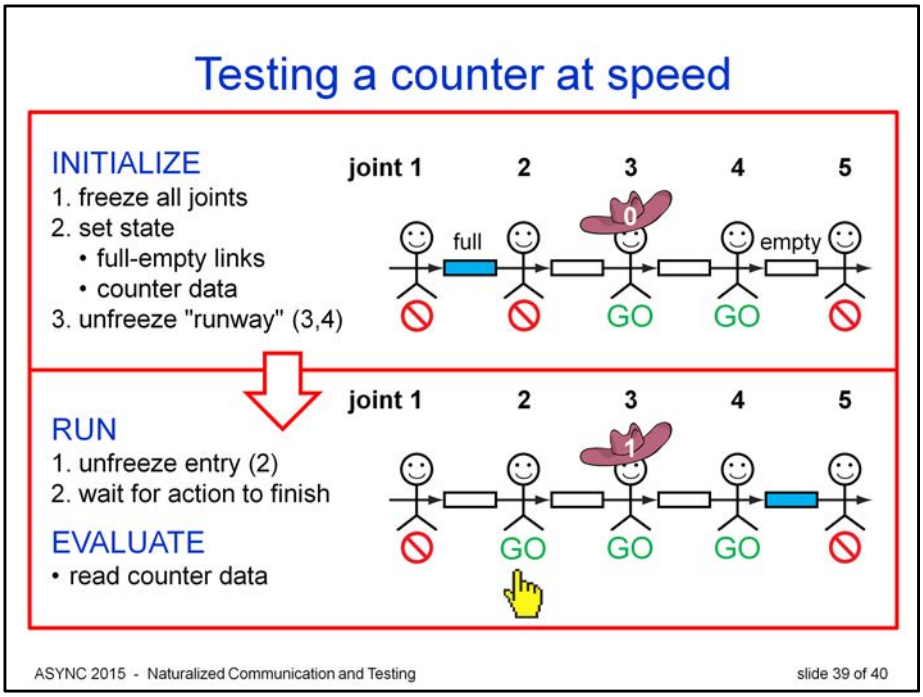
<SELF-TIMED>


Testing a counter at speed
INITIALIZE
1. freeze all joints
2. set state
• full-empty links
• counter data
3. unfreeze "runway" (3,4)
joint 1
2
3
4
5
full
0
empty
GO
GO
RUN
1. unfreeze entry (2)
2. wait for action to finish
EVALUATE
joint 1
2
3
4
5
1
GO
GO
GO
ASYNC 2015 - Naturalized Communication and Testing
slide 37 of 40

<SELF-TIMED>


Testing a counter at speed
INITIALIZE
1. freeze all joints
2. set state
• full-empty links
• counter data
3. unfreeze "runway" (3,4)
joint 1
2
3
4
5
full
0
empty
GO
GO
RUN
1. unfreeze entry (2)
2. wait for action to finish
EVALUATE
joint 1
2
3
4
5
1
GO
GO
GO
ASYNC 2015 - Naturalized Communication and Testing
slide 38 of 40

<SELF-TIMED>



Now you scan the counter data out
and check if it's one.

Well-done!

<next slide ends the talk>

We have two working silicon experiments: Weaver and Anvil.
They both use MrGO and JTAG-scan-access for test, debug, and characterization.

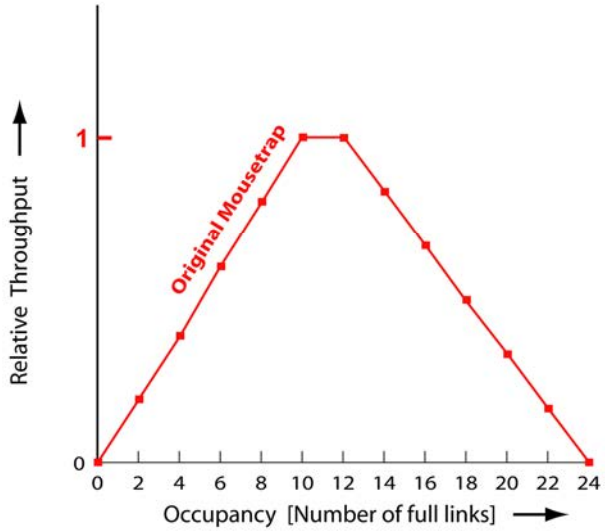
What's more - they're both at the conference.

