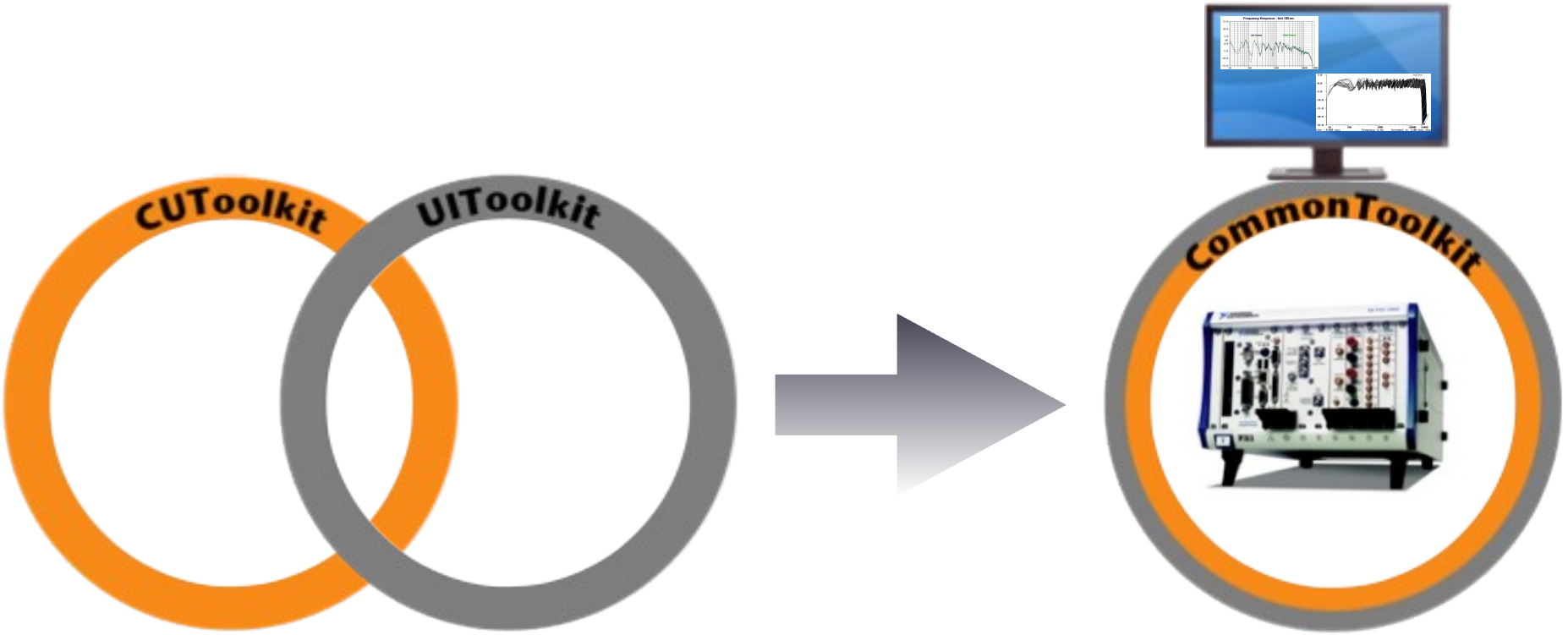
C ontrol system based on a
H ighly
A bstracted and
O pen
S tructure

LabVIEW integration

L. Foggetta – LAL/INFN

!CHAOS - SCHEME

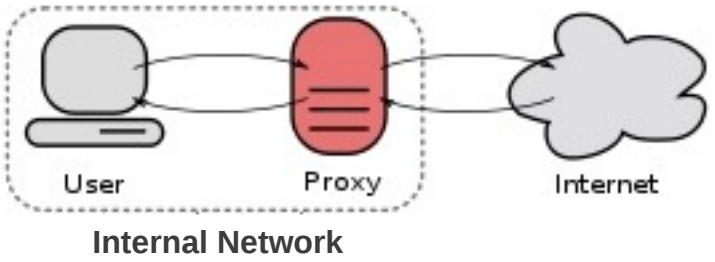


More Abstraction Layers => Very specialized frontends

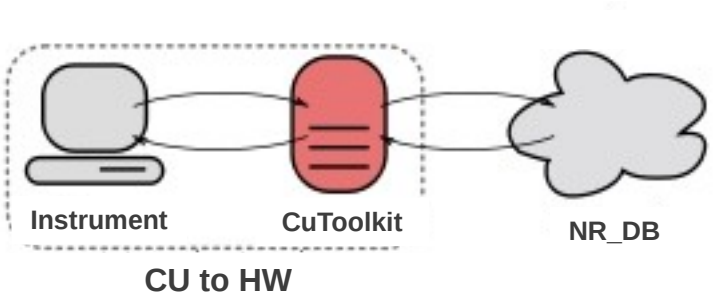
!CHAOS the dataflow starts and ends without knowing:
=> the generator
=> the type of processing that has been done
=> who's reading

