Advanced Scicos, Kepler and Simulink integration

www.scicoslab.org

www.scicos.org

https://kepler-project.org/

Scicos code generation: a very short introduction

Why generating automatically codes for the target?

Because I'm a lazy man with very poor programming skills.

Because life is too short to waste time to write and debug program.

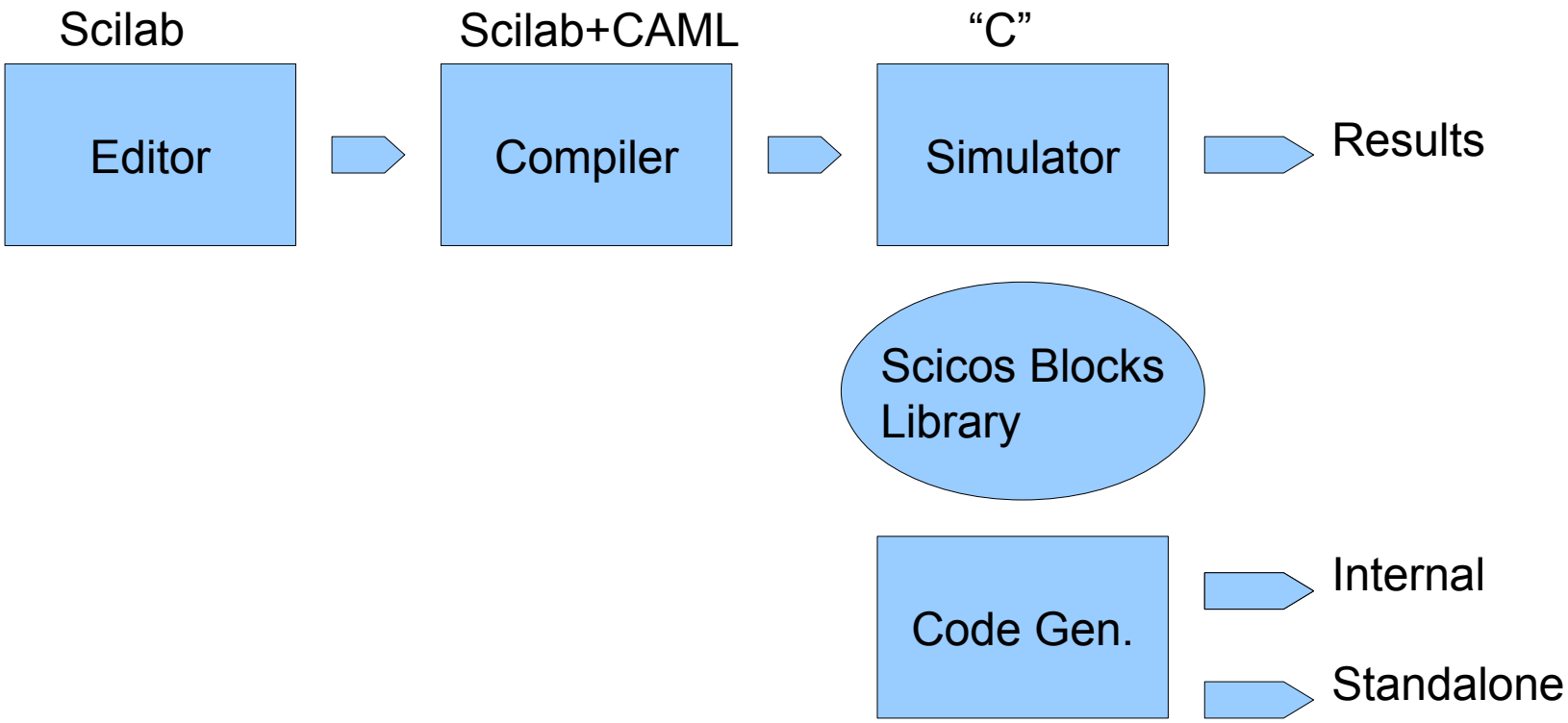
Because I want to sell my product NOW !

Because I have ZERO tolerance on bugs.

Because I believe that a good simulation is very close to reality.

Because .....

Scicos architecture



Scicos block : how does it work ?

Interfacing function: the Scicos block “user's interface”.
A Scilab script that is launched when you “double click” over a Scicos block.

Computational function: the Scicos block simulation function.
The code (typically a C function compiled as shared library) called during the simulation.

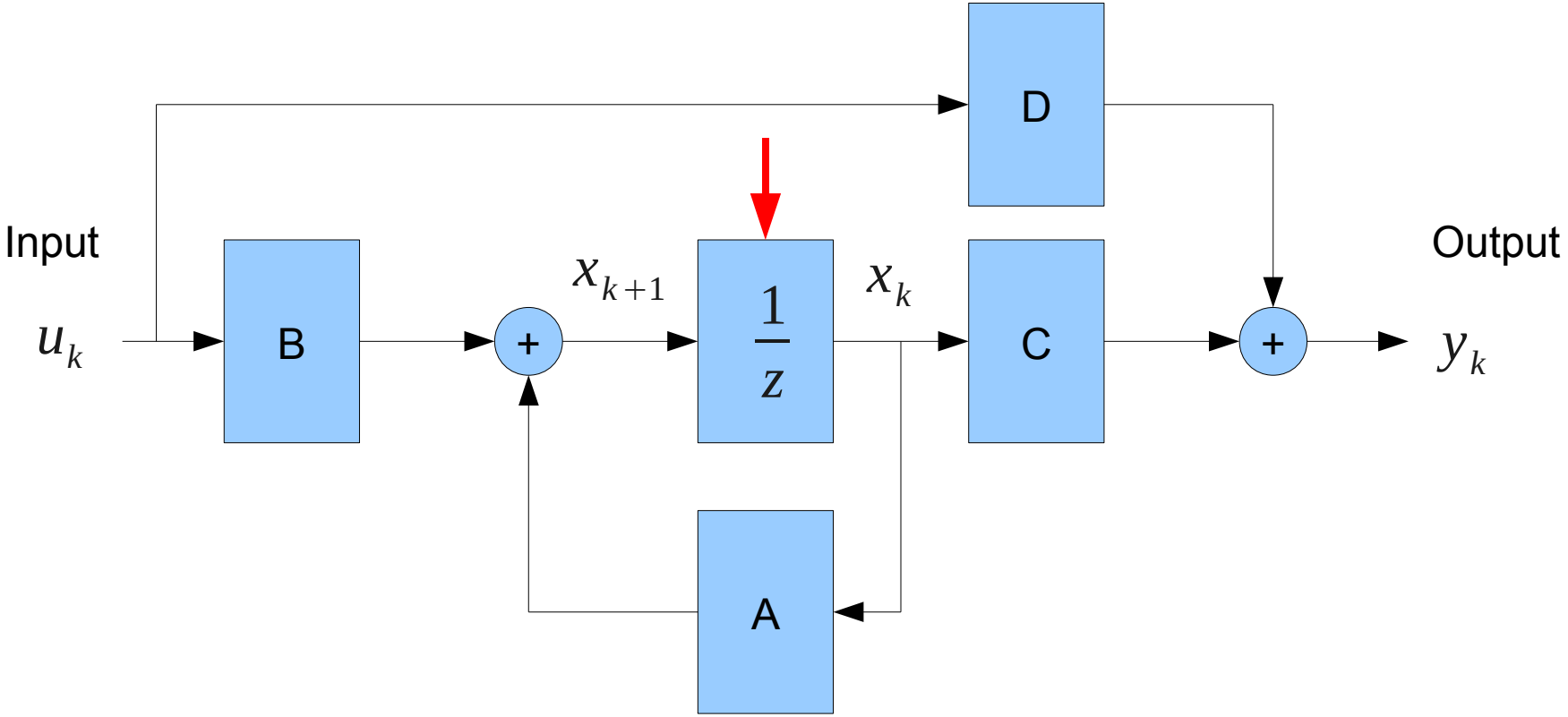
Scicos block computational function

```
#include <windows.h>          /* Compiler's include files's */
#include "scicos_block4.h"    /* Specific for Scicos block development */
#include "machine.h"

void custom_bock(scicos_block *block, int flag)
{
  //** scicos_block is a “C” complex data structure that contains in/out ports parameters and values, block's parameters and states

switch(flag) {

     case Init: //** It is called just ONE TIME before simulation start. Put your initialization code here
     break;

     case StateUpdate: //** It is called EACH CYCLE. Read the input ports and update the internal state of the block
                       //** Use this section for OUTPUT blocks (e.g. D/A converter, digital output, etc.)
     break;

     case OutputUpdate://** It is called EACH CYCLE. Read the internal state and update the output
                         //** Use this section for INPUT block (e.g. A/D converter, digital input, etc.)
     break;

     case Ending://** It is called just ONE TIME at simulation end. Put your “shut down” code here.
     break;

  } // close the switch

} // close the computational function
```

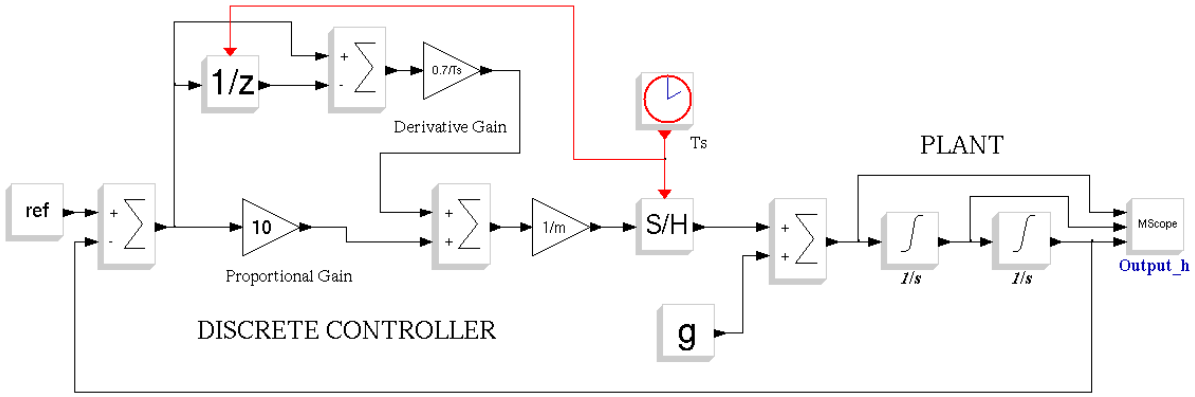
The origin of Scicos computational function