LIVE demos and tests are available.
Just ask Swetha, Hoon, Nav, Chris, or Ivan.
We are here until Thursday.

BACK-UP SLIDES

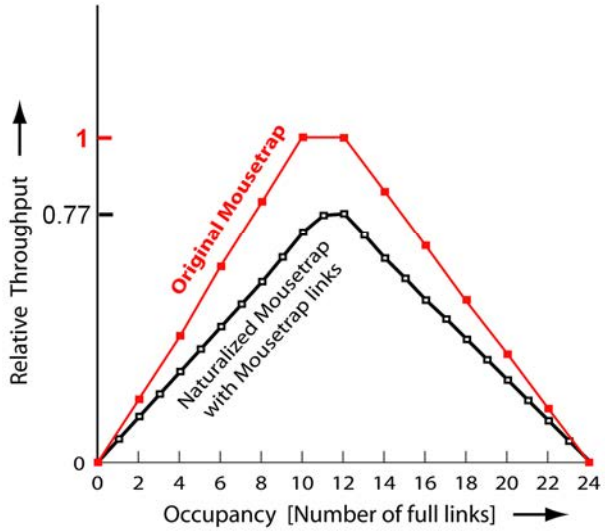
THROUGHPUT
original and naturalized Mousetrap

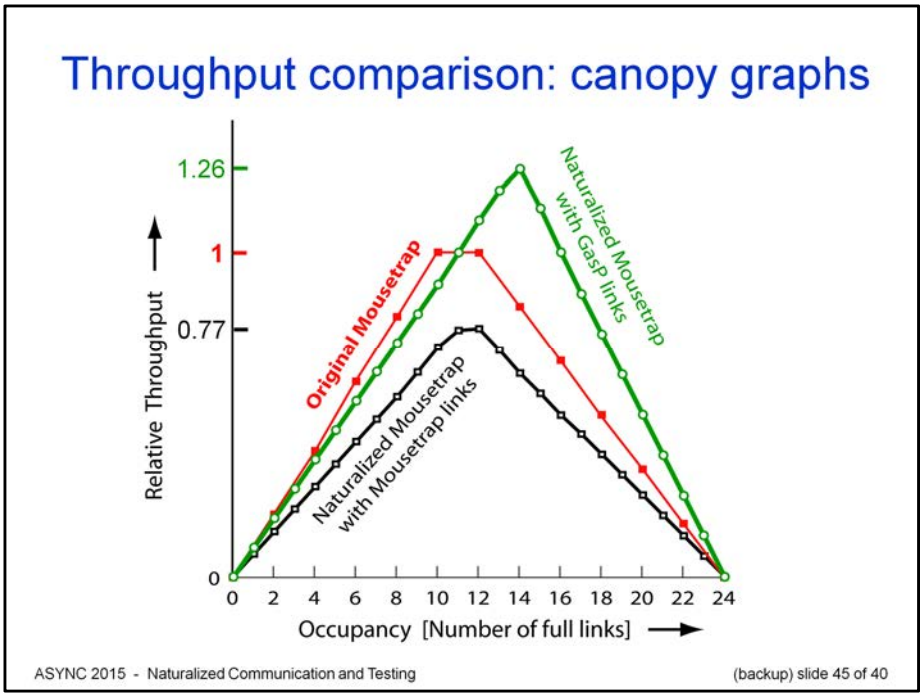
Throughput comparison: canopy graphs