<u>but knowing well who is the boss!</u>

CU acts as Forward Proxy



Forward Proxy => Internet



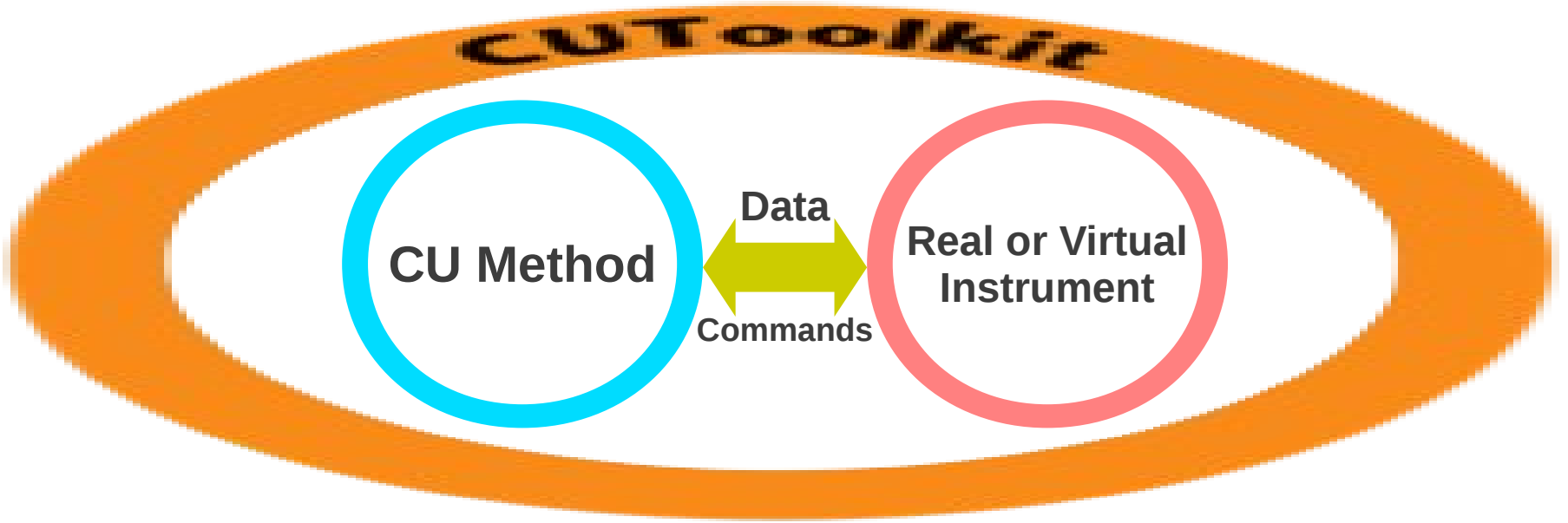
CU => !CHAOS

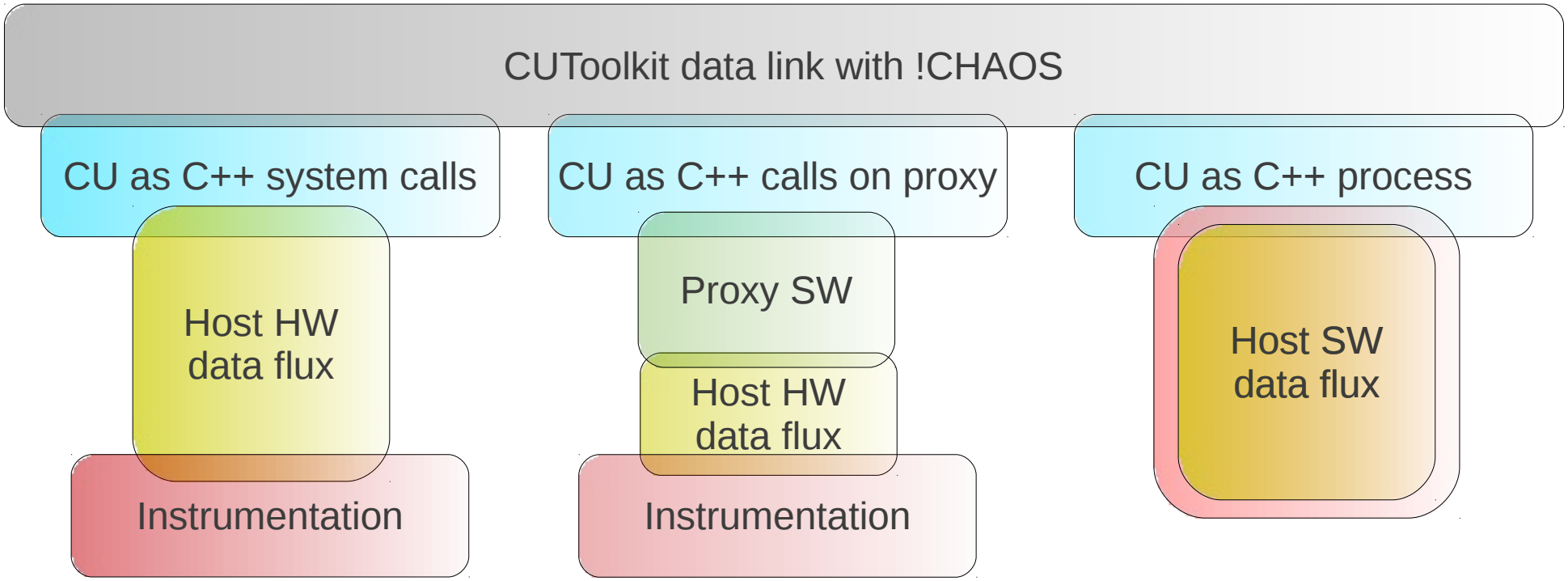
Instruments could be computational processes or real instrument, external or internal to the CUToolkit host.

Non Relational Database (NR_DB) acts as the whole memory of the CS system, where the CU puts the live data of their jobs

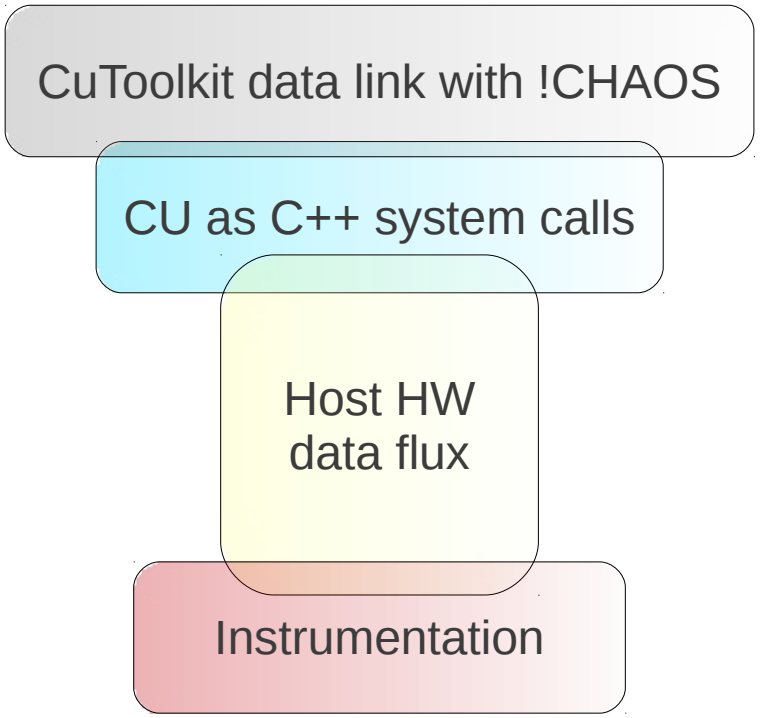
CUToolkit act as proxy layer to make a join with a local “world” to the more extensive one.

