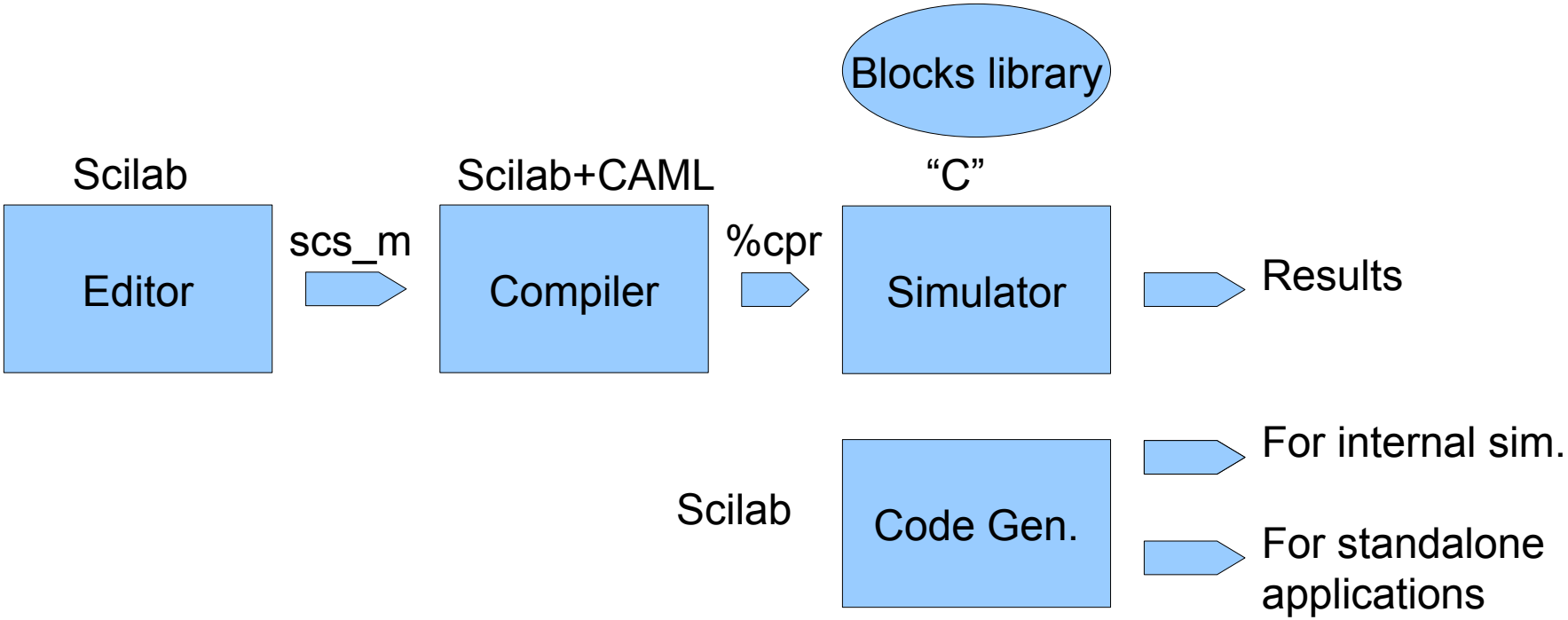
We do ScicosLab - and other things - not because they are easy but because they are hard

www.scicoslab.org

www.scicos.org

Scicos architecture



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 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.

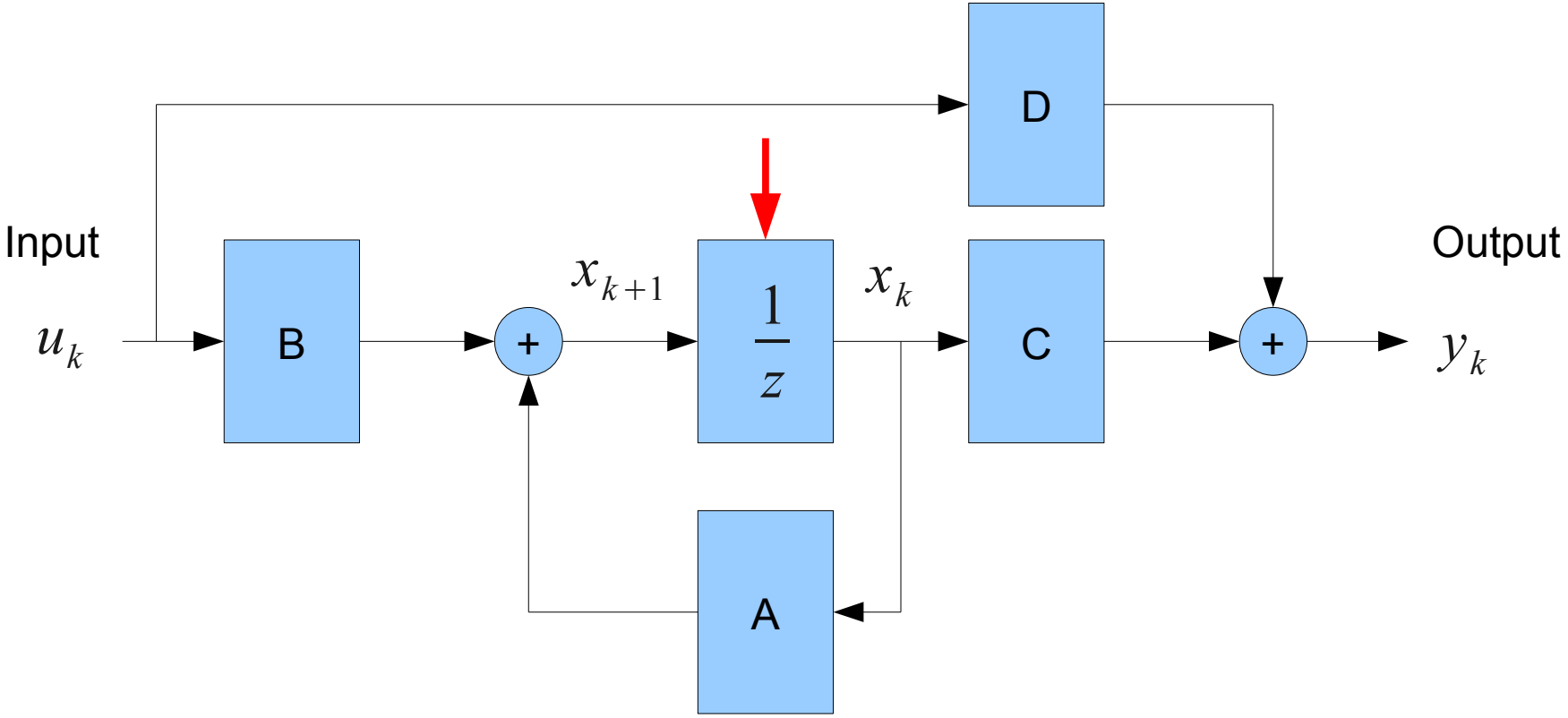
Scicos block computational function

```
#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

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

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



Develop a block in Scicos

Develop a block with two input ports, each port is a vector of three double.
The output is a scalar (double): cumulative dot product of the two input vector.

Folder:
C:\Documents and Settings\Simone Mannori\Desktop\Florence 2010\D2\vector\src\sim_one

Interfacing functions : SIM_ONE.sci

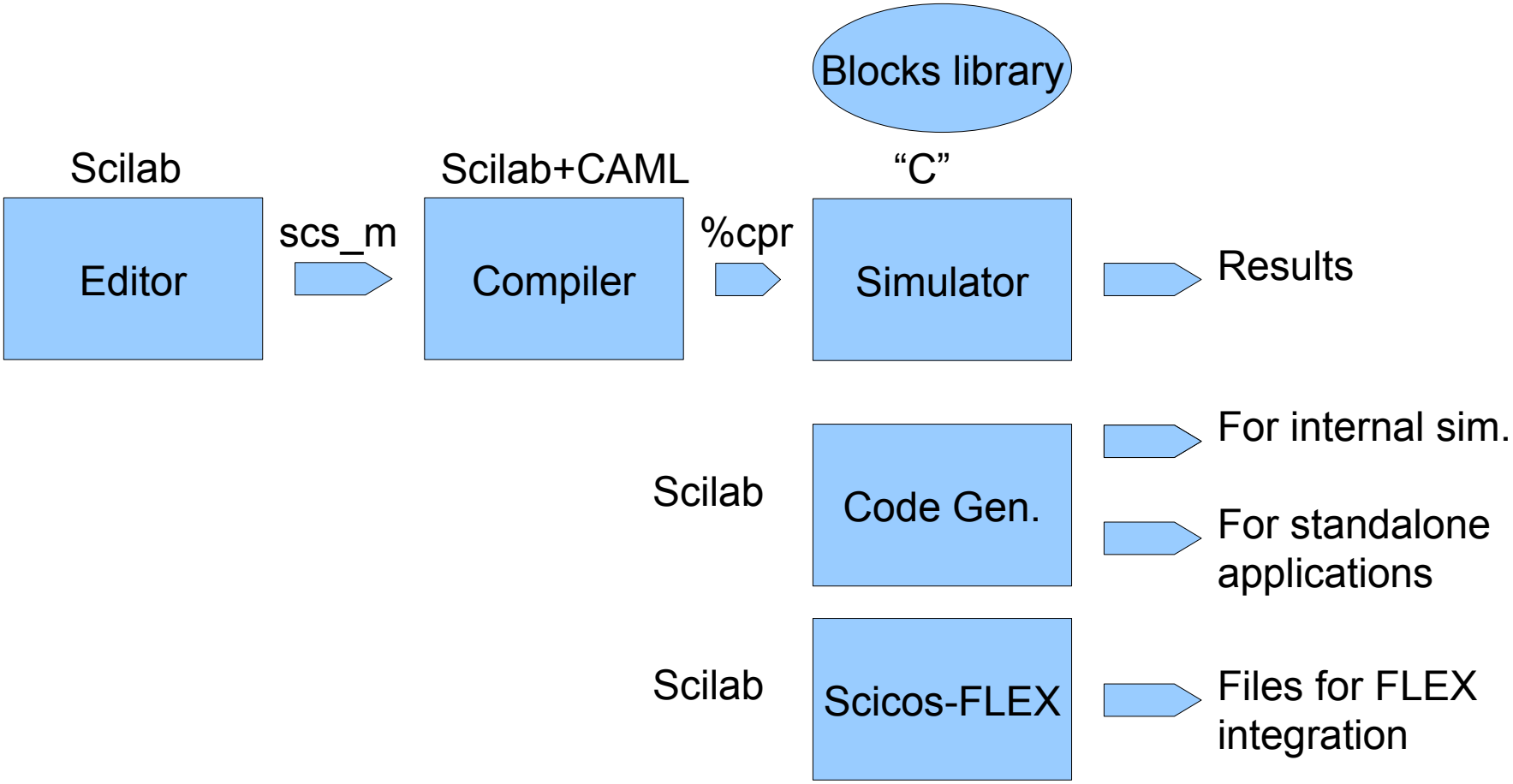
Computational function : sim_one.c

The script builder : build.sce

Code Generation for embedded applications:

Scicos-FLEX

Scicos (Scicos-FLEX) architecture



Scicos-FLEX: where the files are ?

All the “custom” development (like Scicos-FLEX) are store inside the “contrib” folder. For Scicos-FLEX is just “contrib”.

This folder contains several sub folders:

"**dspic**" : most of the code is here (code generator, interfacing functions, etc.)

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

“flex_usb2udp_gateway”: code for FLEX / PC communication protocol

“MSVC2008_Patch” : patch for old Scilab 4.1.2 version (obsolete)

Scicos-FLEX: “contrib/RT_template”

"RT_template" folder

* dspic.gen : this file specify the elements of the tool chain; in this case:

* conf_embcodegen.oil : the template Makefile. For Erika is an “.oil” file
  This is the standard "pattern" used to automatically generate the *.oil file that guide the cross compilation

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

Scicos-FLEX: erika.cmd

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