ASYNC 2015 - Naturalized Communication and Testing (backup) slide 43 of 40

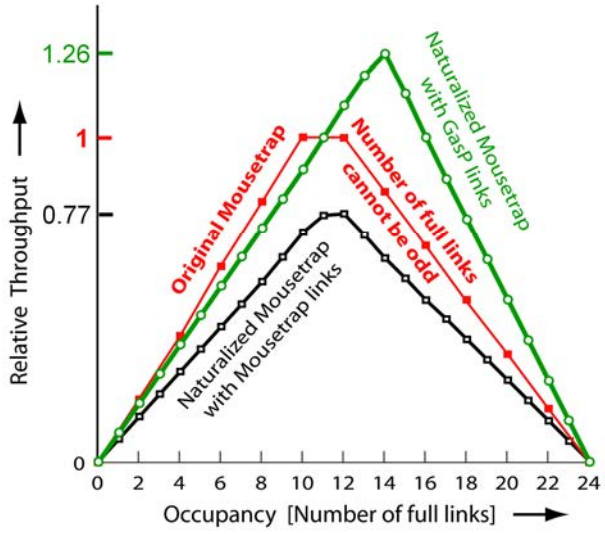
Throughput comparison: canopy graphs



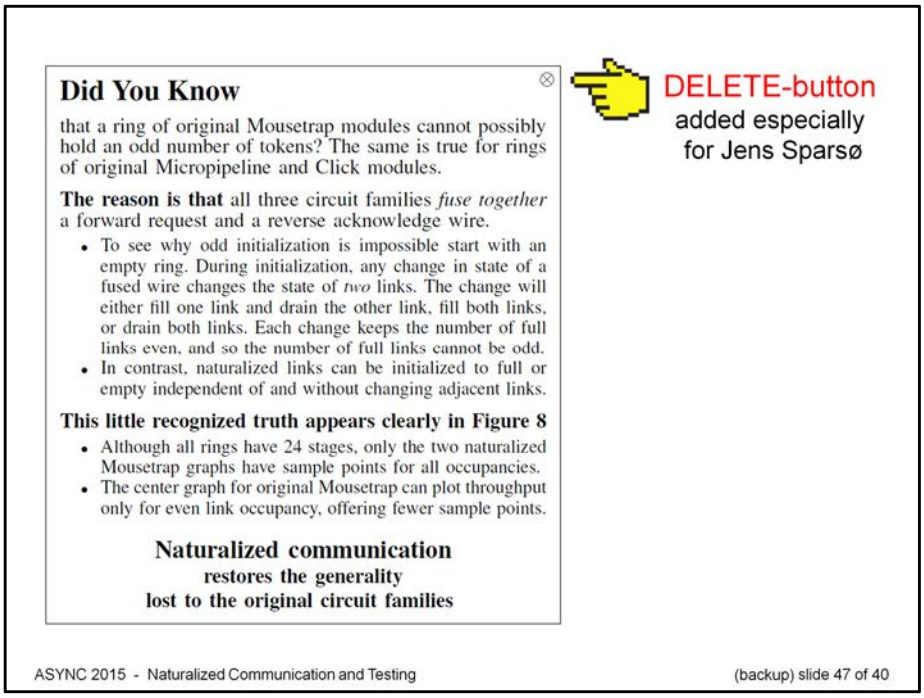


The green graph is below parts of the original Mousetrap, because we used 6-4 GasP. Using 4-6 or 5-5 Gasp would take the green graph completely above the original one.