!CHAOS – CU type





CU Developing by C++ system calls



Like usual,
this development configuration needs of:

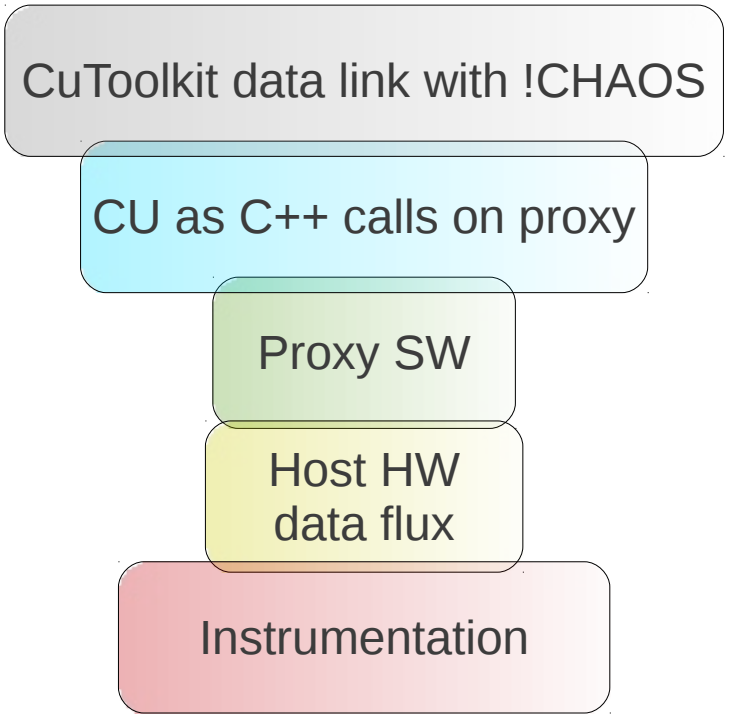
- Knowledge of low level programming
- Direct control of host system environment
- OS system-dependend developing
- OS system-dependend compiling
- Time for debugging
- Time for maintenance

Like usual *(if well done)*, this means:

- Higher data throughtput
- Best host system performances
- Fine tuning of the machine

Best for complex and custom HW subsystem

CU Developing by proxy calls



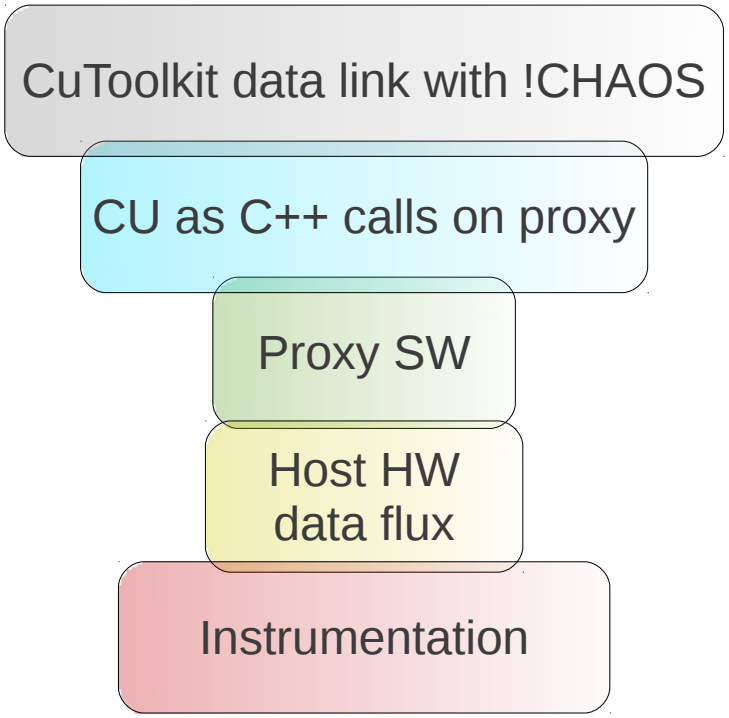
No knowledge of the host machine HW:

- But knowledge of high level proxy language
- Simpler HL control of instrumentation
- No OS system-dependend developing (~)
- No direct compiling
- Less Time for debugging
- Less Time for maintenance

Like usual *(**even** if well done)*, this means:

- Less data throughput
- Reduced host system performances
- Less fine tuning of the machine

CU Developing by proxy calls



No knowledge of the host machine HW:

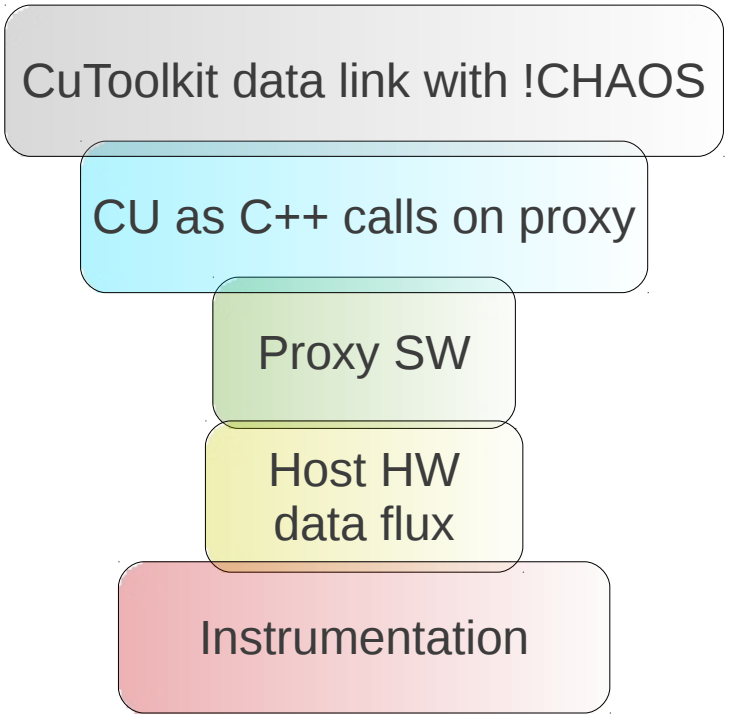
- But knowledge of high level proxy language
- Simpler HL control of instrumentation
- No OS system-dependend developing (~)
- No direct compiling
- Less Time for debugging
- Less Time for maintenance

Like usual *(**even** if well done)* this means:

- Less data throughput
- Reduced host system performances
- Less fine tuning of the machine

Proxy SW act as HW abstraction layer!