```
[Ccode,FCode] = gen_blocks(); //** generate the code of the “dynamic” C and FORTRAN
                              //** Scicos blocks

[Code,Code_common]=make_standalonert(); //** generate the code for the main code

Files = write_code(Code,CCode,FCode,Code_common); //** generate the source files

imp_dspicf(rpat,template); //** “dspic/macros/misc” copy several aux files for cross
                           //** compilations

EE_get_diagram_info(rdnom,XX); //** explore the diagram and produce an optimized
                               //** Makefile that compile ONLY the block really used
```

Scicos-ITM: “contrib/ITM/macros”

"dspic/macros" folder (Scilab scripts) :

“codegen” : the files of the code generator

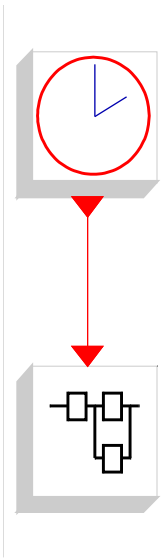
“flex_blocks” : interfacing functions of the FLEX blocks

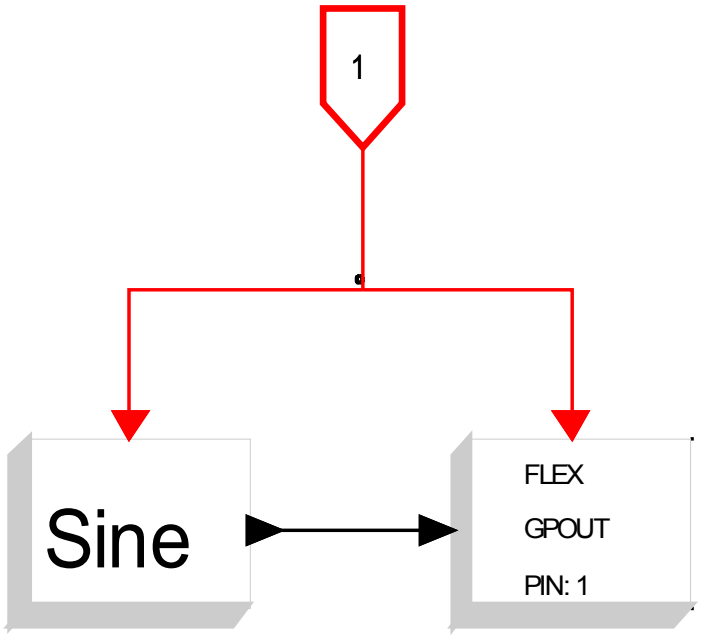
“misc” : various interfacing functions;

“palettes” : palette *.cosf files for Scicos

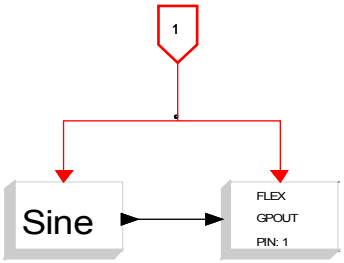
"dspic/NativeInteger" : interfacing and computational functions for native integer blocks for Scicos.

Scicos-FLEX : simple example





“Desktop\Florence 2010\Evidence\FLEX\Test_1.cos



Scicos-FLEX : simple example

	Scicos	*Notes*
Initialization: int t1_init()	Init()	One-to-one correspondence
Run time (periodic) int t1_isr(double t)	OuputUpdate() StateUpdate()	Inside our code generator we have merged the the two calls in a single function (called *<name>_isr()*) that summarize the job.
Simulation ends int t1_end()	Ending()	One-to-one correspondence

"Desktop\Florence 2010\Evidence\FLEX\Test_1.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 ?

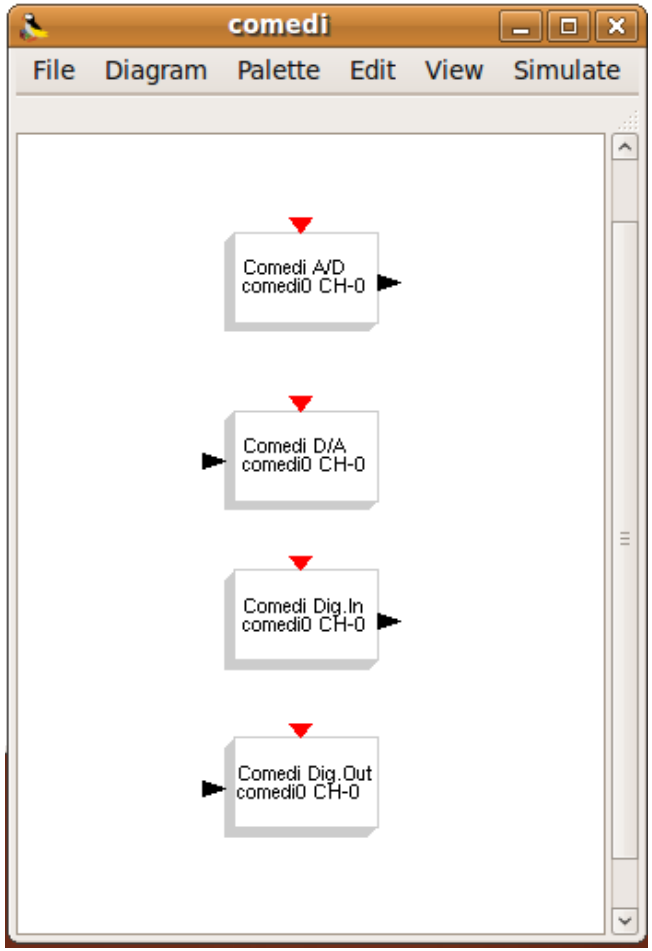