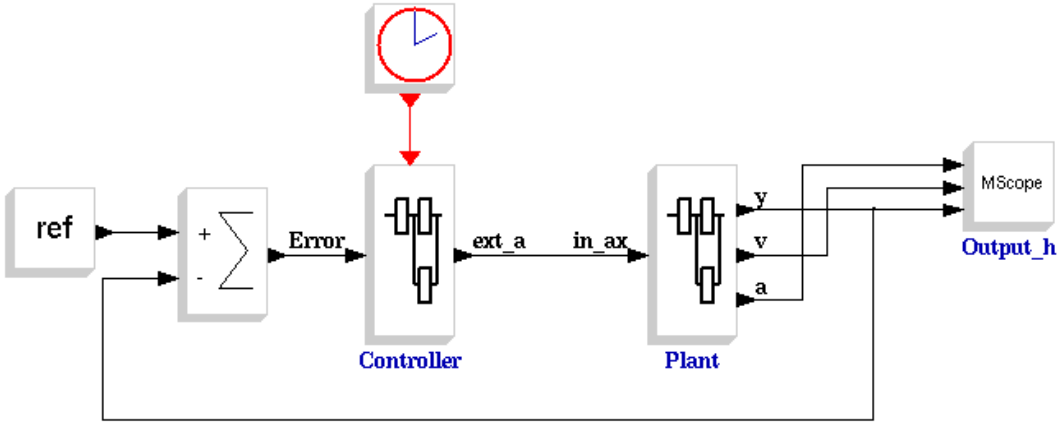
$$y_k = C\,x_k + D\,u_k\,;\,OutputUpdate$$

$$x_{k+1} = A\,x_k + B\,u_k\,;\,StateUpdate$$



Scicos code generation: PD controller example

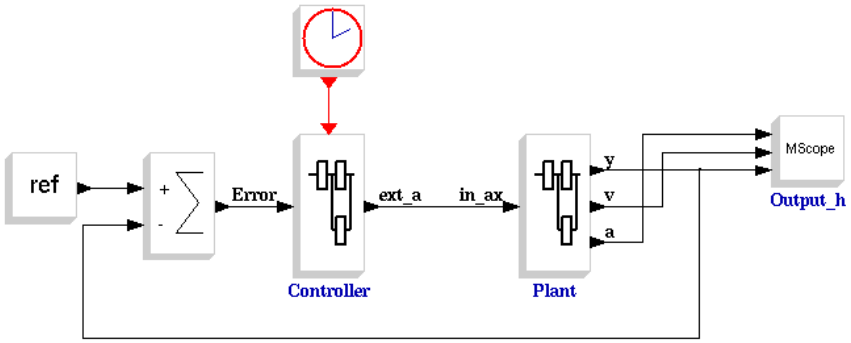


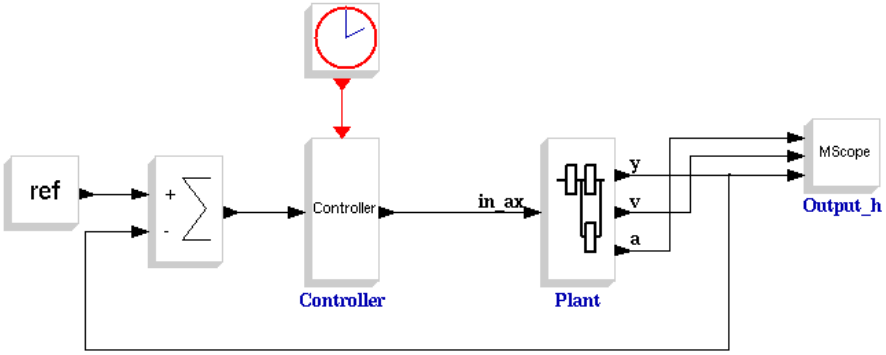


Super blocks help a lot to simplify and organize complex diagrams

(apple_hc_03.cos, apple_hc_05.cos)

Scicos code generation: PD controller example





Before and after code generation

(apple_hc_05.cos, apple_hc_06.cos)

Scicos code generation and rapid prototyping

Two types of HIL mode

Basically, there are two ways to implement HIL with Scicos:

Interpreted: the simulation runs as usual BUT the user activates the “real time” option inside the simulation's control panel. Scicos-HIL. (DEMO+VIDEO)

Standalone: you generate a “C” code and you compile it for the target platform. You run the code on the target and you recover the data using specific Scicos blocks (Scicos-RTAI, Scicos-FLEX). (VIDEO)

Scicos-HIL

What is "Hardware In the Loop" ?

HIL means that part of your system is "virtual" e.g. running on a suitable computer. The "virtual section" is connected to the real, physical, system using A/D (sensors) and D/A (actuators) interfaces;

Why do we need Hardware In the Loop ?

Because HIL is a very effective technique for model validation and controller tuning. Do you want to spend your life debugging low level codes ?

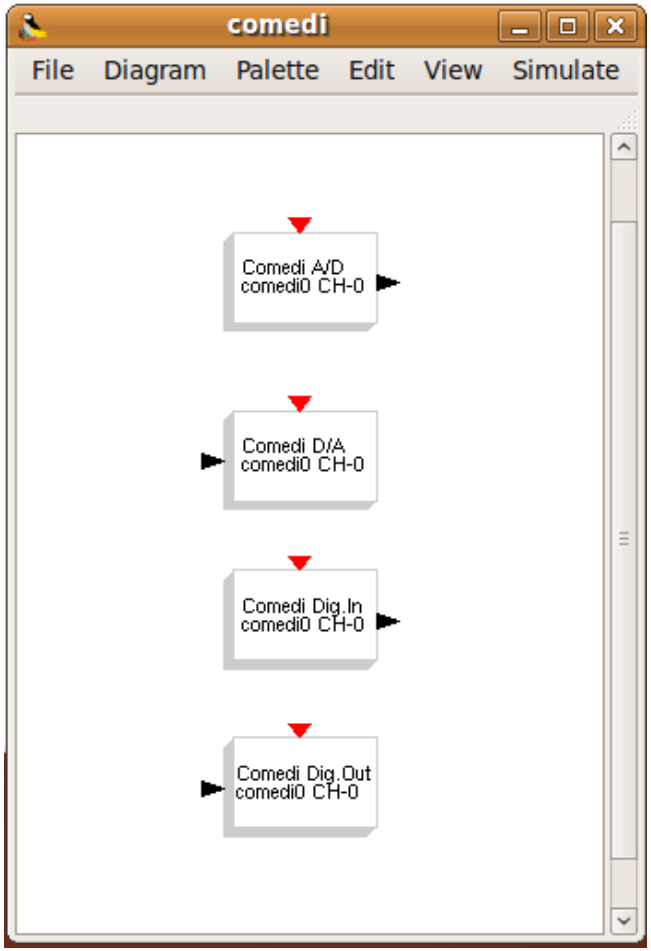
What is necessary to implement HIL ?

You need a simulator with real time capability and I/O interfaces support.

Scicos-HIL : Hardware In the Loop

In its base form Scicos-HIL is constituted by four blocks

- Analog Input
- Analog Output
- Digital Input
- Digital Output



Device driver support for Scicos

Open Source: Comedi (www.comedi.org) is the only available option for a complete OS solution (GLP2 license). Comedi covers the most used data acquisition cards available on the market.

Proprietary. Unfortunately, some manufactures provide neither detailed technical specifications of their cards nor an open source driver. From ScicosLab standpoint it is not a problem, because the only thing that you really need is a shared library (*.dll or *.so).

Custom. You can develop custom driver or use “direct access” code inside a Scicos block. If you develop “direct access” blocks you need to run Scicos (ScicosLab) as a “root” user.

ScicosLab includes real time support ?

Yes, of course.

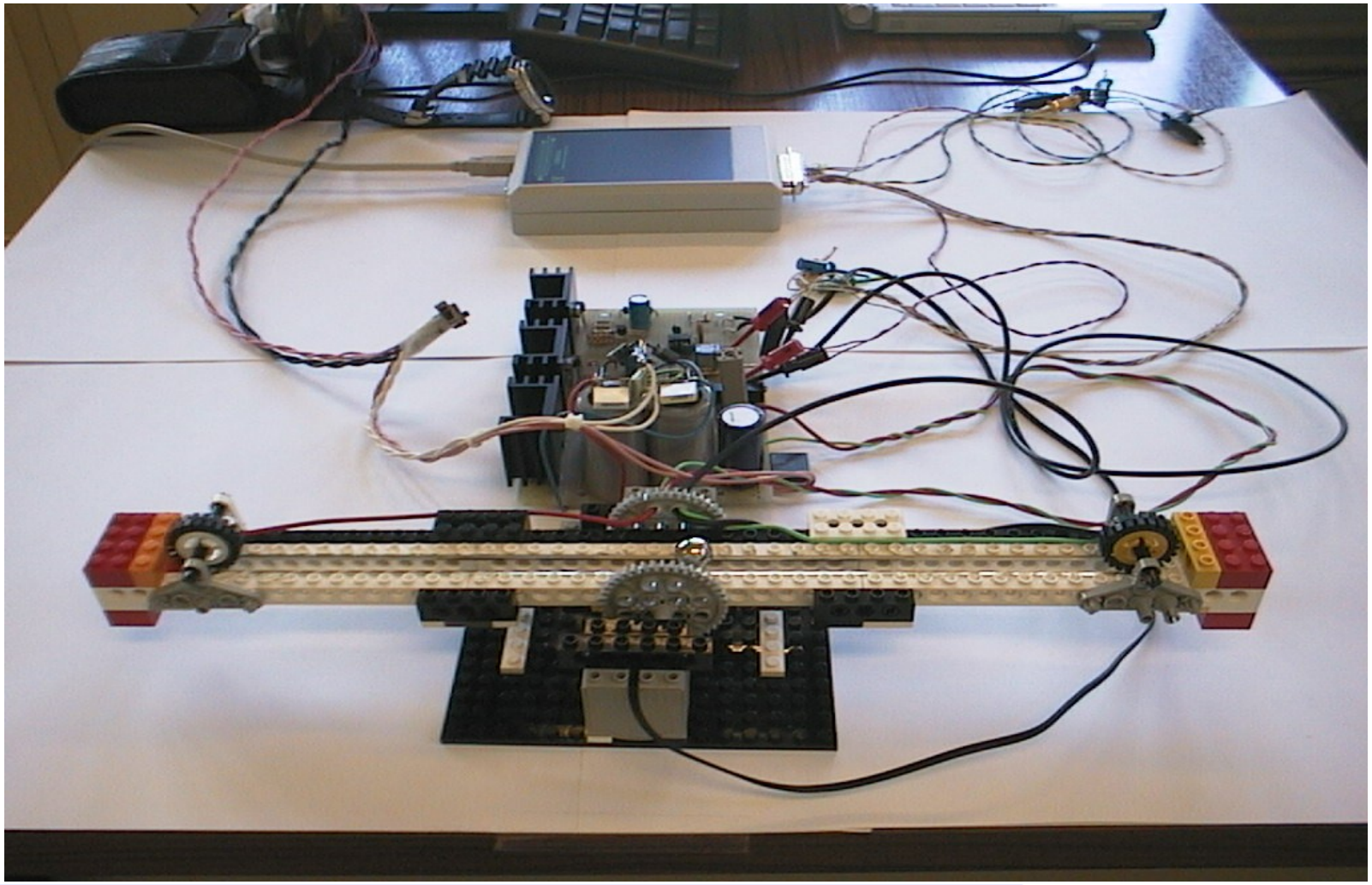
The quality of service is very operating system dependent.