CU Developing by proxy calls



No knowledge of the host machine HW:

- But knowledge of high level proxy language
- Simpler HL control of instrumentation
- No OS system-dependend developing (~)
- No direct compiling
- Less Time for debugging
- Less Time for maintenance

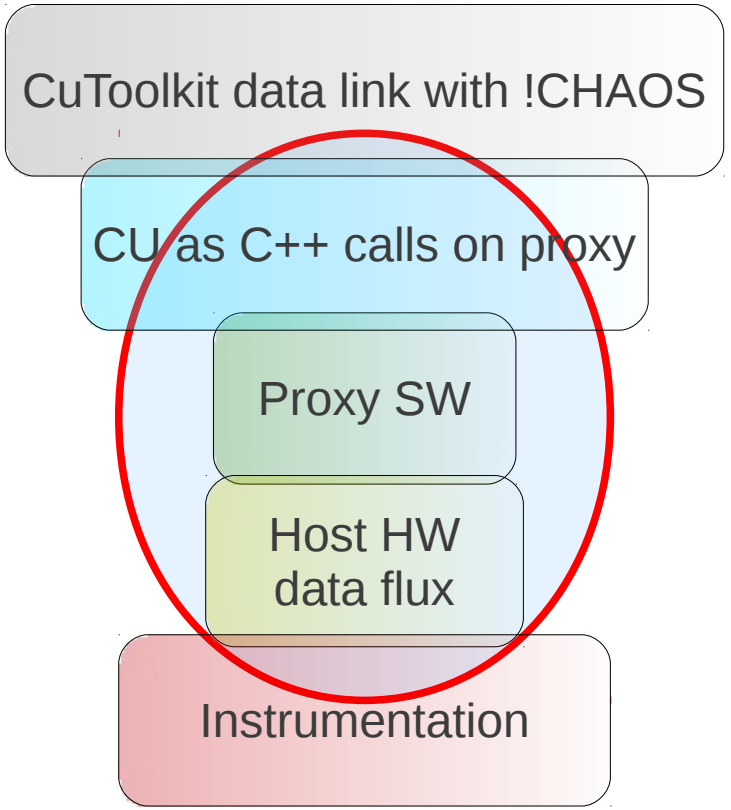
Like usual *(**even** if well done)* this means:

- Less data throughput
- Reduced host system performances
- Less fine tuning of the machine

Proxy SW act as HW abstraction layer!

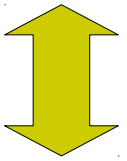
Best for huge number of simpler and standard instrumentation

CU – An in-depth glance



The most inner layer is when we use a proxy SW to join:

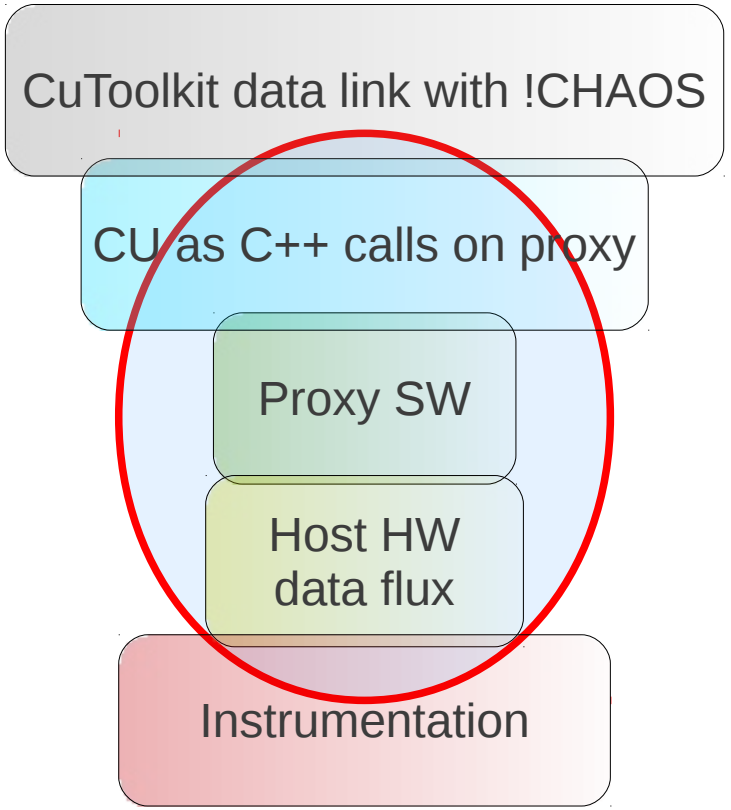
CUToolkit (!CHAOS)



HW (Host and external one)

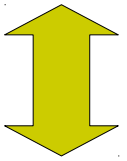
Proxy SW Layer has two connections:
=> C++ calls from CuToolkit to Proxy SW
<= Proxy calls to External HW via Host HW

CU – An in-depth glance



The most inner layer is when we use a proxy SW to join:

CUToolkit (!CHAOS)



HW (Host and external one)

Proxy SW Layer has two connections:
=> C++ calls from CuToolkit to Proxy SW
<= Proxy calls to External HW via Host HW

So what Proxy?

ONE POSSIBLE CHOICE

So what connections?

Some few word on “WHAT?”

Our needs are to keep the stuff:

- simple => reduce developing problems

- simple => fast develop

- simple => robust infrastructure

- simple => reliability

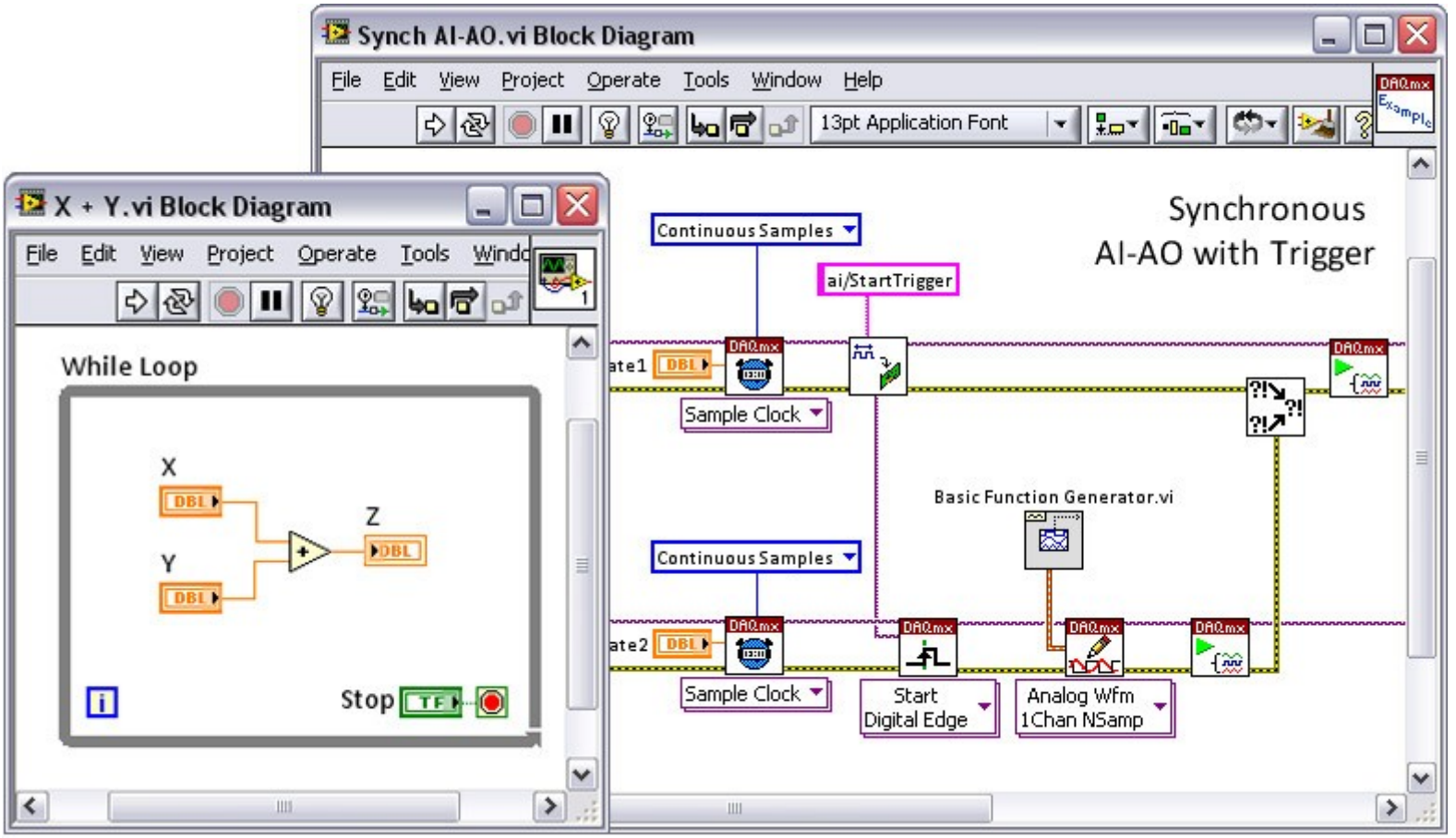
It seems that we need SIMPLE procedures and simple tools to specify (= make it real) an highly abstracted infrastructure

National Instrument LabVIEW

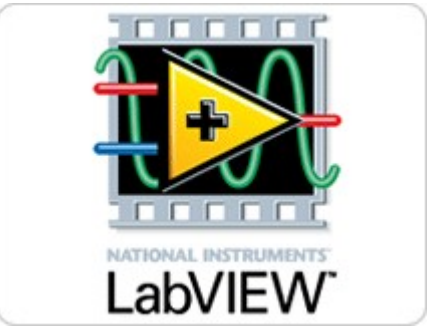
LabVIEW (from NI site, www.ni.com) is:

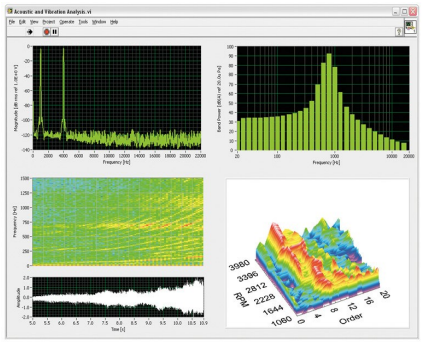
- *a **graphical programming environment** using intuitive graphical icons and wires that resemble a flowchart*
- used by **millions** of engineers and scientists
- *develop sophisticated **measurement**, test, and control systems*
- **integration with thousands of hardware devices** and provides hundreds of built-in libraries for advanced analysis and data visualization
- ***platform is scalable across multiple targets and OSs***
- since its introduction in **1986**, it has become an industry leader.

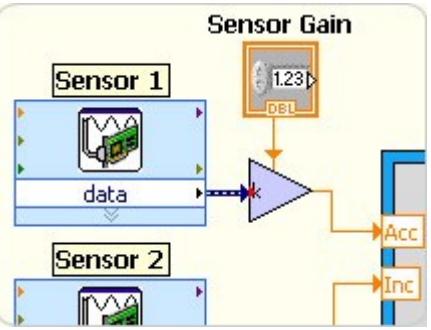
National Instrument LabVIEW



LV as PROXY SW?







→ **Tool for very fast, uniform developing**

→ **Multi platform with few differences => can be overcome**

→ **Not needs of low level programming experts**

→ **fast and reliable expert and prototype application could be implemented by “low level” LV tools**

→ **Easily learning by all**

→ **Fast and visual debugging and compilation**

→ **It owns a large field of typical engineering/physicist application and routines**

→ **It owns also a huge number of old and new hardware driver**

→ **It is widely used also for large project in big physics experiment**
- **DAΦne & SPARC CSs**
- **RADE project @ CERN**

→ **Also widely used by scientifical industrial partner involved in**

LV as PROXY SW?

→ Tool for very fast developing with huge community and supported hardware

→ Multi platform with few differences => can be overcome

→ Not needs of low level programming experts but fast and reliable expert and prototype application could be implemented by “low level” LV tools

→ Easily learning curve

→ Fast and visual debugging and compilation

→ It owns a large field of typical engineering/physicist application and routines

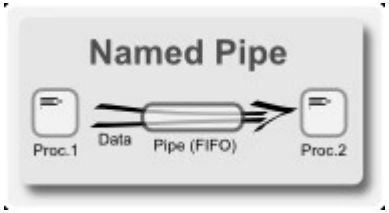
→ It owns also a huge number of old and new hardware driver

→ It is widely used also for large project in big physics experiment
- DAΦne CS
- RADE project @ CERN

→ Also widely used by scientifical industrial partner involved in

WHY NOT?

WHAT PROXY CONNECTIONS?