Throughput comparison: canopy graphs



ASYNC 2015 - Naturalized Communication and Testing (backup) slide 46 of 40

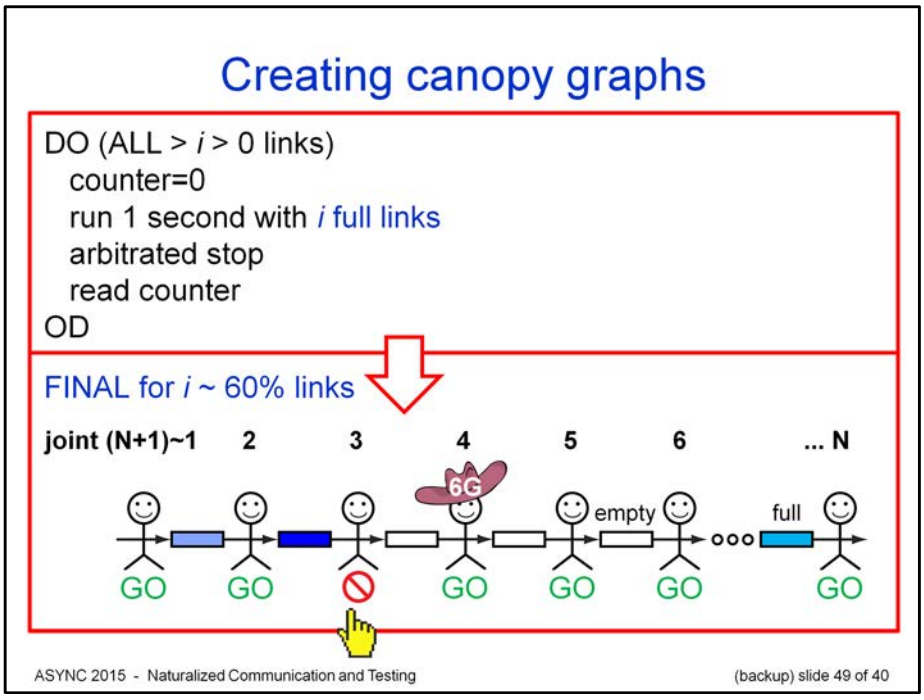


From the paper.

Note the special "delete" button in the top-right corner: ⊗

We added this especially for Jens Sparso
who did not like the advertisement look and feel.

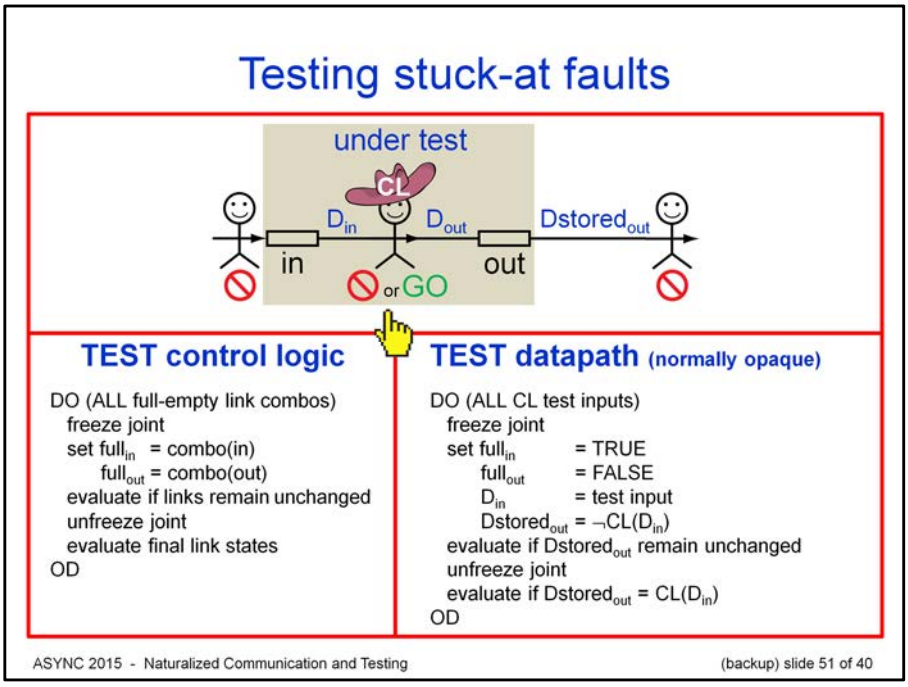
CANOPY GRAPHS
characterization with MrGO



The Weaver's throughput of 6 Giga Data Items per second correspond to **3.5 Terabits per second** !!!