Windows does not offer guarantee about quality of service. The minimum sampling time is around 20ms.

Recent Linux kernels are 95% "*soft*" real time up to 1.0 ms sampling time.

There are many "hard" real time Linux versions/patches (RTAI, Xenomai, RT_PREEMPT). ScicosLab could be easily modified in order to use the RT services available. For the maximum compatibility we suggest the POSIX compliant API offered by RT_PREEMPT.

Ball and beam experiment with ScicosLab



Ball and beam: the MODEL

"We choose the ball and beam, not because it is easy, but because it is hard".

Why it is so hard to control ? Just look at the model ...

$$\dot{\omega} = \frac{mgx}{J_b + mx^2}\cos(\theta) - \frac{K_V K_C^2 K_T}{R_A\left(J_b + mx^2\right)}\omega + \frac{K_C K_T}{R_A\left(J_b + mx^2\right)}U_M$$

$$\dot{\theta} = \omega$$

$$\dot{v} = \frac{5}{7}g\sin(\theta)$$

$$\dot{x} = v$$

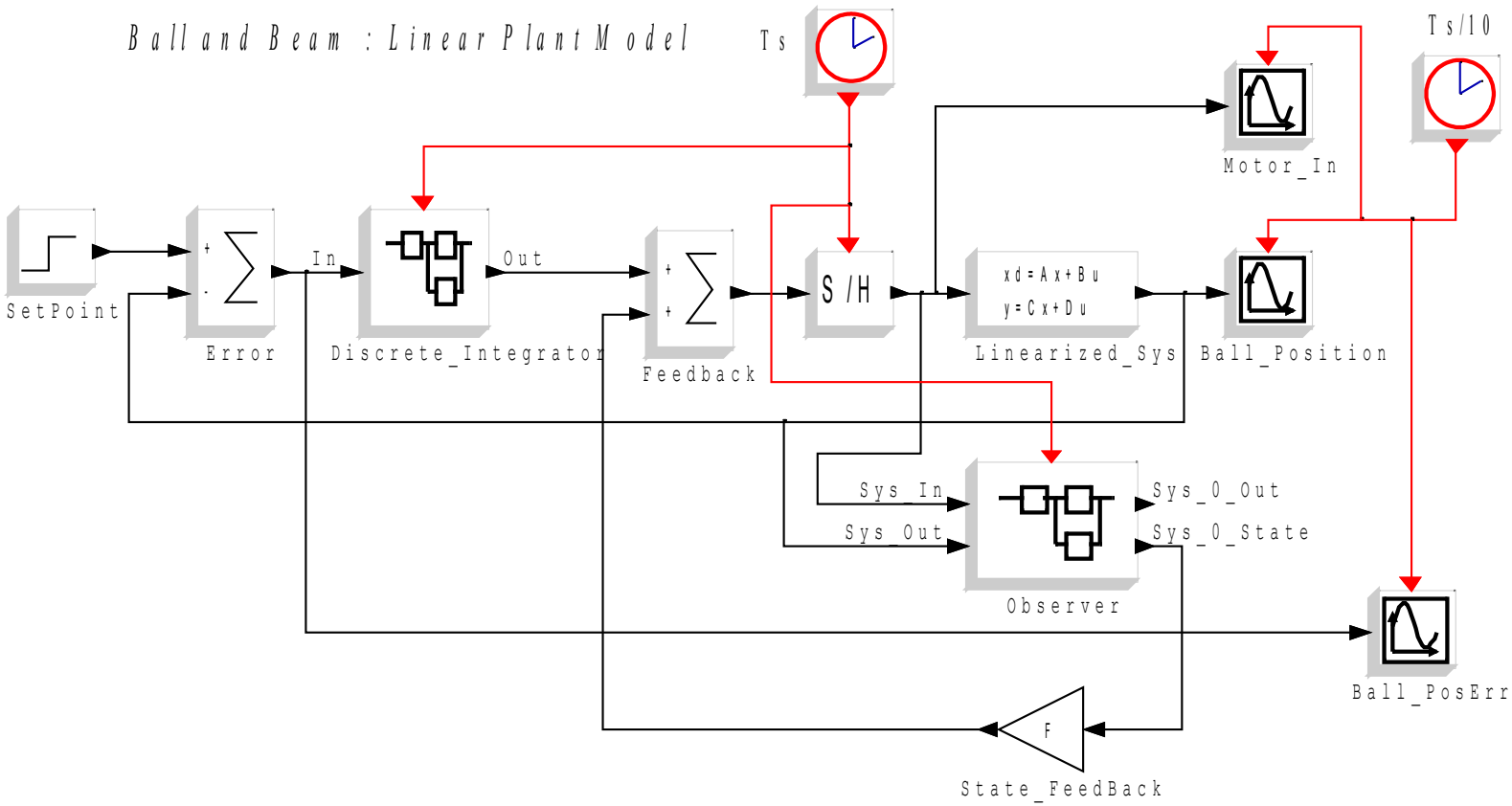
- *Non linear (in many ways)*
- *Unstable*
- *Complex (4th order)*

Ball and beam experiment challenge

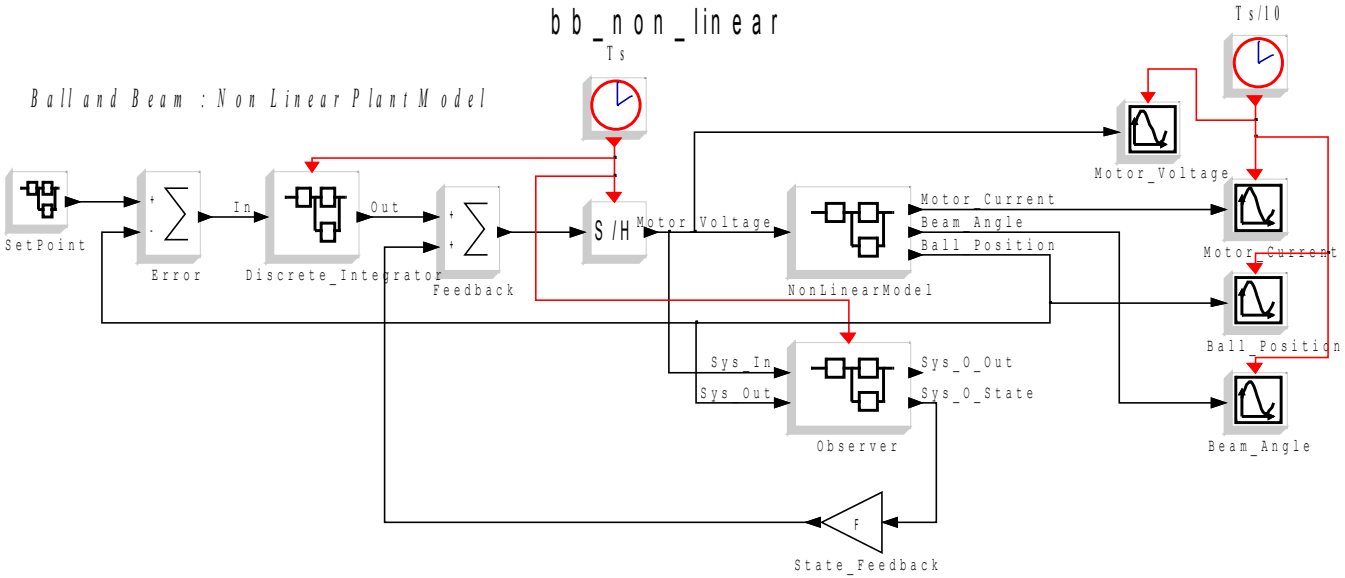
Additional difficulties:

- Cheap (low performances) easy to find LEGO components; low cost DAQ card (USB Dux) usable also on laptop PCs
- Full state LQR digital feedback controller
- ONLY one sensor (ball position)
- We need an state OBSERVER in order to recreate the full system's state
- We are not satisfied of the accuracy of a simple state feedback: we want zero error in the ball position (steady state). We add an additional digital integrator in the position feedback loop.