→ **"One of the fundamental features that makes Linux and other Unices useful is the pipe" (1997)**

→ **Multi platform => pipes are also on MAC and WIN**

→ **Pipes allow separate processes to communicate without having been designed explicitly to work together**

→ **CUToolkit | LV (but with all other software capable to streaming data)**

→ **NAMED Pipe are FIFO managed directly from the kernel within the host. It are directly embedded in every OS distribution.**

→ **Named pipes are very simple to use.**

→ **mkfifo is a thread-safe function, no synchronization mechanism is needed when using named pipes**

→ **Write (using write function call) to a named pipe is guaranteed to be atomic**

→**Obviously, LV has pipe tools..**

LV_CU – First word of CHAOS

Our implementation of LV was intended on following three roads:

- make a LV shared object (or dll) loadable from CU, but needs OS-type dependent call and tricky compilation.

LV_CU – First word of CHAOS

Our implementation of LV was intended on following three roads:

- ~~make a LV shared object (or dll), but needs OS-type dependent call and tricky compilation.~~

- make the CUToolkit as shared object (or dll) loadable from the LV, but some logical conflict start to Arise

LV_CU – First word of CHAOS

Our implementation of LV was intended on following three roads:

- ~~make a LV shared object (or dll), but needs OS-type dependent call and tricky compilation.~~

- ~~make the CUToolkit as shared object (or dll) loadable from the LV, but some logical conflict start to arise~~

- make a CU executable (LV_CU) with calls via pipe toward a LV executable that make the dirty job with the HW.
  => CUToolkit owns only simple calls to deliver BSON DS
  => LV gets DS, setup HW, run HW, collects data in DS

LV_CU – First word of CHAOS

Our implementation of LV was intended on following three roads:

- ~~make a LV sha~~ ~~ds OS-type dependent c~~

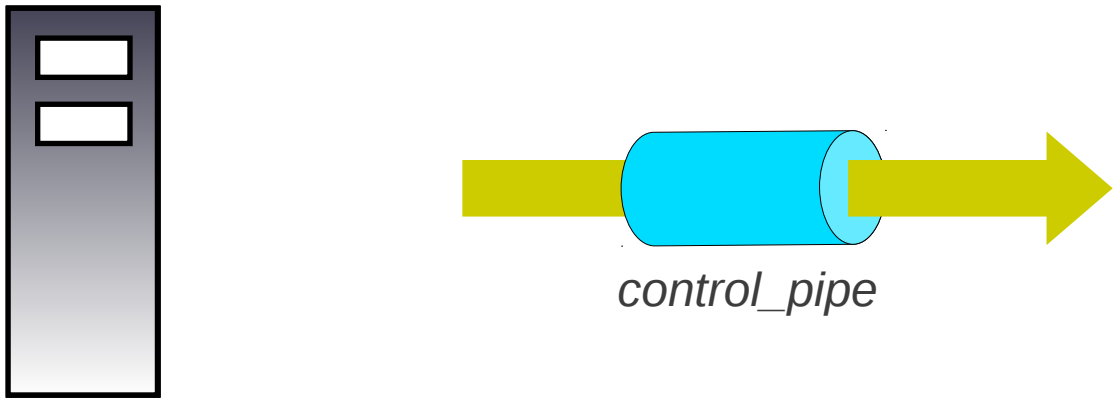
- ~~make the CUT~~ ~~dll) loadable from the LV,~~ ~~art to arise~~

- make a CU executable (LV_CU) with calls via pipe toward a LV executable that make the dirty job with the HW.
  => CUToolkit owns only simple calls to deliver BSON DS
  => LV gets DS, setup HW, run HW, collects data in DS

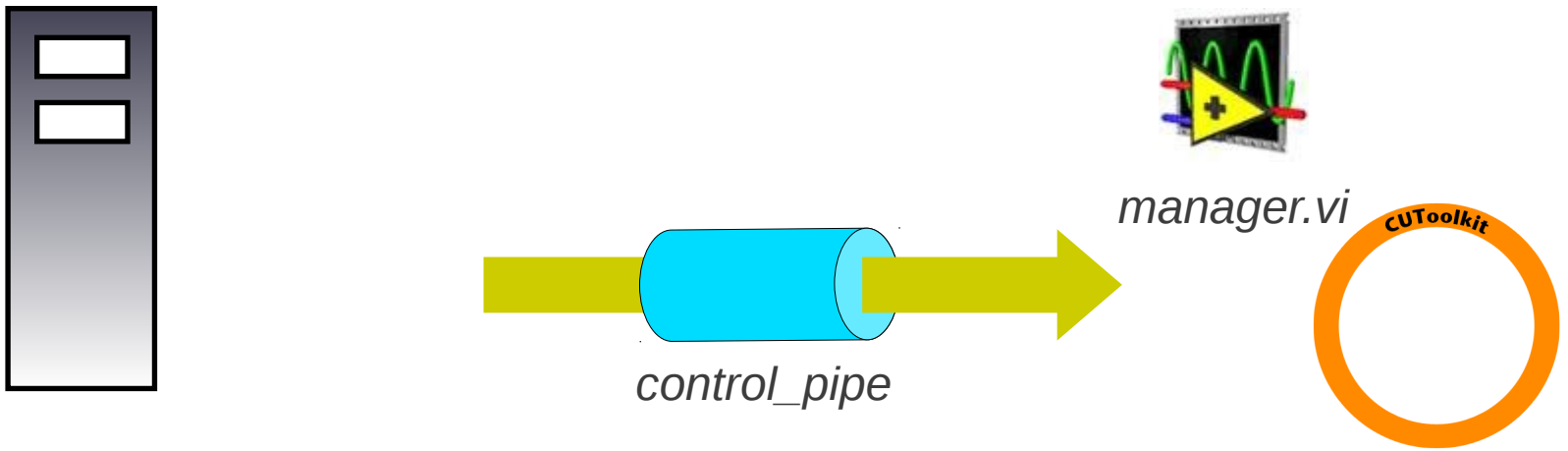
LV_CU – INIT PHASE

INIT PHASE

- Host opens a named pipe used for the communications from CUToolkit to a "unique host pipe manager" via unique pipe