What is necessary to implement HIL ?

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.

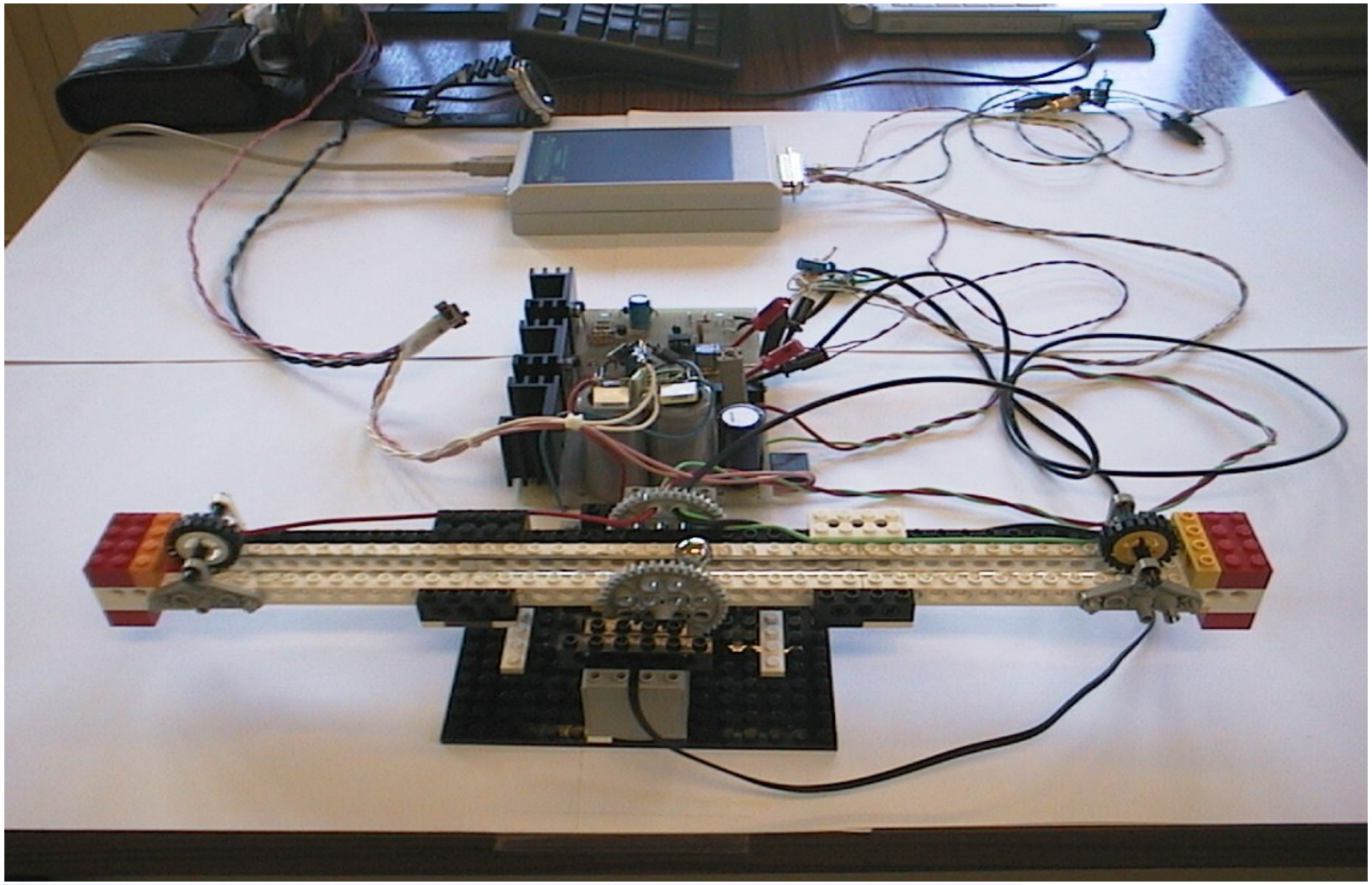
The quality of service is very operating system dependent.

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 ...

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

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

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

$$\dot{x} = v$$

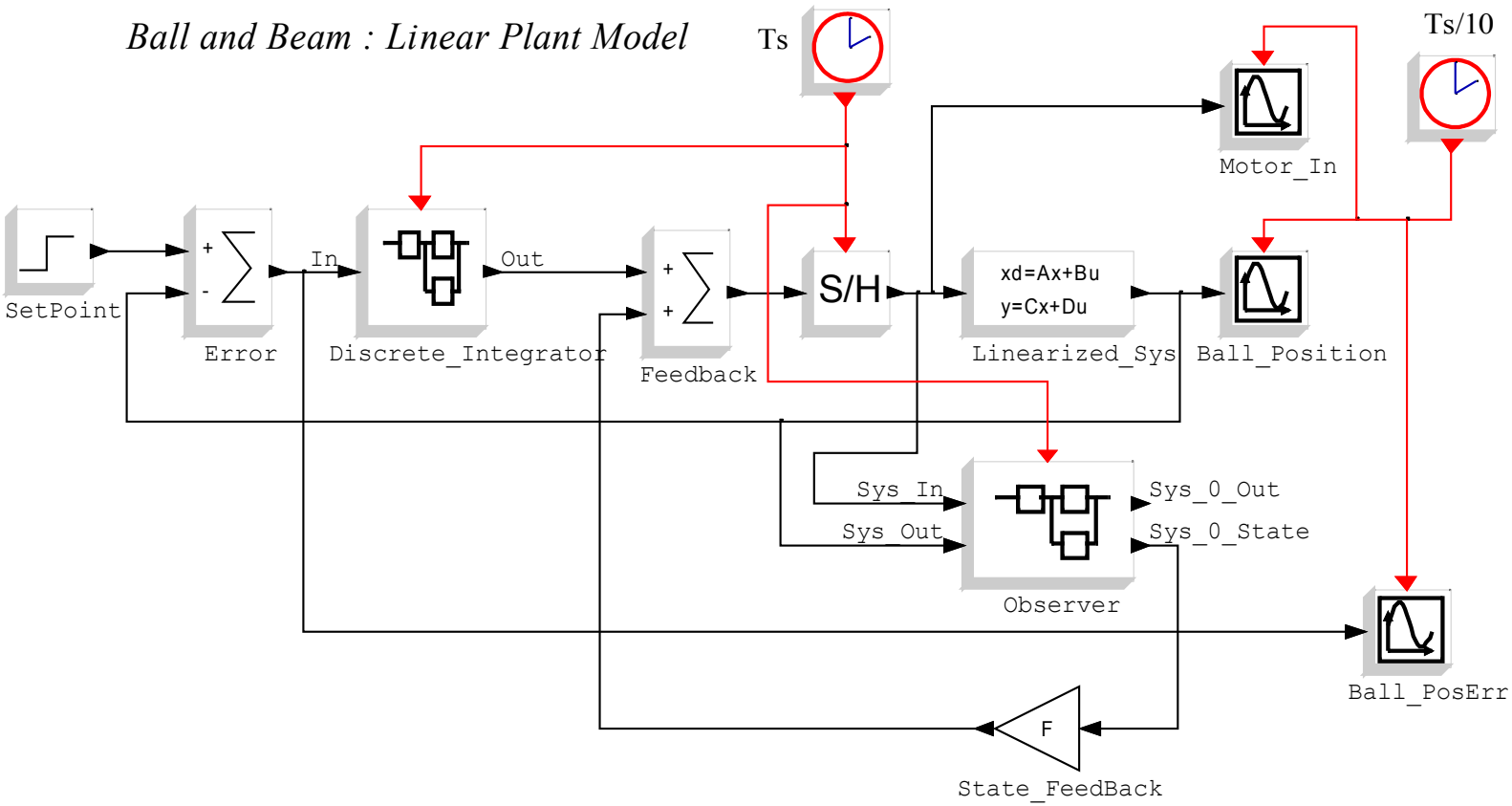
- *Non linear (in many ways)*
- *Unstable*
- *Complex (4th order)*

Ball and beam experiment challenge

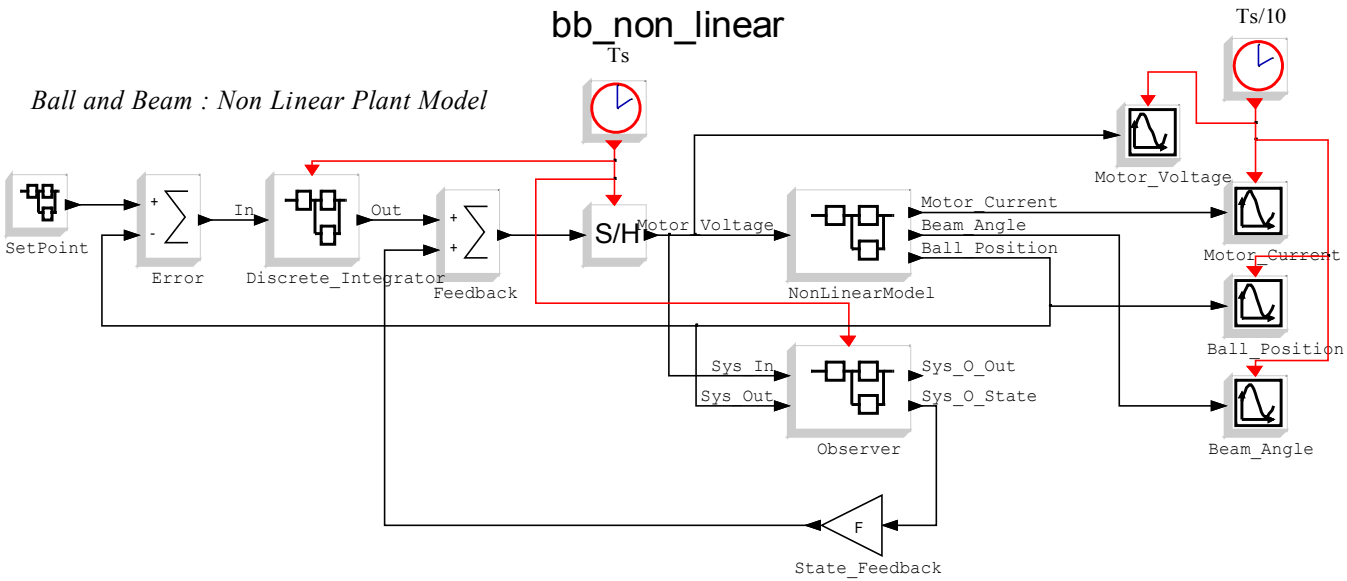
Additional difficulties:

- 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

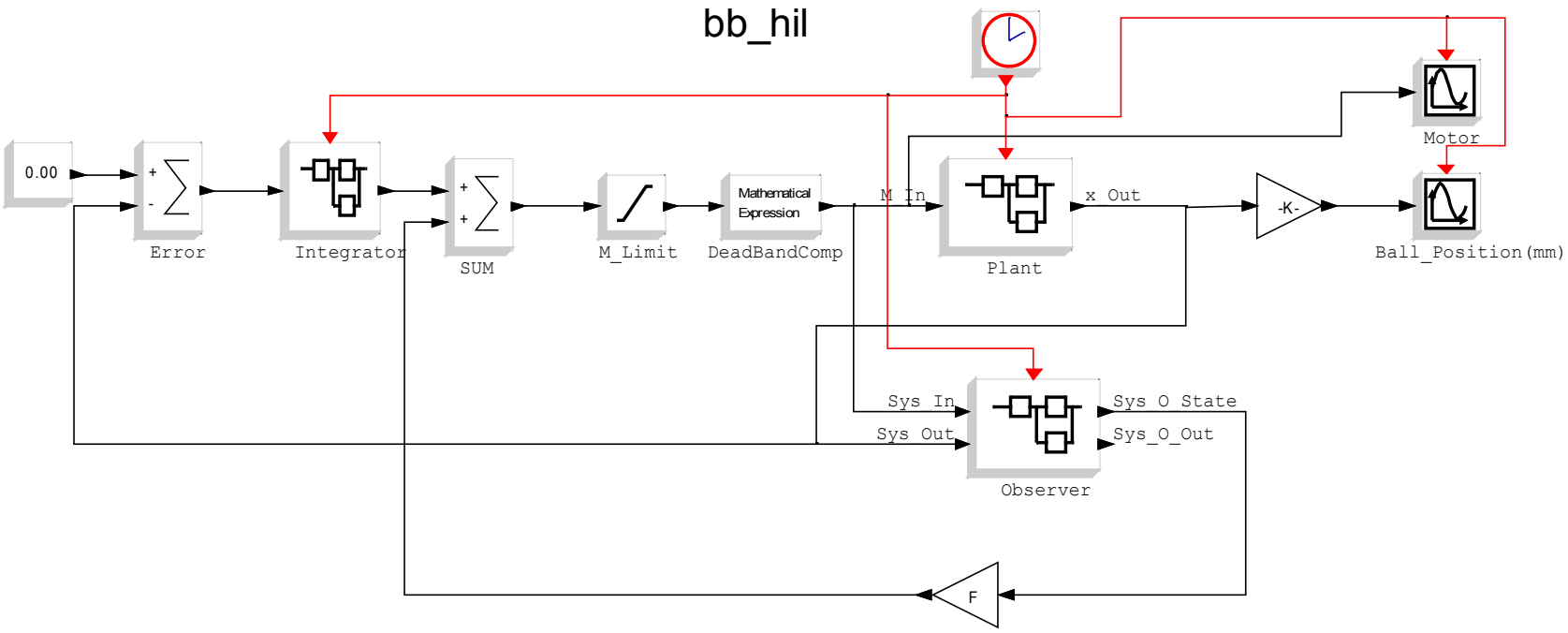


Ball and beam experiment: non linear model

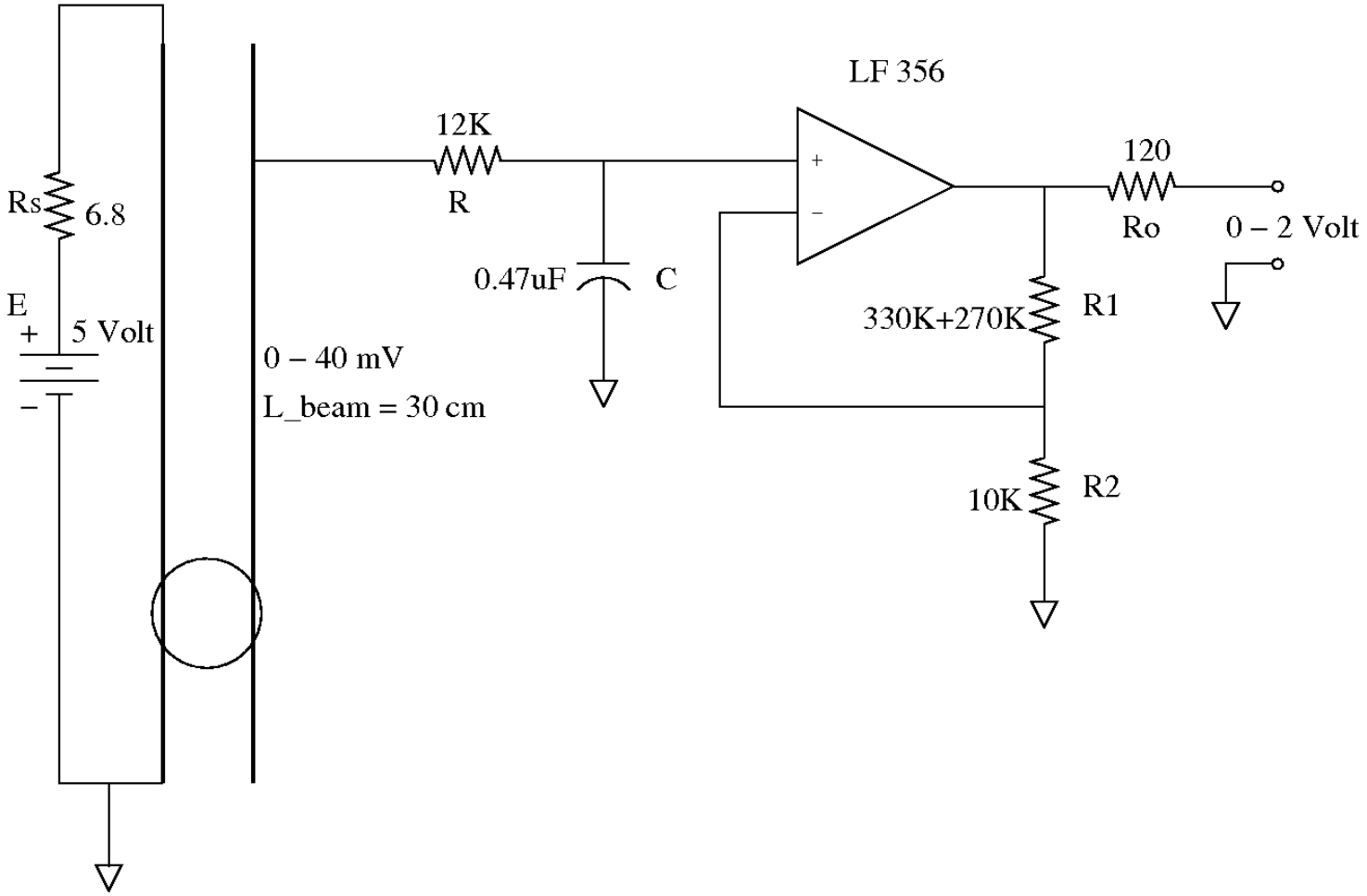