Ball and beam experiment: linear model

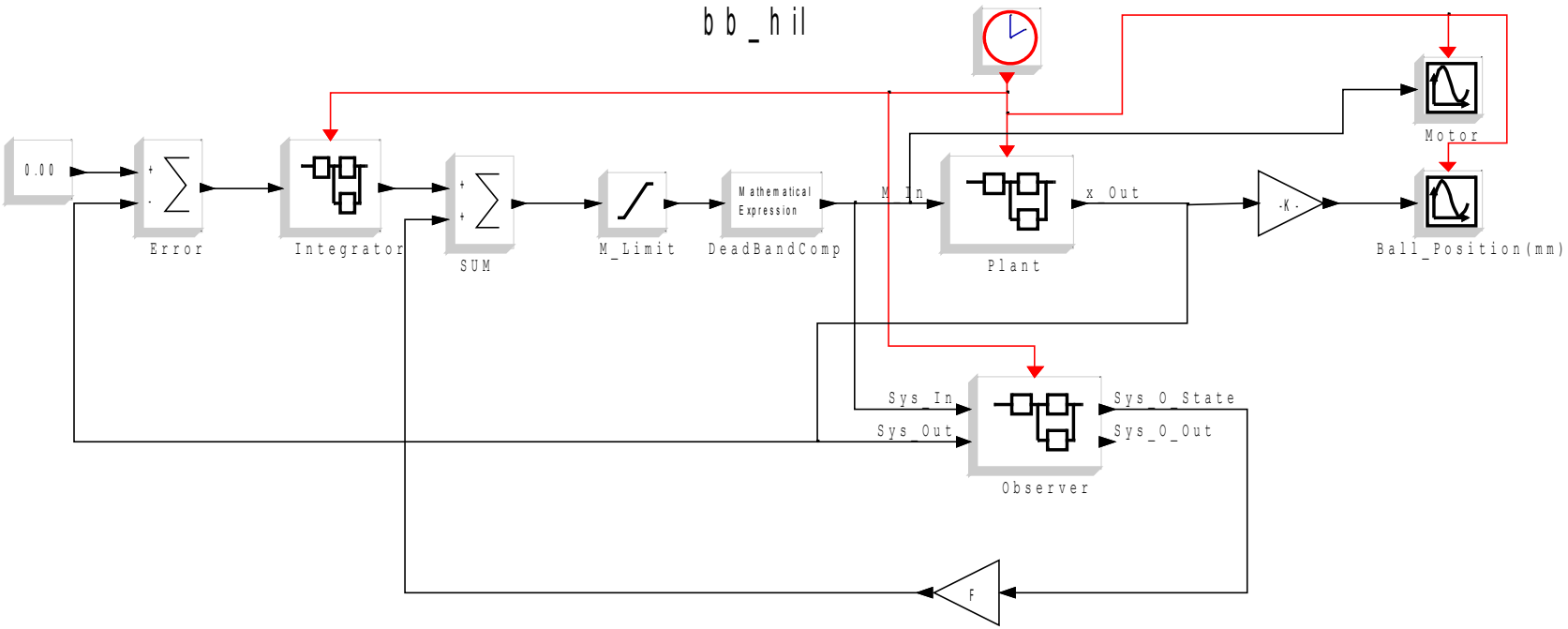


Ball and beam experiment: non linear model

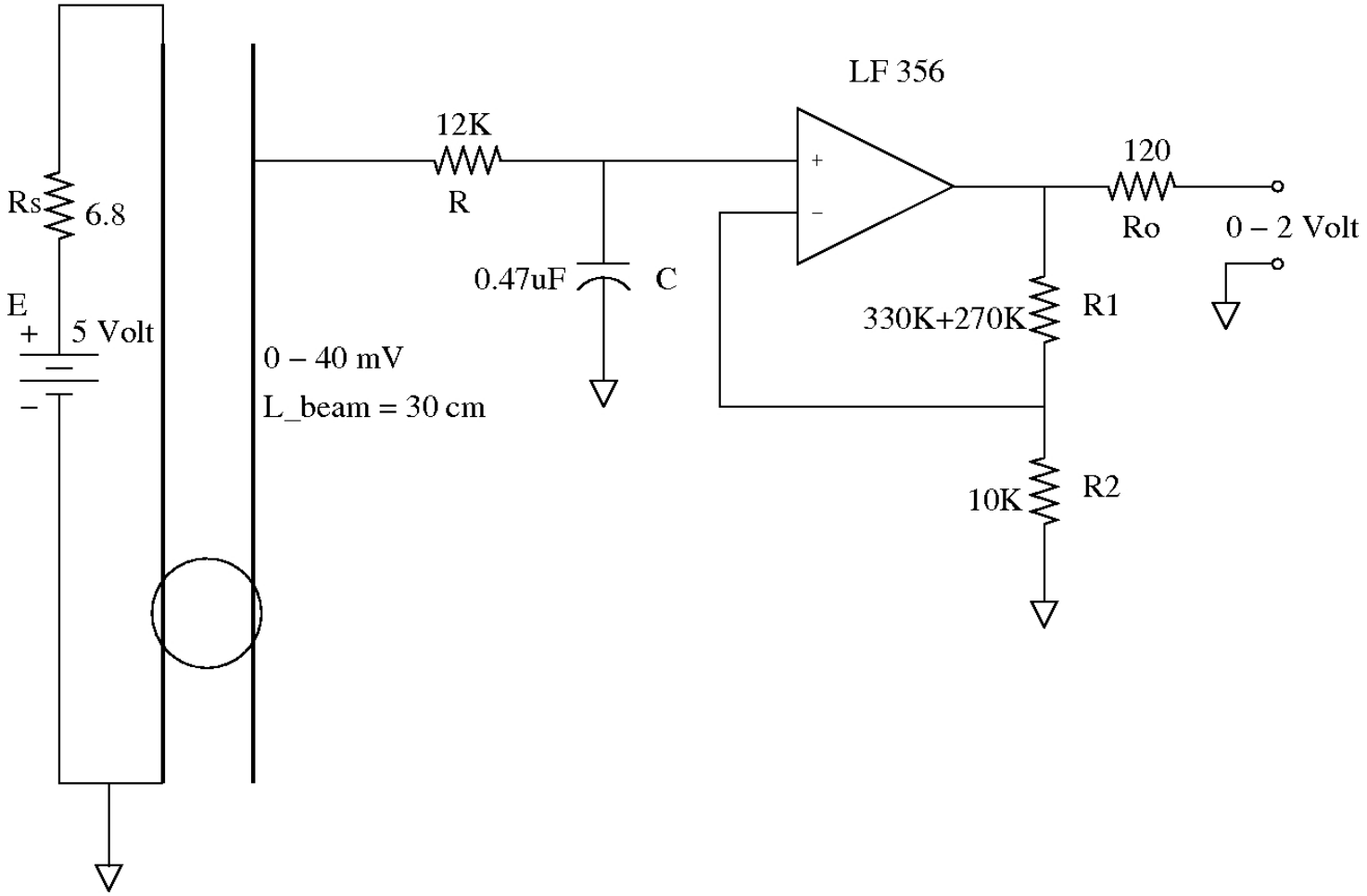



```
/* [ x_dot v_dot theta_dot omega_dot ]  ;  [ x v theta omega]  */

block->xd[0] =    v ;
block->xd[1] =    5.0/7.0*g*sin(theta) ;
block->xd[2] =    omega ;
block->xd[3] =    (m*g*x)/(Jb+m*x*x)*cos(theta)-
                  (Kv*Kc*Kc*Kt)/(Ra*(Jb+m*x*x))*omega +
                  (Kc*Kt)/(Ra*(Jb+m*x*x))*Um ;
```

Hardware In the Loop experiment with ScicosLab



Ball position sensor

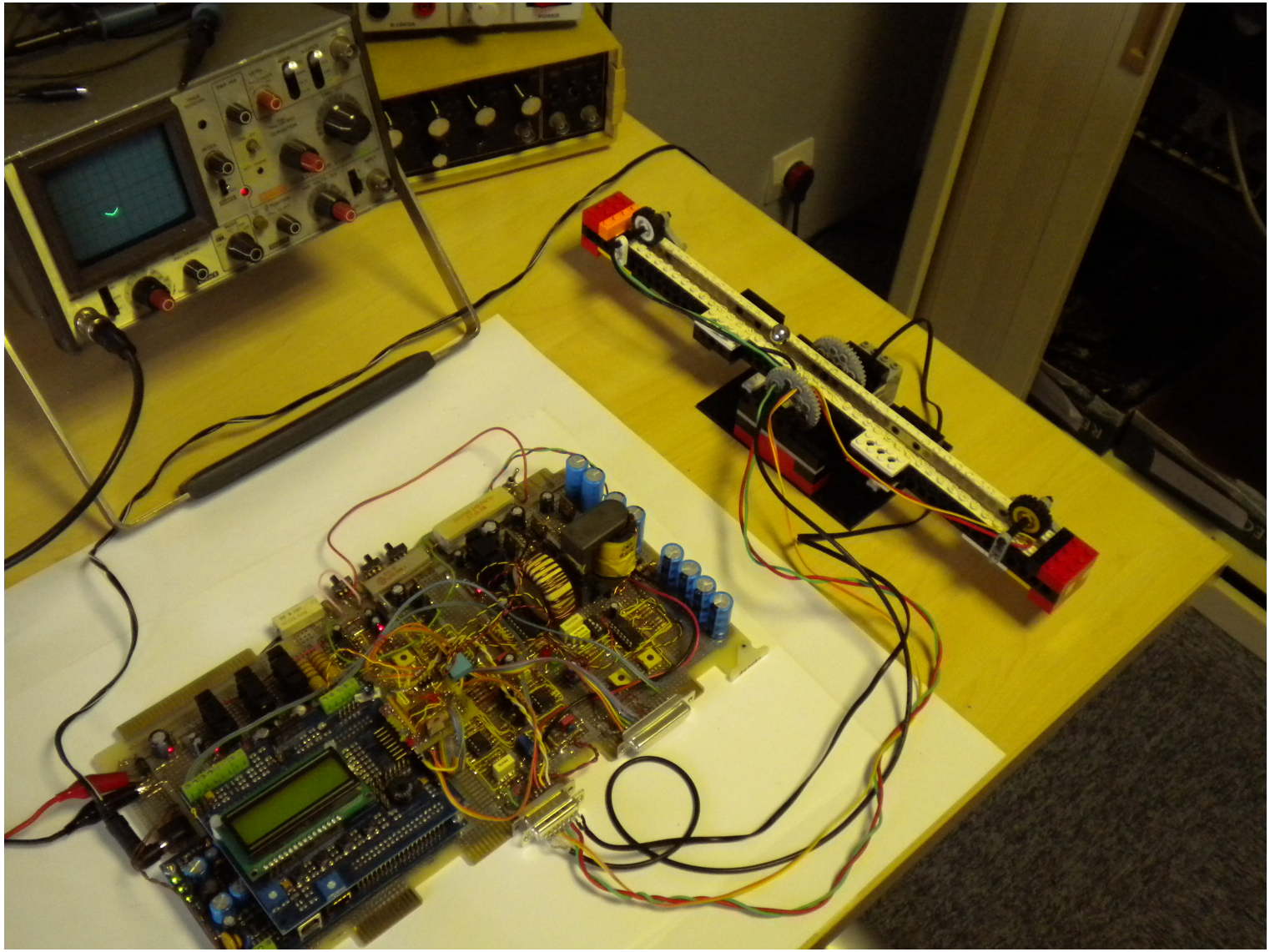


Standalone HIL mode

We have developed two ways to implement HIL in standalone mode:

Scicos-RTAI: the code is generated from a Scicos diagram (super block) and compiled for a Linux RTAI target (usually x86 type). The compiled code runs as RTAI task (in user space). You can interact with the task using RTAI-Lab (see next slide).
(www.rtai.org)

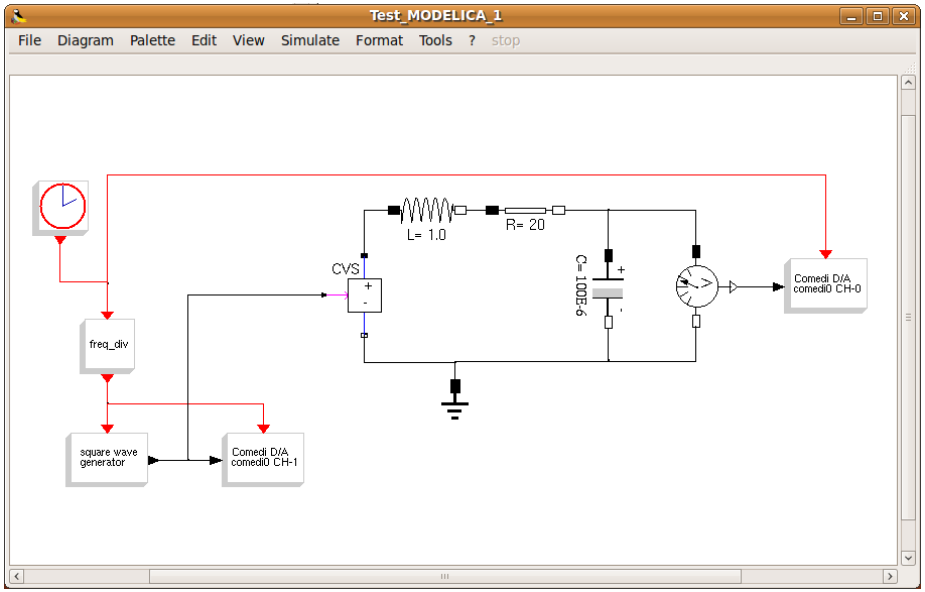
Scicos-FLEX: the code is generated from a Scicos diagram (superblock) and cross-compiled for a specific target (Microchip DSPIC). The code is "flashed" in the chip. You can interact with the task using specific Scicos blocks and USB communication.
(http://www.evidence.eu.com/content/view/175/216)