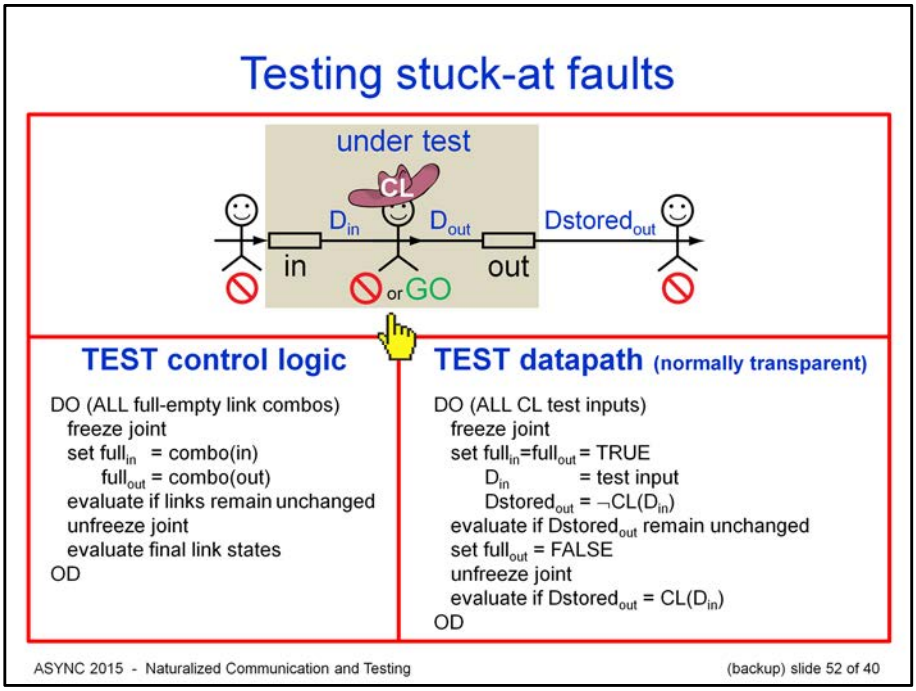
- The Weaver is an 8x8 crossbar.
- Words of 72 bits each, also known as "Data items."
- Non-blocking, totally concurrent.
- TSMC 40 nm CMOS.
- 6 x 10^9 Data items per second per channel
  = 430 Gigabits per second per channel
  = 8 x 430 Gigabits per second
  ~ 3.5 Terabits per second
- Parts suitable for Network-on-Chip

STUCK-AT FAULTS
one-shot testing with MrGO



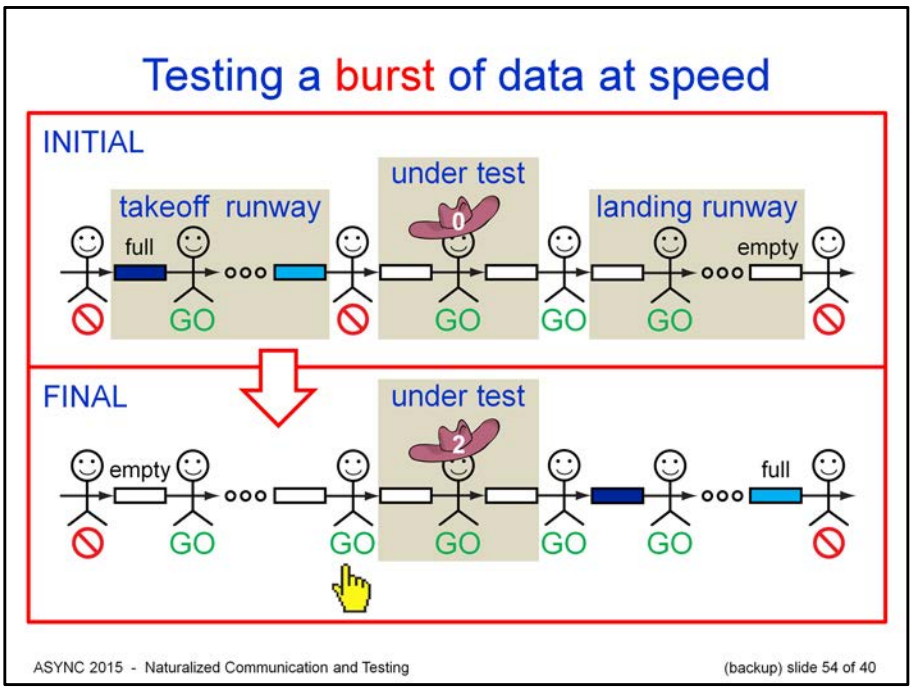
This picture is the equivalent of traditional scan test with one GO control.

Evaluating prior to unfreezing the joint allows us to detect premature actions.



Minor differences to testing the normally-opaque data capture version in the previous slide.

AT-SPEED TESTING
of data burst with MrGO


Testing a burst of data at speed
INITIAL
takeoff runway
full
GO
under test
0
GO
GO
landing runway
GO
empty
FINAL
empty
GO
GO
under test
2
GO
GO
GO
full
ASYNC 2015 - Naturalized Communication and Testing
(backup) slide 54 of 40


MrGO
ASYNC 2015 - Naturalized Communication and Testing
(backup) slide 55 of 40

MrGO: dedicated action control

- go is high (GO) – start in to out
- go is low (⊘) – stop or freeze in to out
- arbiter for safe stop – "proper stopper"
- scan chain delivers go signals