```
/* [ x_dot v_dot theta_dot omega_dot ]  ;  [ x v theta omega]  */

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



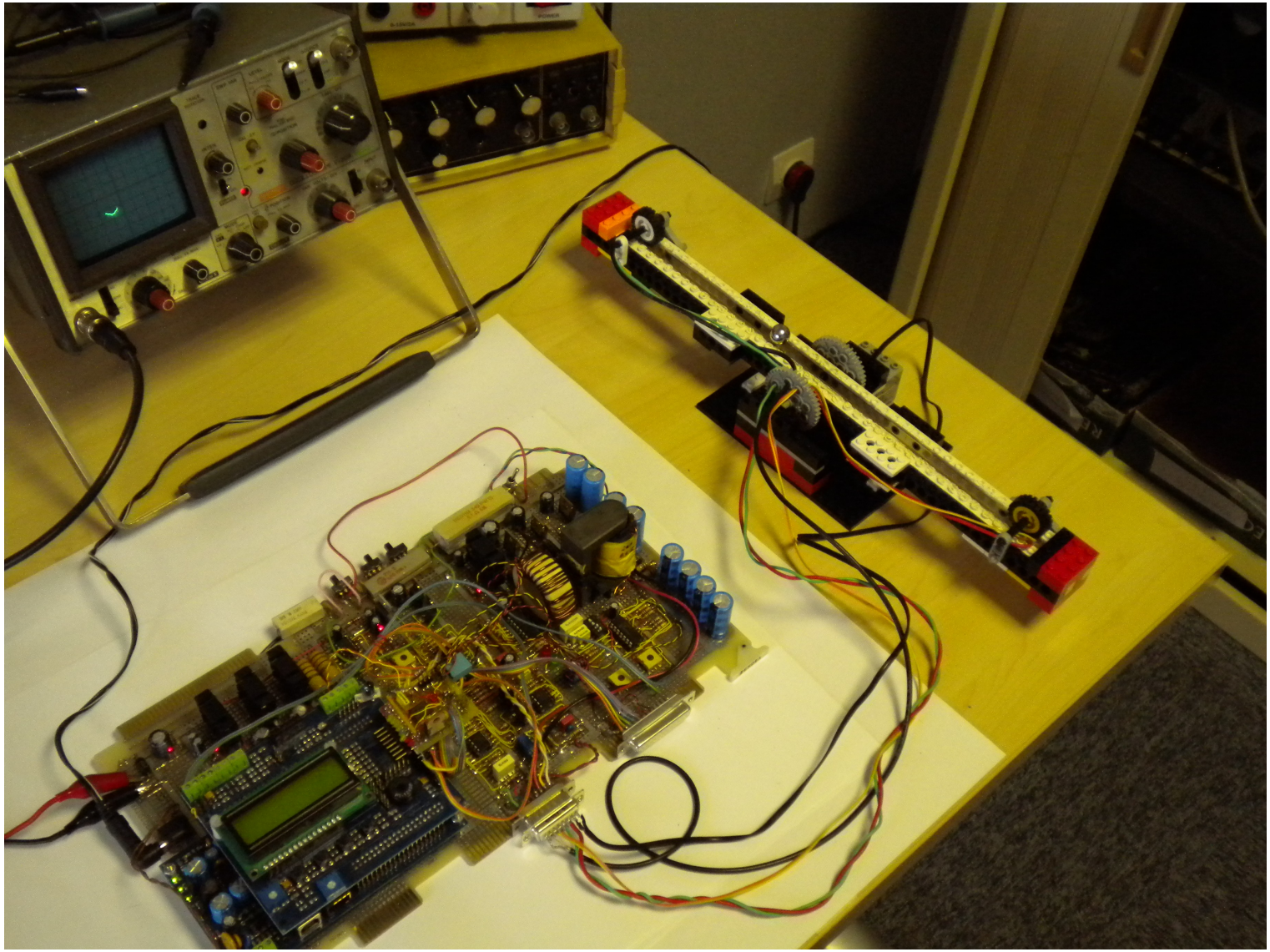
Standalone HIL mode

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)

Advanced Scicos

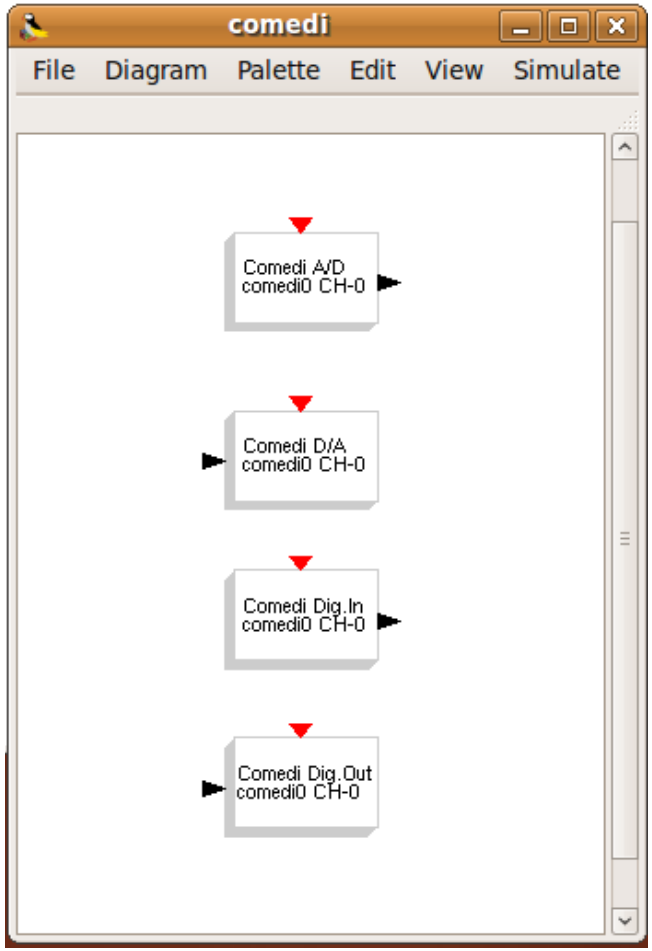


How to mix Modelica and Scicos-HIL blocks

Scicos-HIL : Hardware In the Loop

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

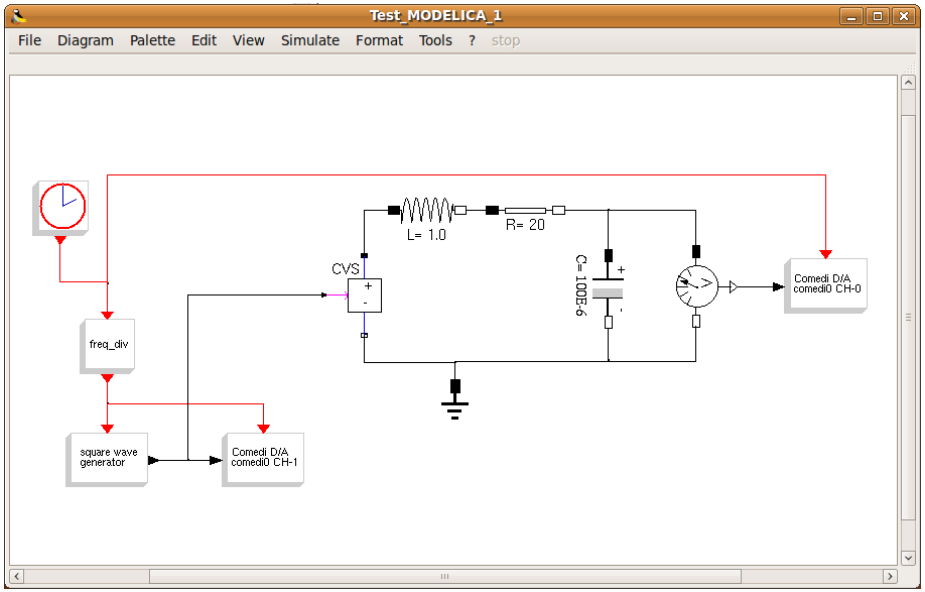
- Analog Input
- Analog Output
- Digital Input
- Digital Output

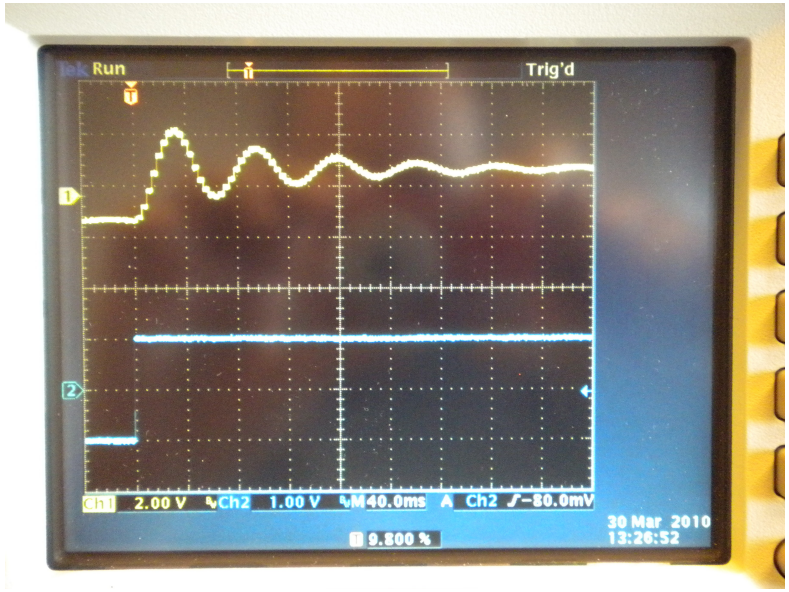


Modelica and Scicos-HIL

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.