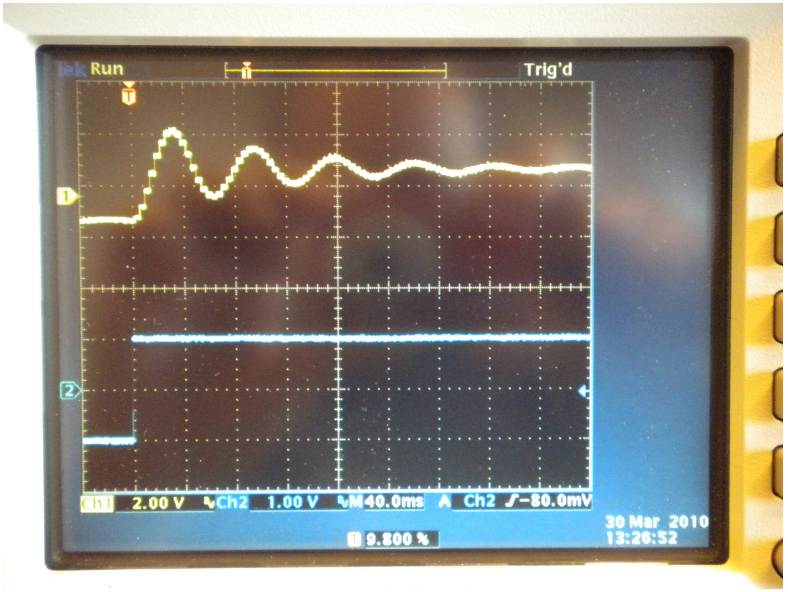


Modelica and Scicos-HIL

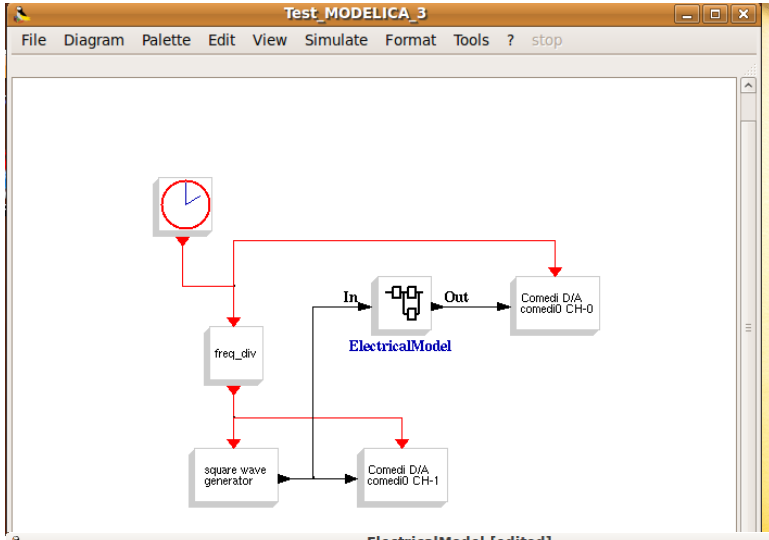
Within some hardware and software (operating system) limitation you can run a Modelica simulation in real time and interface it with real signals using Scicos-HIL.

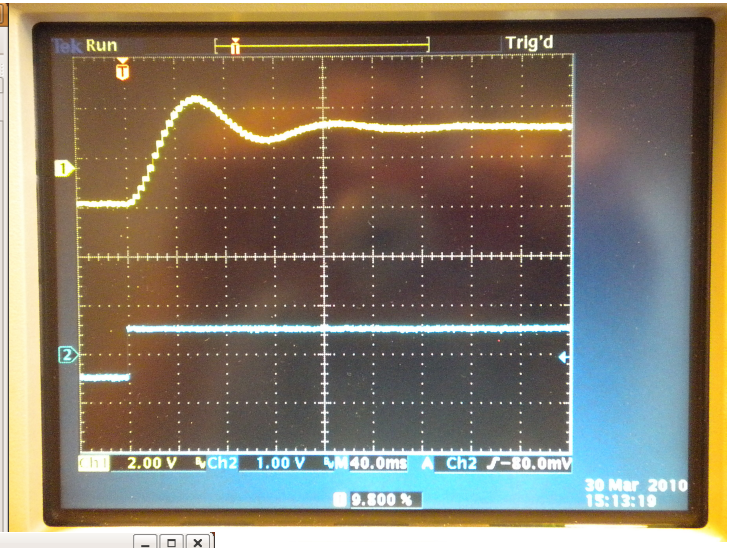
A simple RCL circuit is simulated and the input and output signals are visualized using a real scope connected at the D/A outputs of a data acquisition card.

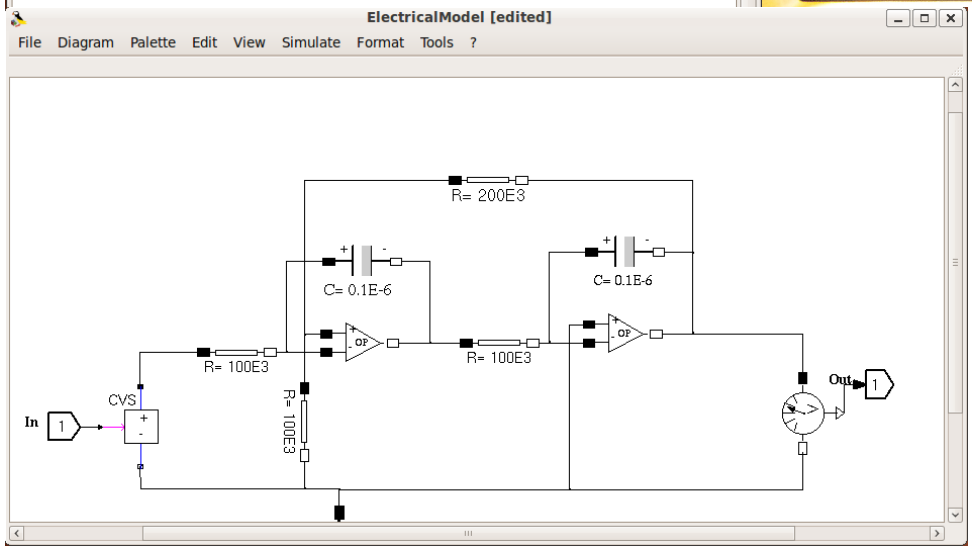




As the previous example, but using an electrical circuits that uses op-amp.





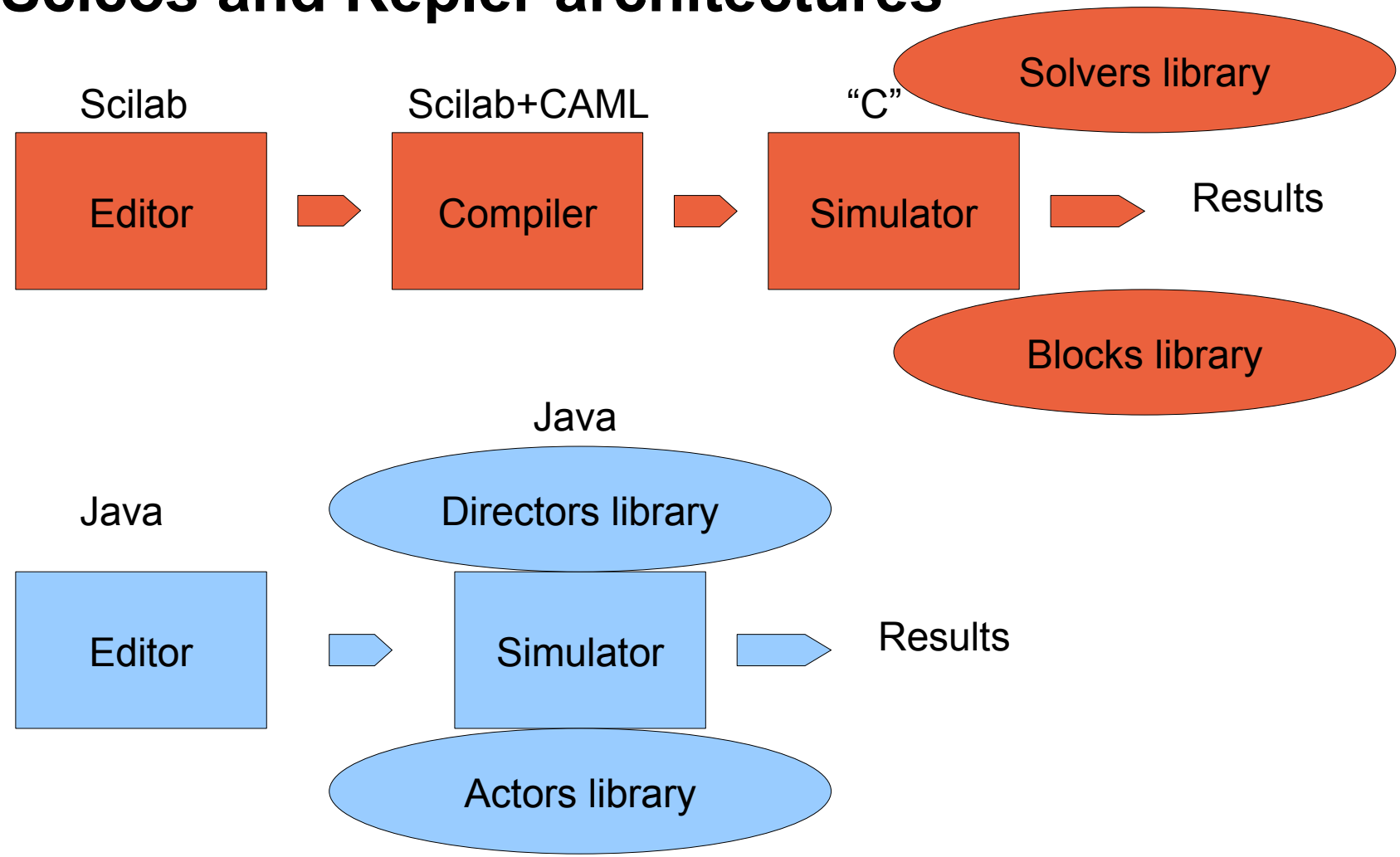


This circuit is simple to realize than the previous one because it uses standard components of reasonable values.