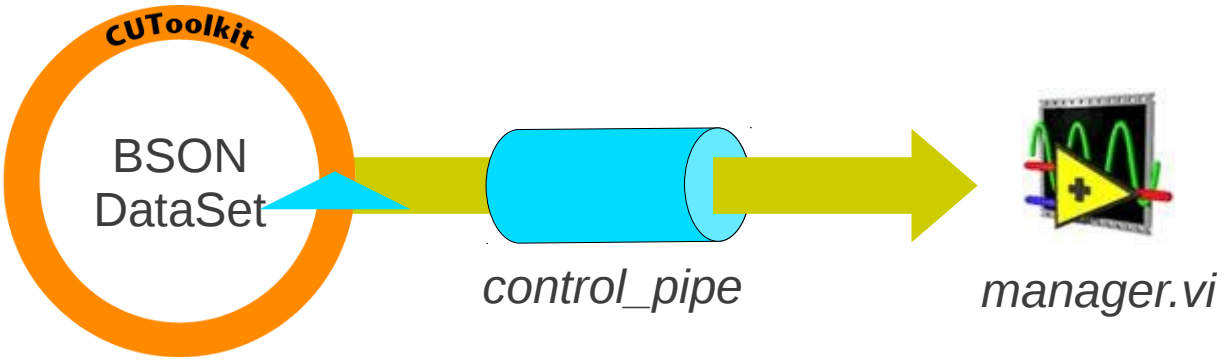
- Also starts a pipe manager VI template and CUToolkit



LV_CU – CONTROL PHASE

CONTROL PHASE

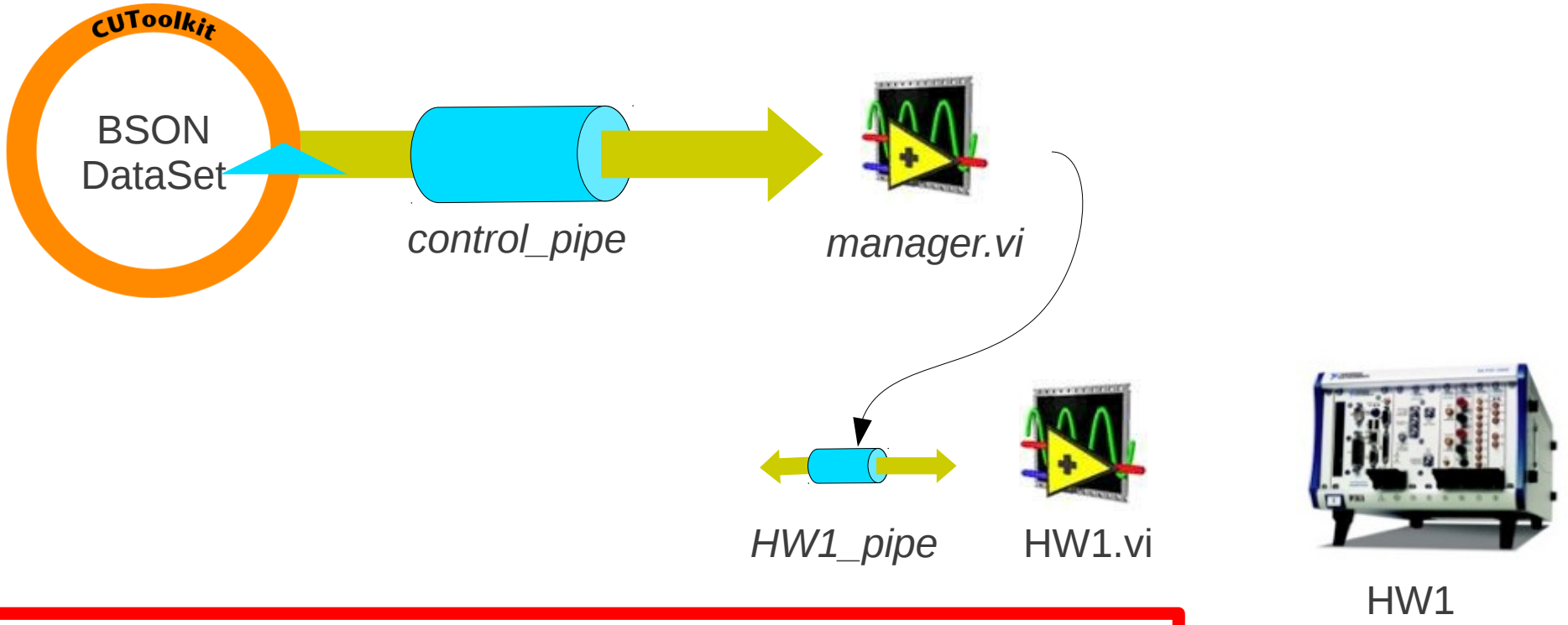
- !CHAOS send to CUToolkit to instantiate a LV_CU to control some HW via BSON DS



LV_CU – CONTROL PHASE

CONTROL PHASE 1

- LV_CU passes BSON to manager.vi that opens and launch setup HW template and bidir named pipe