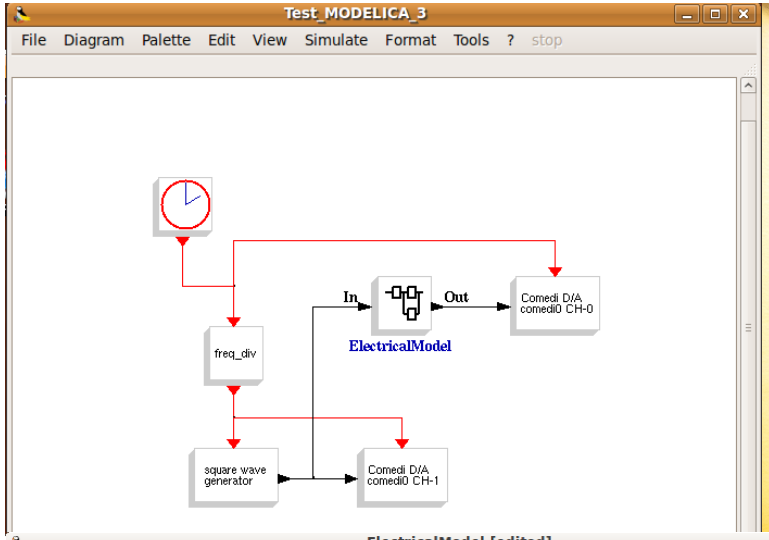
A simple RCL circuit is simulated and the input and output signals are visualized using a real scope connected at the D/A outputs of a data acquisition card.

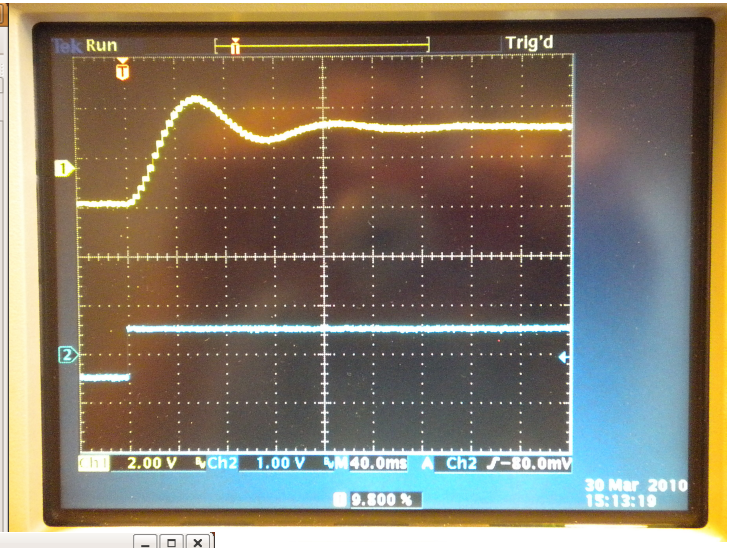


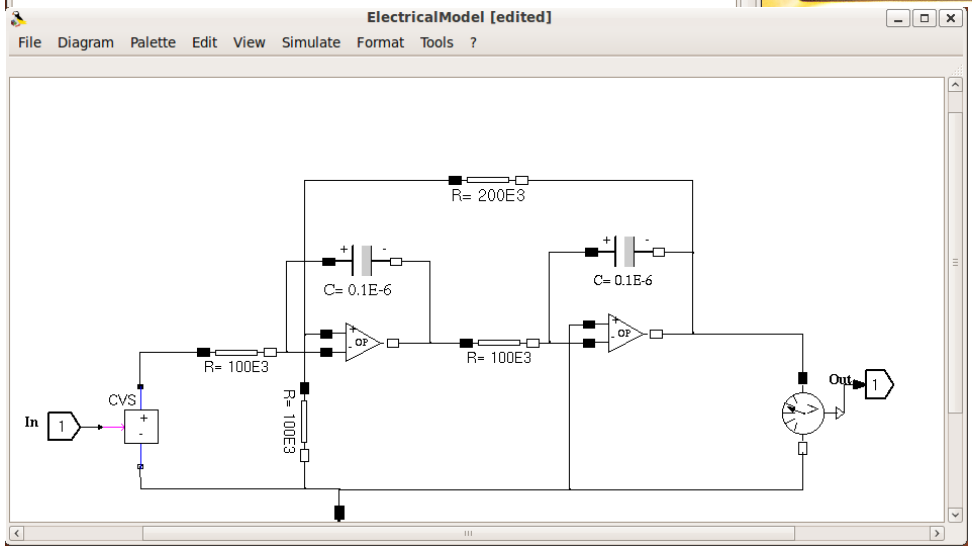


Advanced Scicos

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



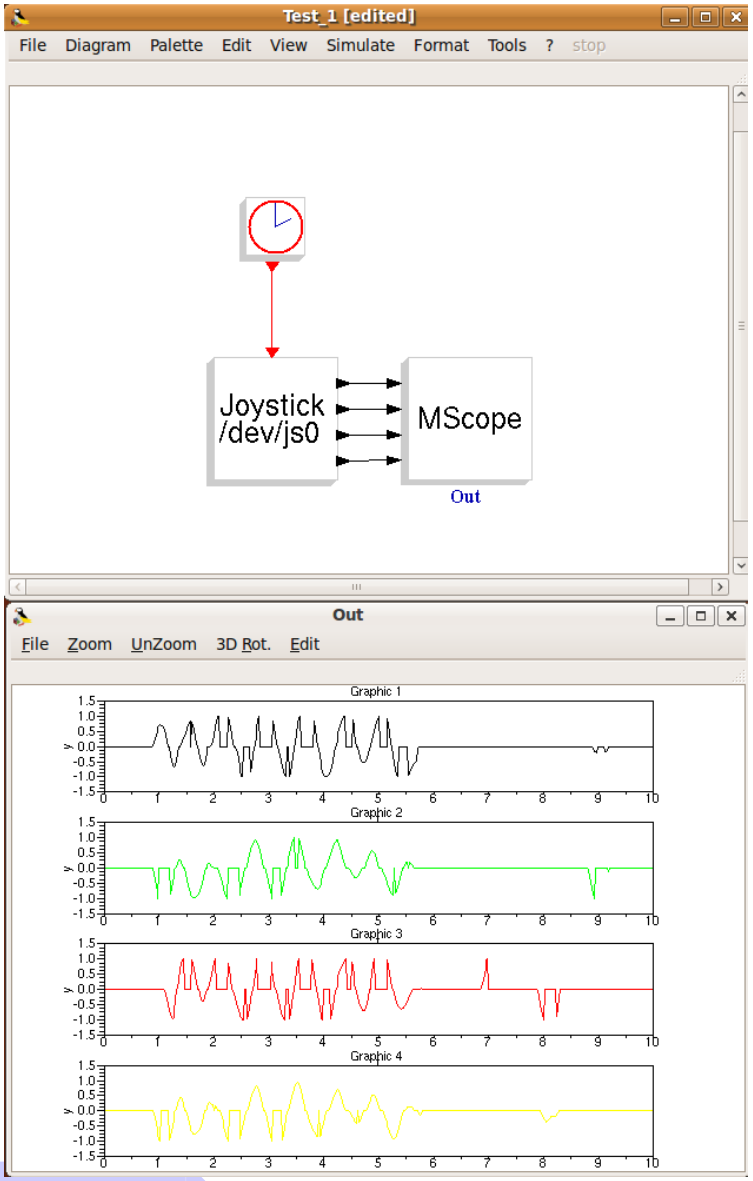




This circuit is simpler to realize than the previous one because it uses standard components of reasonable values.