Hybrid systems

Hybrid systems: internal and external signals are a mix of sampled and continuous time signals.

Scicos and Kepler architectures

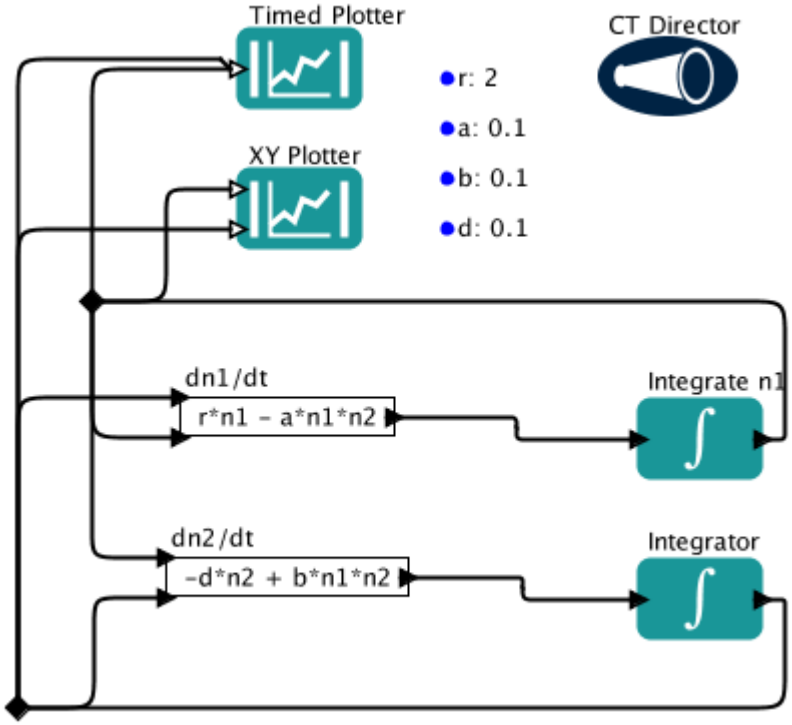


Simulink, Scicos and Kepler architectures: different names, same s**t

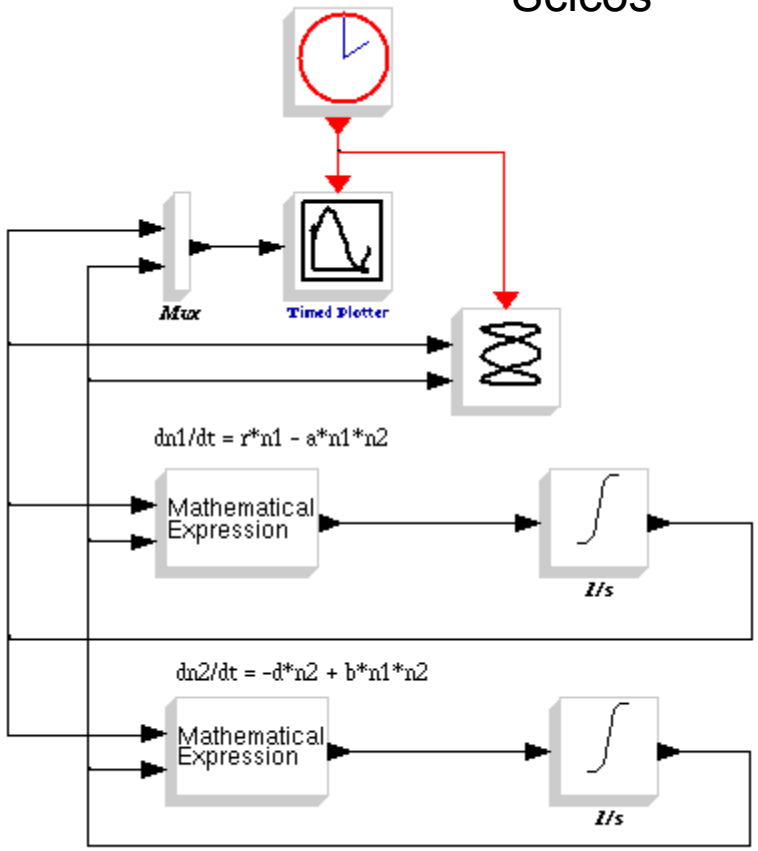
	Simulink	Scicos	Kepler
Main entity	Diagram	Diagram	Work flow
Atomic entity	Block (C)	Block (C, Scilab)	Actor (Java)
Sub assembly	SubDiagram	SuperBlock	Composite Actor
Connection	Link (line)	Link	Relation
Script language	Matlab (*.m)	Scilab (*.sci)	Not Available
Code Generation	Real Time Workshop	Scicos Code Generators	Not available

Scicos and Kepler: same simulation, same s**t

Kepler



Scicos



Kepler and Scicos simulation engines (continuous)

The *CT Director* is designed to oversee workflows that predict how systems evolve as a continuous function of time (i.e., "dynamic systems").

The continuous time section is handled by the Scicos simulator using solvers (solvers library). Scicos uses different solvers in according to the specific problem. Usually Scicos choose automatically the right solver and the default parameters works most of the time.

Notice that the solver parameters in Scicos are the same of Kepler.

Same parameters, same ....

Kepler and Scicos simulation engines (discrete)

The *SDF Director* is often used to oversee fairly simple, sequential workflows. Types of workflows that run well under an *SDF Director* include processing and reformatting data, converting one data type to another, and reading and plotting a series of data points.

The *DE Director* is often used for modeling time-oriented systems: queuing systems, communication networks, and occurrence rates or wait times.

The *DDF Director* is often used for workflows that use looping or branching or other control structures, but that do not require parallel processing (in which case a PN Director should be used).

The “discrete Scicos directors are built in inside the simulator.
This “monolithic” architecture was the standard at the time.
Both Scicos and Simulink use the very same approach.

Scicos block / Kepler actor

A Scicos block is composed by two separate functions:

the interfacing function:
Scicos block “user's interface”.
A Scilab script that is launched when you “double click” over a Scicos block;

the computational function:
Scicos block simulation function.
The code (typically a C function compiled as shared library) called during the simulation.

A Kepler actor is a monolithic Java code that include the user interface code (“double click”) and the code (methods) used for the simulation. A Kepler actor is capable to host an external C/C++/ FORTRAN code with an appropriate Java wrapper.

FC2K do this job for you :-).

Scicos block computational function (simplified)

```
#include <windows.h>          /* Compiler's include files's */
#include "scicos_block4.h"    /* Specific for Scicos block development */
#include "machine.h"

void custom_bock(scicos_block *block, int flag)
{
  //** scicos_block is a “C” complex data structure that contains in/out ports parameters and values, block's parameters and states

switch(flag) {

     case Init: //** It is called just ONE TIME before simulation start. Put your initialization code here
     break;

     case StateUpdate: //** It is called EACH CYCLE. Read the input ports and update the internal state of the block
                       //** Use this section for OUTPUT blocks (e.g. D/A converter, digital output, etc.)
     break;

     case OutputUpdate://** It is called EACH CYCLE. Read the internal state and update the output
                         //** Use this section for INPUT block (e.g. A/D converter, digital input, etc.)
     break;

     case Ending://** It is called just ONE TIME at simulation end. Put your “shut down” code here.
     break;

  } // close the switch

} // close the computational function
```

Kepler actor Java skeleton (simplified)

```
/* HelloWorld actor for getting used to PtolemyII/Kepler concepts.*/
public class HelloWorld extends TypedAtomicActor {

/* This is the implementation of a HelloWorld actor. This actor outputs "Hello World!" as string from its output port.  */
    public HelloWorld(CompositeEntity container, String name) throws NameDuplicationException, IllegalActionException  {
    }
    ////                ports and parameters                ////
    public TypedIOPort output = null;
    ////                    public methods                  ////

   public void preinitialize() throws IllegalActionException { //** Set port types and/or scheduling information.
                //** The preinitialize() method is only invoked once per workflow execution and is invoked before any of the other action methods.
   }

   public void initialize() throws IllegalActionException { //**Initialize local variables and begin execution of the actor. }

   public void prefirefire() throws IllegalActionException { //**Determine whether firing should proceed. This method is invoked each time the actor is
                    //**  fired, before the actor is fired. The method can also be used to perform an operation that will happen exactly once per iteration.}

   public void fire() throws IllegalActionException { //** Read actor inputs and current parameter values, and produce outputs. }

   public void postfire() throws IllegalActionException { //** Determine if actor execution is complete, schedule the next firing (if appropriate) and
                                                          //**  update the actor's persistent state.  }

   public void wrapUp() throws IllegalActionException {//** Display final results. The wrapUp() method is only invoked once
                                                          //** per workflow execution. }
}
```

Scicos and Kepler computational sections (for discrete time applications)

	Scicos	**Kepler**	*Notes*
Initialization	Init()	initialize()	There is a one-to-one correspondence
Run time (periodic)	OuputUpdate() StateUpdate()	fire()	Inside our code generator we have merged the the two calls in a single function (called <*name*>*_isr()*) that summarize the job.
Simulation ends	End()	wrapUp()	There is a one-to-one correspondence