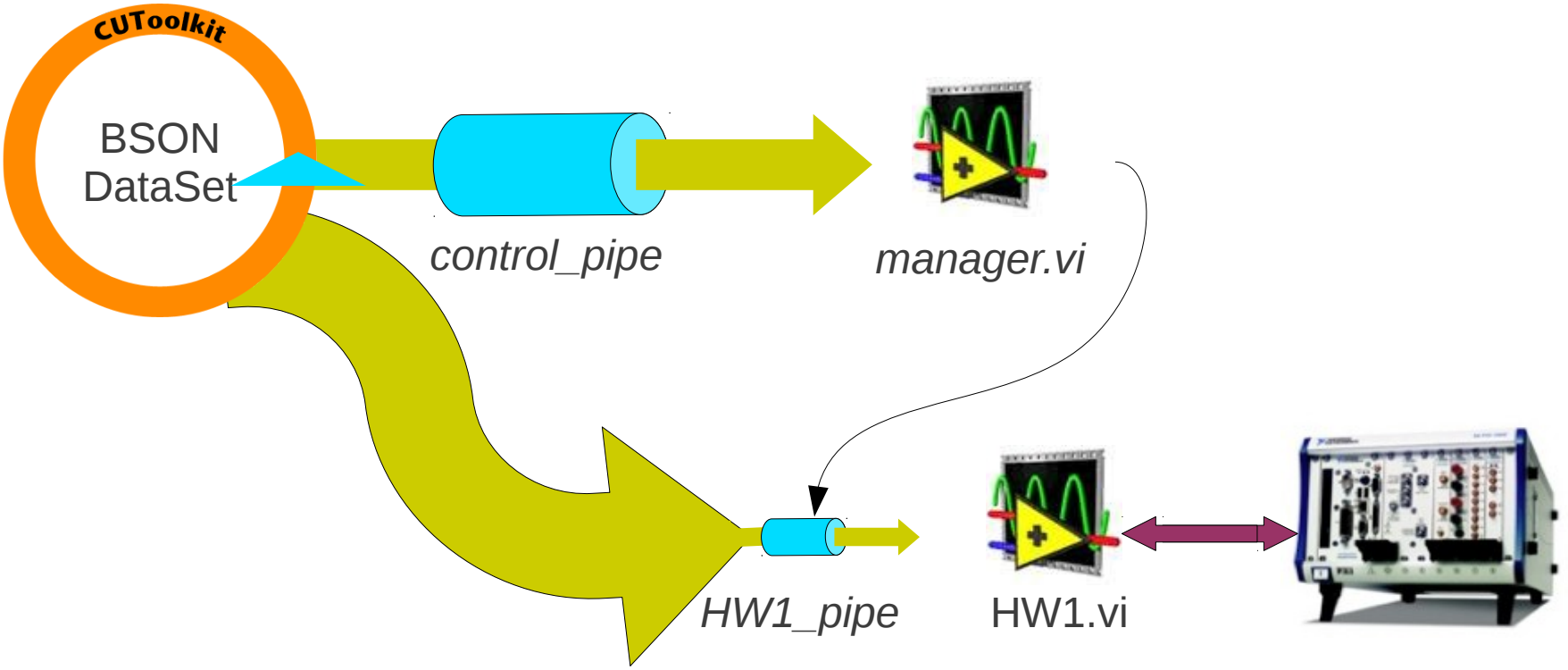


- LV_CU is detached from HW startup procs
- LV_CU runs every time like the whole system is up and ready

LV_CU – SPECIALIZATION PHASE

SPECIALIZATION PHASE 1

- LV_CU takes direct control of HW1_pipe, after HW1.vi starts and runs
- LV_CU now specialize itself to a specified HW, calling the driver that matching the BSON data

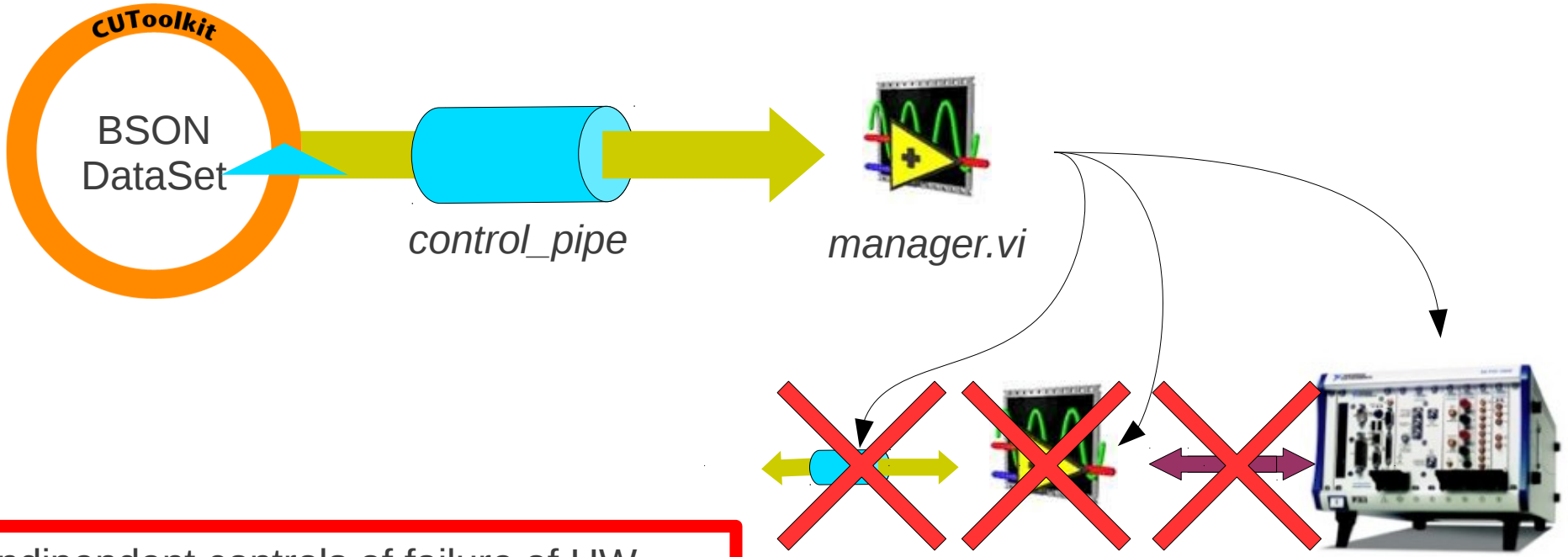


- manager.vi acts as local watchdog (alive)

LV_CU – DEINIT PHASE

DEINIT PHASE

- LV_CU will close all the HW dependent comms and process by manager.vi



- indipendent controls of failure of HW and SW related to it

- LV_CU still alive during closing procs

HW.vi template

Our goal is to create device-class indipendent type of template VI where some simple subprocesses do all the jobs, letting the **DYNAMICALLY CALLED** driver to specify the LV_CU.

HW.vi SUBPROCESSES:

BSON coding and decoding SP

HW's pipe reading and writing BSON data packet: none of them is processor
compsuming (processing on demand)

DISPATCHER SP

Take the input dataset (decoded above) and select the CU method-like subVI to run

METHOD SP

It contains dynamic subVI's calls to put in run machine processes, store temporary data, data manipulation

What we use to do so?

- **LabVIEW QUEUE** to buffer BSON DS

- **LabVIEW VARIANT** polymorphic structure to match the flexibility to store data and managing dynamically created data since one structure is different from each DS

- **LOW LEVEL LabVIEW's vi tool** to increase the whole system portability in the future development of LABVIEW

- **Custom Pipe's CIN** to match blocking feature and to be non OS dependent

- **Custom BSON decode and code** VI and CIN to improve the VARIANT processing speed (un-flatten_to_BSON like)

WHAT remains to the LV_CU developer?

Since nowadays we intend to use only six elementary CU method:

- defineActionAndDataset
- setDatasetAttribute
- init
- deinit
- run
- stop

The LV developer has to develop only six cases, each of those will be called by the DISPATCHER subprocess,
no matter what is !CHAOS but understanding well what DS is.

SUMMARY of devel

- Prototype of the template of LV_CU
- !CHOAS to HW init, startup, run, deinit procedures
- HW to !CHAOS delivering in MemCached
- Custom LV tools for BSON
- Custom LV tool for MongoDB and Memcached
- Some pieces (intermediate level) of DAΦNE CS VME memory as been diverted to Memcached in the LV environment to improve performances and test stability
- Some successful attempts in displaying data with LV directly from Memcached