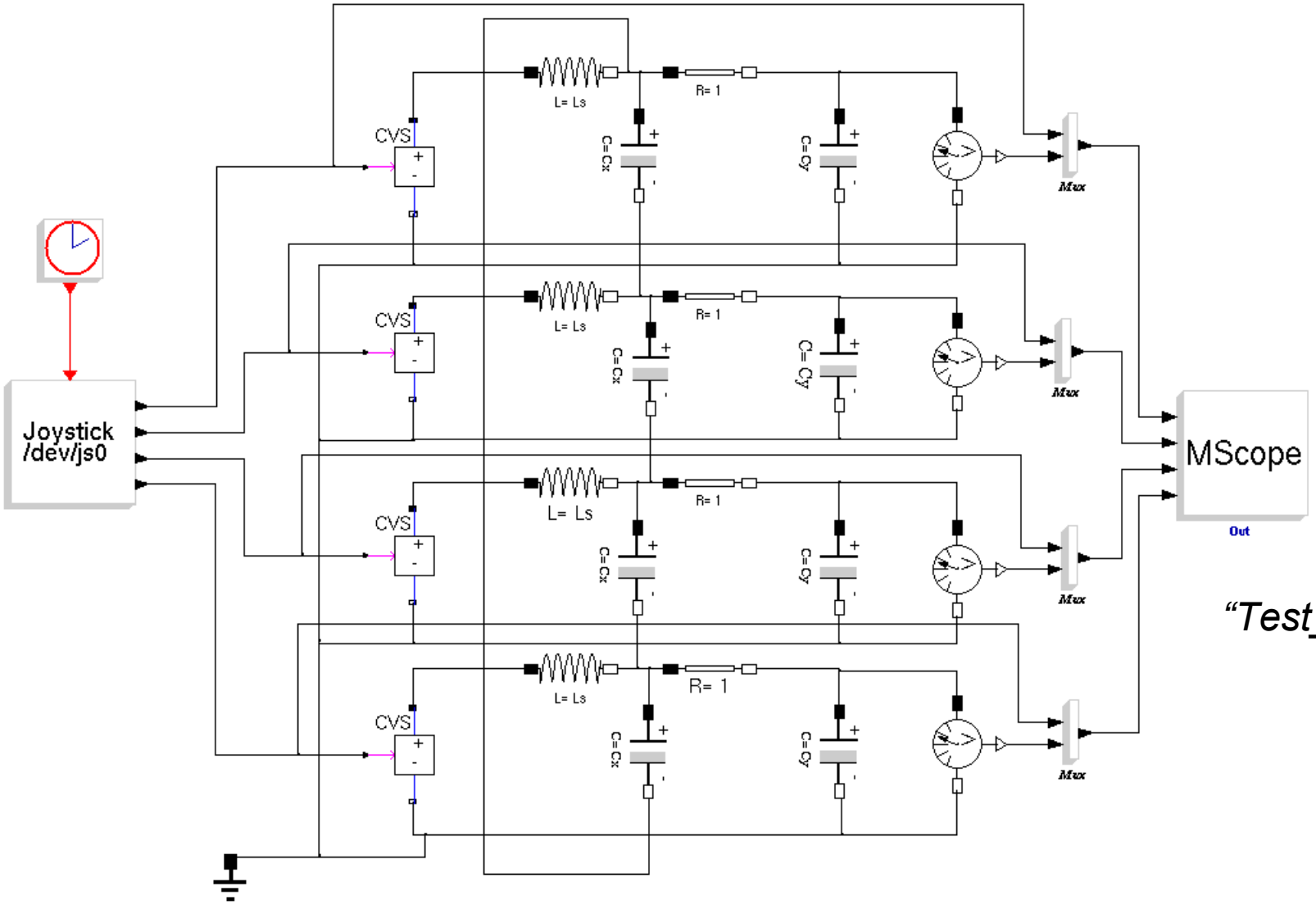
Joystick interface

Scicos-HIL can be extended to support HID (Human Interface Devices) like mouse and joystick.



Joystick interface

Human input for HIL/SIL Modelica applications.



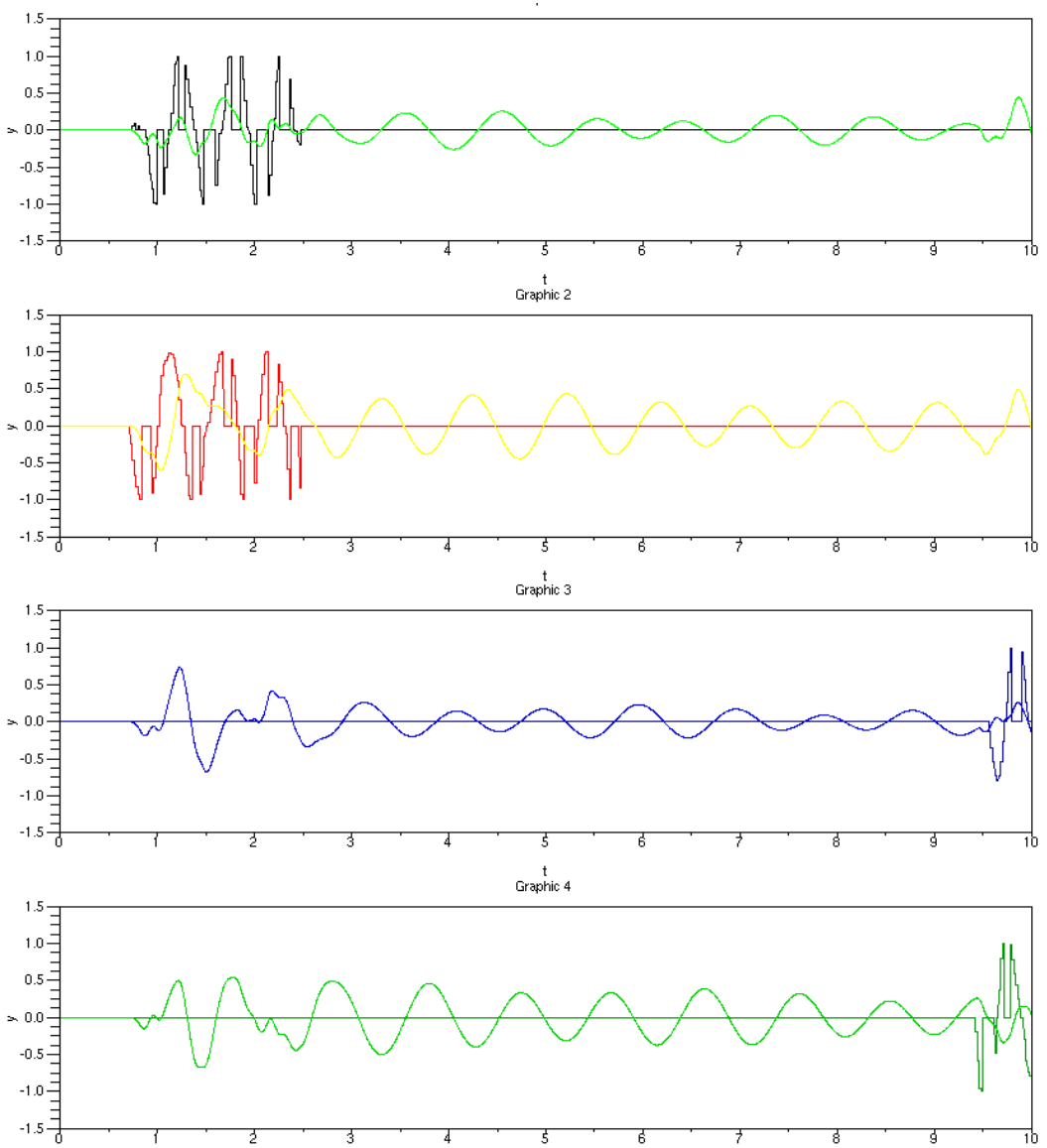
"Test_2.cos"