How ScicosLab-ITM (Scicos-ITM) was born

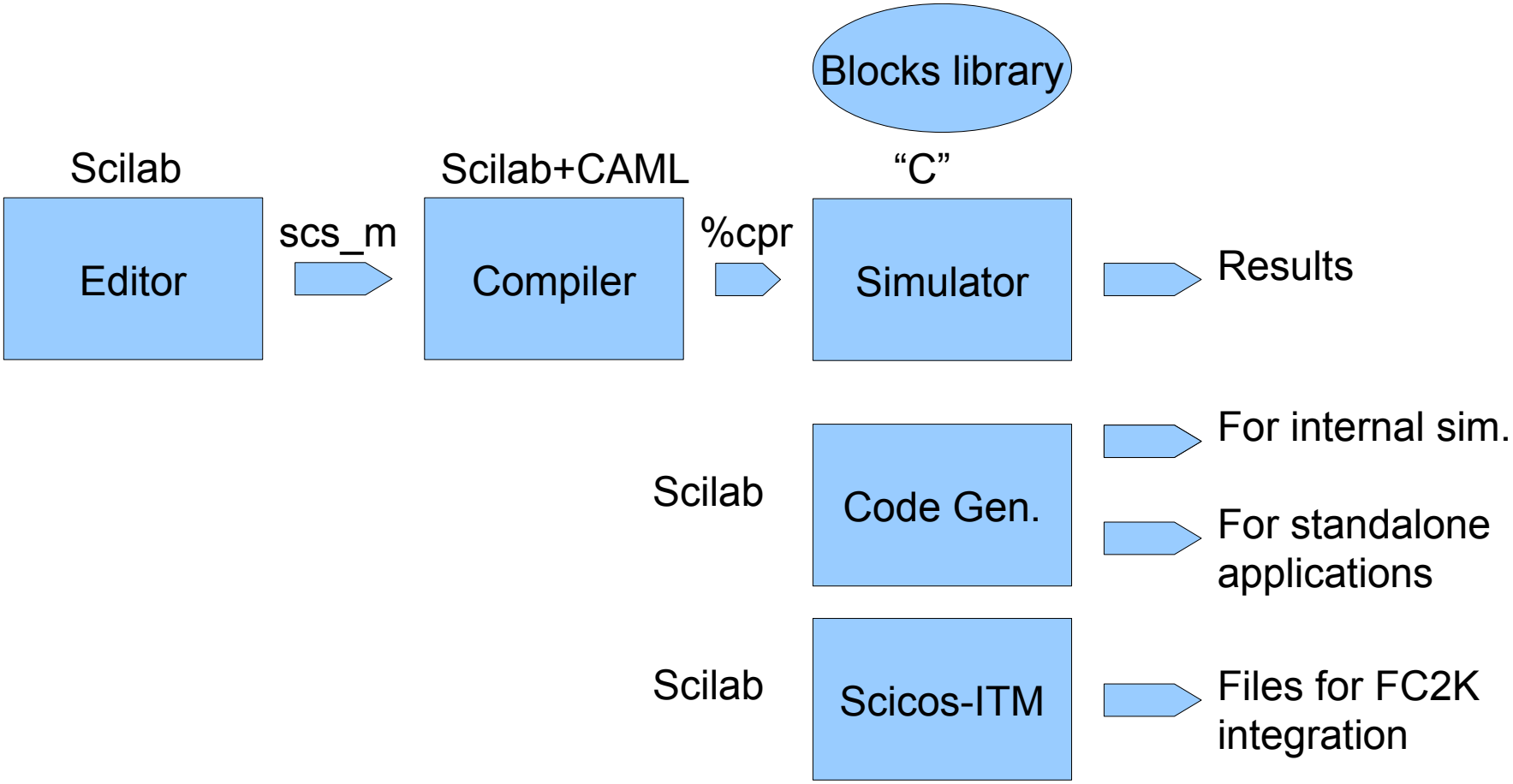
September 2009, status:

- ScicosLab GTK 4.3 : the stable version, full source code available.
- Scicos-RTAI : the last stable version of the code generator of embedded, real time applications, developed by Roberto Bucher (SUPSI, Lugano)

Development actions:

- ScicosLab 4.3 has required a minimum of customization to become able to run on the 64 bit remote Linux server.
- Scicos-ITM: Scicos-RTAI has been modified and extended, in order to become capable to produce all the files required by FC2K.

Scicos (Scicos-ITM) architecture



Scicos-ITM: where the files are ?

All the “custom” development (like Scicos-ITM) are store inside the “contrib” folder. For Scicos-ITM is “contrib/ITM”. This folder contains several sub folders:

"**macros**" : all the Scilab macros (code generator, interfacing functions, etc.)

"**RT_template**": the files that “program” the code generator.

"**routines**" : for the moment this folder is empty. Usually this folder contains the C "static" code, e.g. not dynamically generated. If this folder is not empty, the code must be recompiled before usage (please read the buit-in README file).

Scicos-ITM: “contrib/ITM/macros”

"**macros**" folder (Scilab scripts) :

ITMCodeGen_.sci : the code of the code generator

SetTarget_.sci : allow the re-targeting of the code generator on different architectures/targets. It is use to switch several default options of the code generator;

kepler_generic_inp.sci : input block for Kepler application. This block dynamically generate a C code that recover data from the Kepler environment;
kepler_generic_out.sci : output block for Kepler application. As above, but to Kepler.

loadmacros.sce : the startup file used to load and activate the toolbox; this file is automatically called by "scilab.star" when ScicosLab starts.

Scicos-ITM: “contrib/ITM/RT_template”

"RT_template" folder

* itm.gem : this file specify the elements of the tool chain; in this case:

* kepler.mak : the template makefile. This is the standard "pattern" used to automatically generate the Makefile required to compile the shared library used by FC2K

* kepler.cmd: a list a Scilab command used to "program" the code generator

Scicos-ITM: kepler.cmd

Actually, “kepler.cmd” is just a Scilab script, a sequence of Scilab functions activated in sequence inside the code generator (Scicos-ITM) main loop.

```
lines(0); //** scroll free

[CCode,FCode]     = gen_blocks(); //** generate the C and Fortran code of dyn. blocks

[Code,Code_common, xml, xsd] = make_standalone42(); //** generate the C code
                                                    //** and the parameters files

files           = write_code(Code,CCode,FCode,Code_common); //** write into files

Makename        = rt_gen_make (rdnom, files, archname); //** gen. Makefile

cppwr_file_name = rt_gen_cppwr(rdnom, files, archname); //** the C++ wrapper (FC2K)

cwr_file_name   = rt_gen_cwr(rdnom, files, archname);   //** the C wrapper (“no-init”bug

xml_file_name   = rt_gen_xml(xml, rdnom, files, archname); //** write the file xml

xsd_file_name   = rt_gen_xsd(xsd, rdnom, files, archname); //** write the file xsd

ok = build_fc2k_lib(); //** build the library for FC2K using the Makefile
```

Synchronous and asynchronous events in Scicos

www.scicoslab.org

www.scicos.org

Synchronous and asynchronous (events) signals

Examples :

- Keyboard
- Mouse
- Two clocks on the same MB
- Two clocks on the same Scicos diagram
- The clock and clock/17
- The clock and the clock multiplied by 56/12
- A CPU at 2.5GHz and a DDR2 RAM at 800MHz
- Two watches
- A zero crossing signal
- Two radio controlled watches
- A telephone call

Synchronous and asynchronous (events) signals

In a SYNCHRONOUS system operations are coordinated under the centralized control of a fixed-rate clock signal or a combination of harmonically linked clocks.

An ASYNCHRONOUS system, in contrast, has no global clock: instead, it operates under distributed “just in time” control.

Pure discrete time systems can be simulated and realized as pure synchronous systems.

Mixed (hybrid) continuous time / discrete time systems require a mix of synchronous and asynchronous signals.

Synchronous and asynchronous events

Examples of event handling using Scicos.

Example_1 : pure discrete system

Example_2 : events from continuous systems (sync/async, zero c.)

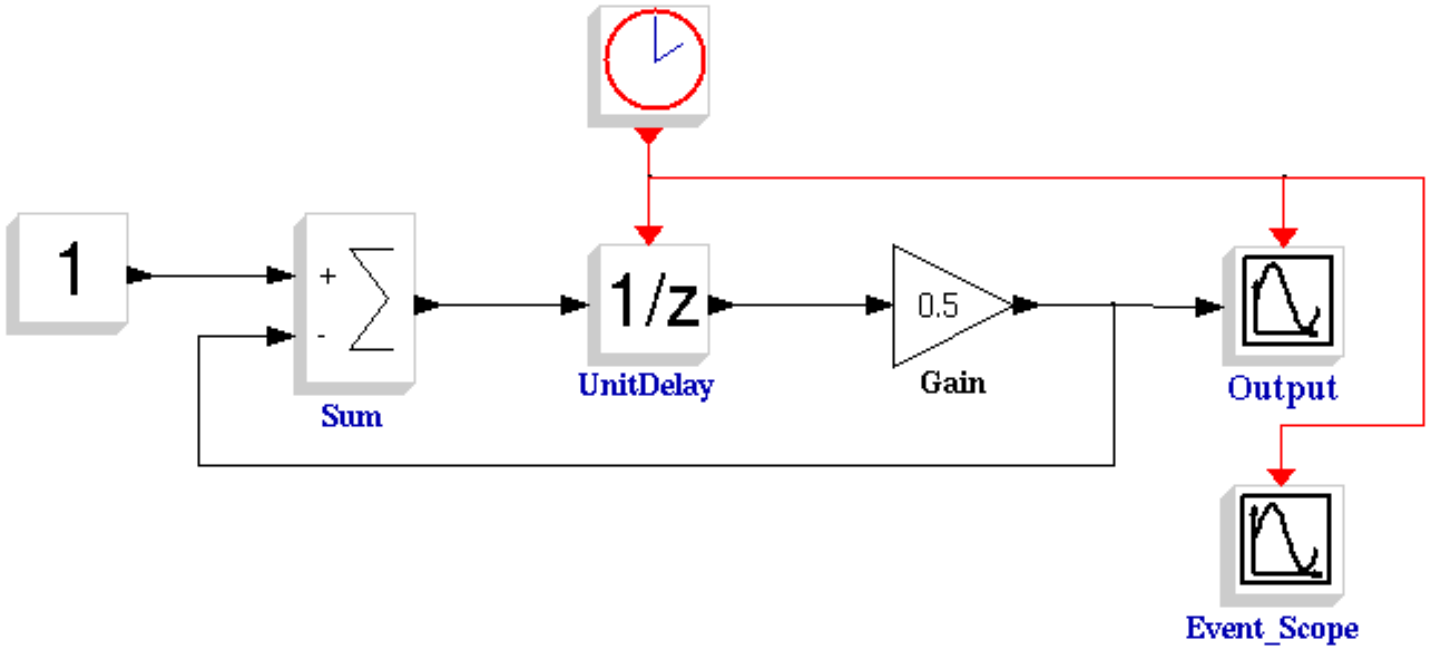
Example_3 : sync/async CLOCKS

Example_4 : typical mistakes using CLOCKS

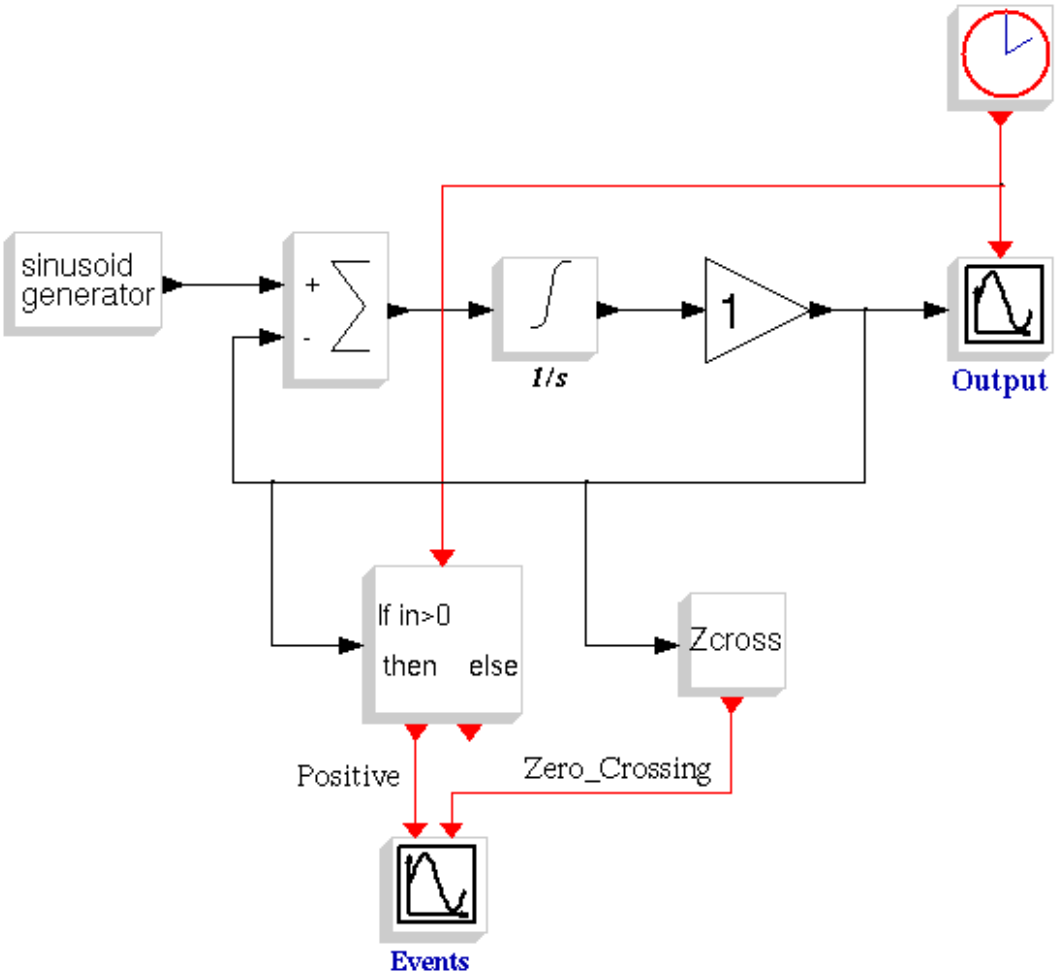
Example_5 : Scicos can talk to you.

The thin Red line (in Scicos)

Simple discrete time simulation (Event/Example_1.cos)



Synchronous and asynchronous events

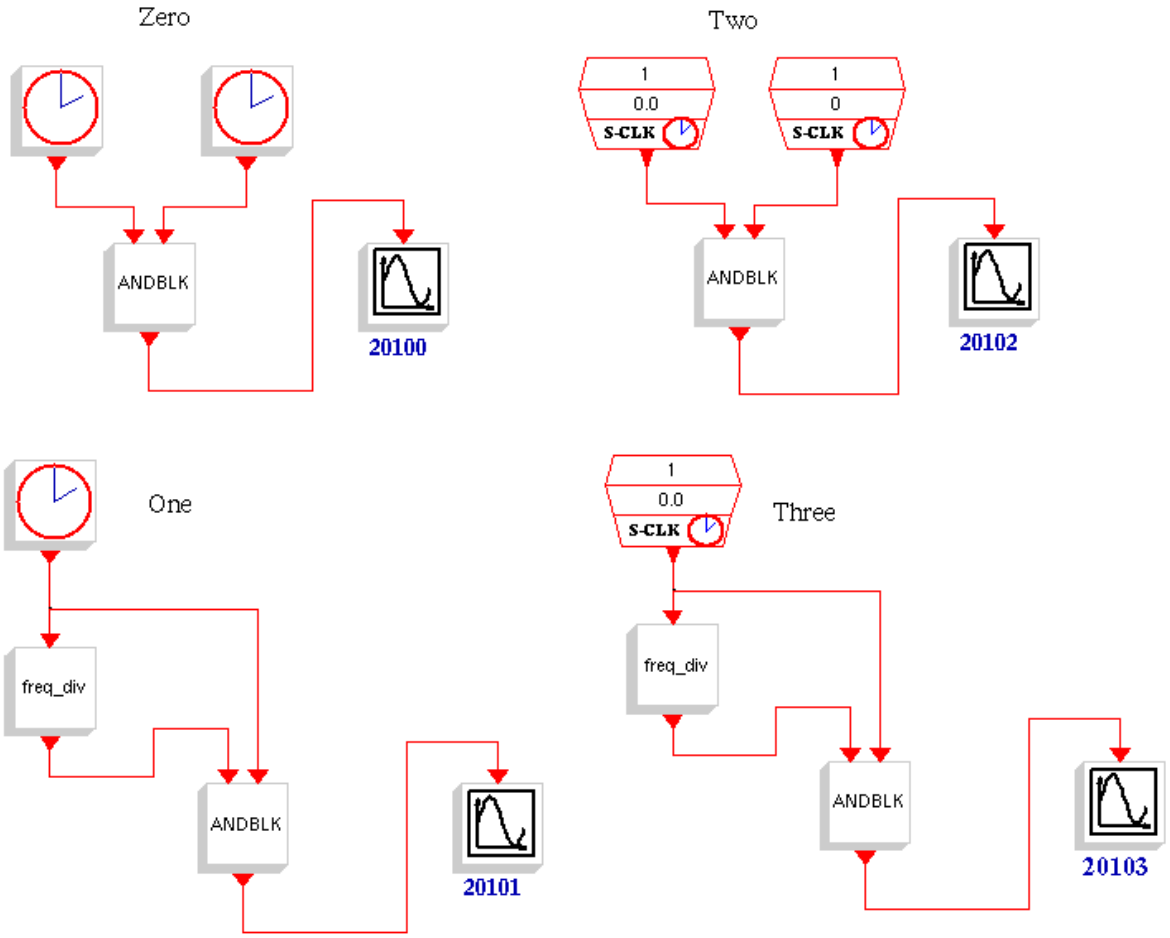


Events from a continuous system.
(Events/Example_2.cos)

Question:

Are “Positive” and “Zero_Crossing” sync or async events ?

More the clocks, more the fun.



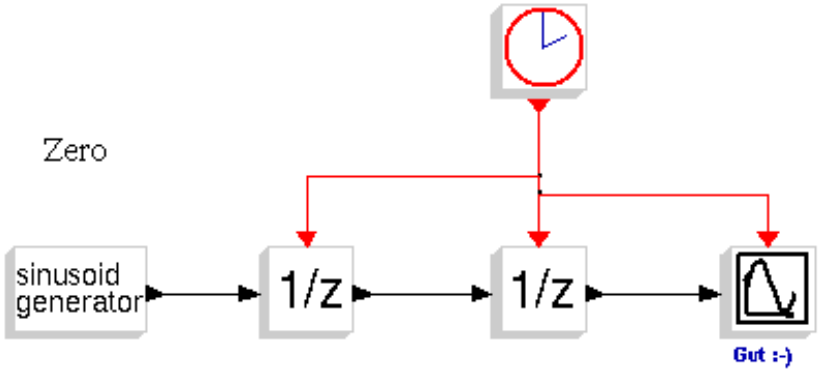
Question:

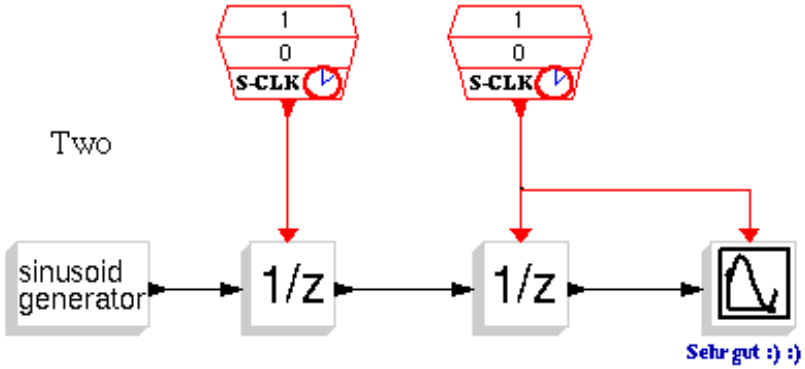
"Are you capable to anticipate simulation results?"

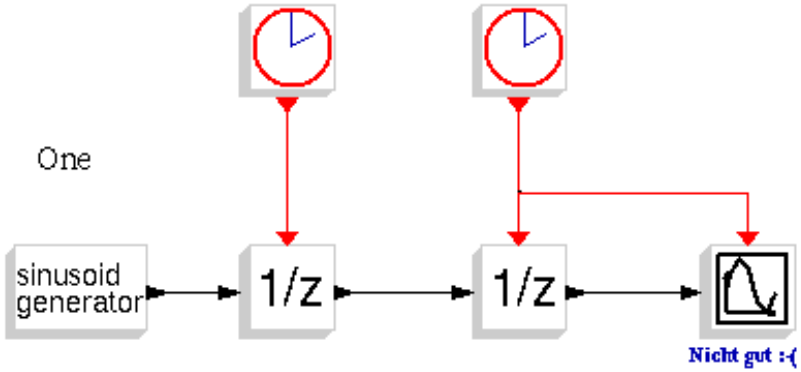
Events/Example_3.cos

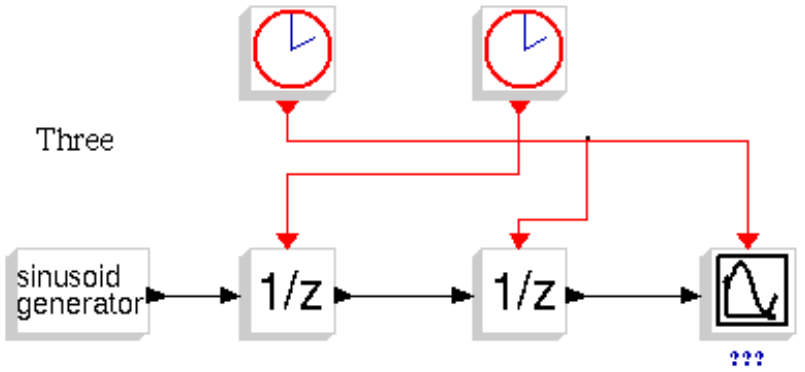
There is more than one of everything (in Scicos)

Question: “Are you capable to anticipate simulation results?”
(Events/Example_4.cos)









Scicos can talk to you

Welcome to the last circle of the “Scicos Inferno”. (Example_5.cos).