Joystick interface

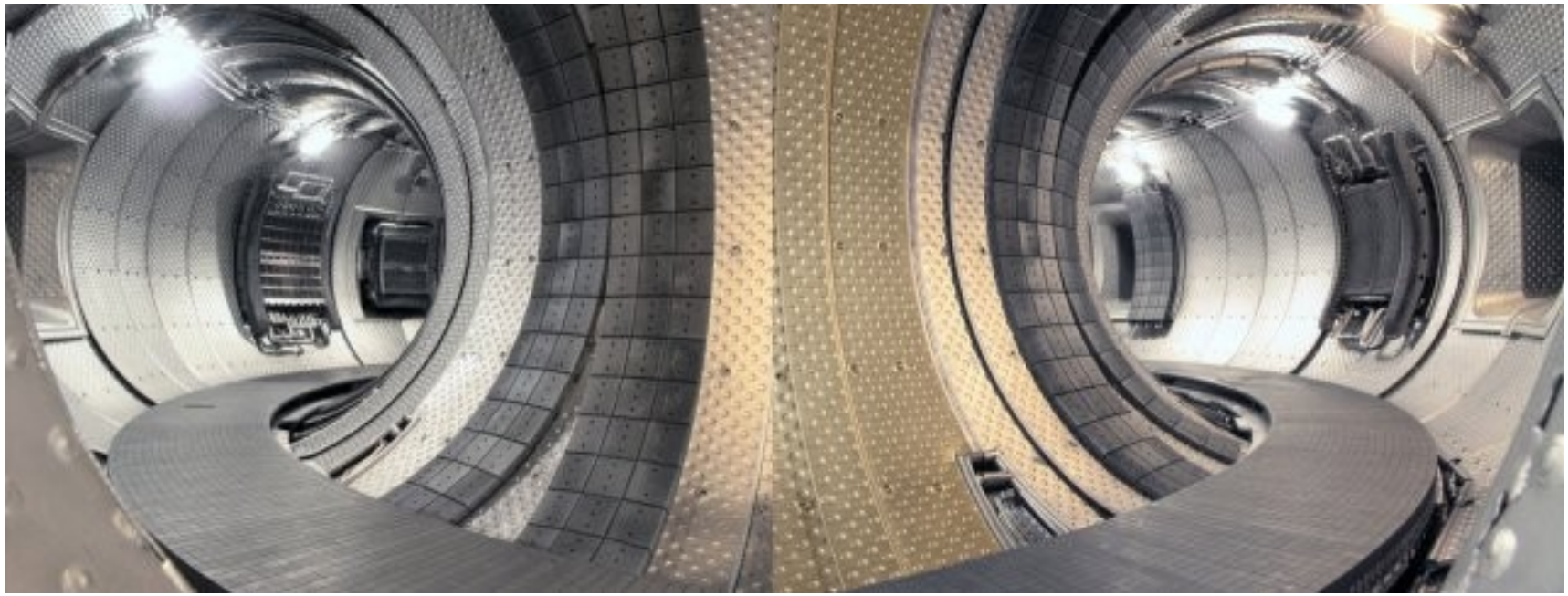
Human input for HIL/SIL Modelica applications.

First two axes produce moving values and the oscillation propagate to the other two.

“Test_2.cos”



Controlled Nuclear Fusion for energy production

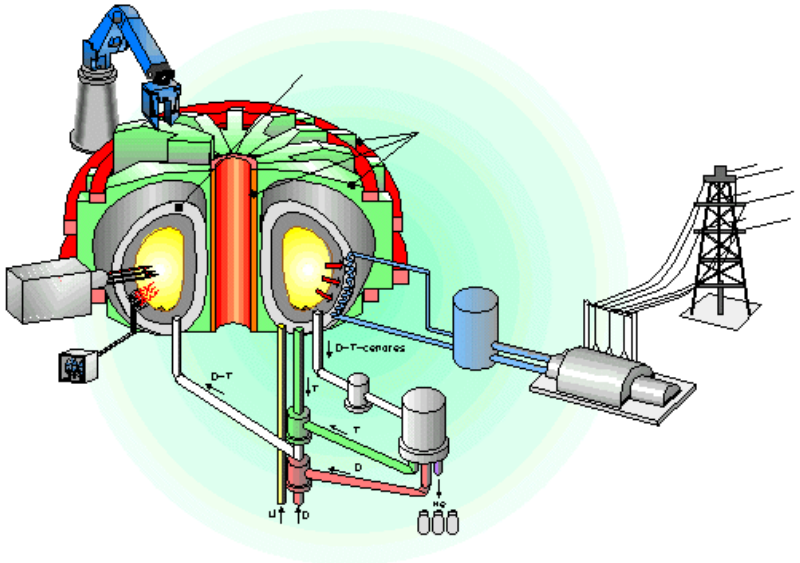


ITER (www.iter.org): Pin = 50 MWatt Pout = 500 MWatt Q = 10

Controlled Nuclear Fusion for energy production

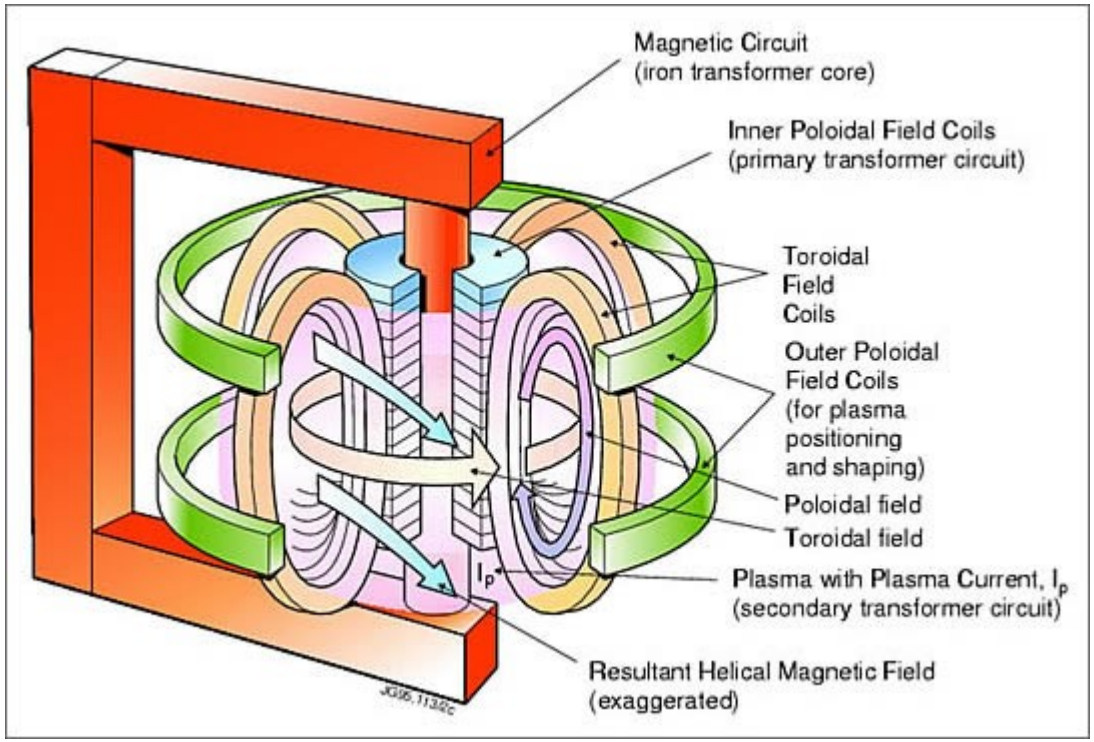
Use some field coil to:

- Heath the plasma
- Force temperature/density profiles
- Containment (position control)
- Cancel internal instabilities
- Recovering of fusion byproducts (divertor)



***Beware**: this is only a part of the problem…*

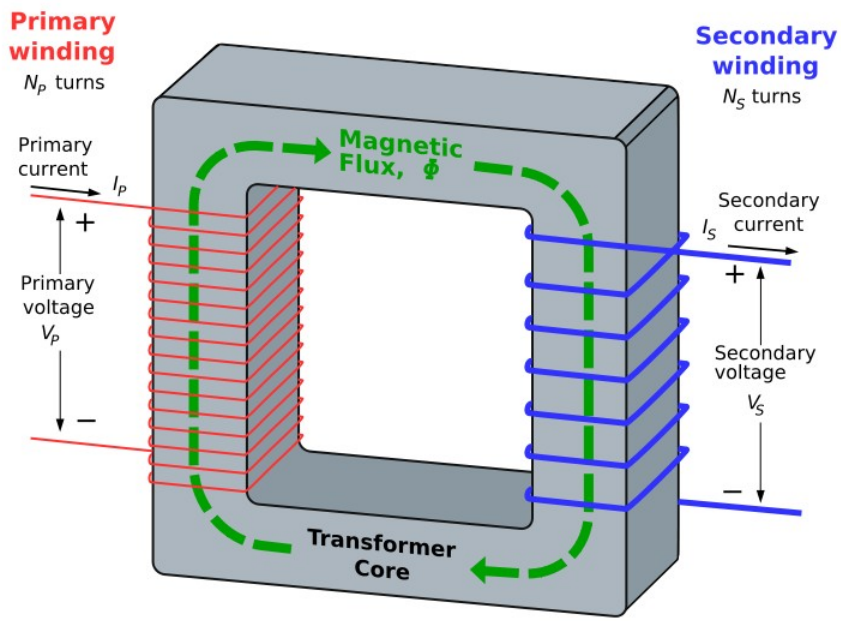
Flux Equations



$$\dot{\Phi}_C(I_T, I_P, W) + R_T I_T = V_T$$

$$\dot{\Phi}_P(I_T, I_P, W) + R_P I_P = 0$$

"Simple but not simpler"



$$\dot{\Phi}_P(I_P, I_S, W) + R_P I_P = V_P$$
$$\dot{\Phi}_S(I_P, I_S, W) + R_S I_S = V_S = 0$$

In the electric transformer "W" is constant: geometry does not change. The iron core is "rigid" and well fixed.

Plasma geometry control ("W") is one of the challenges of this problem.

In the simulations "W" is computed using a very complex model.

In the real applications, "W" is computed using many magnetic field sensors placed inside the "doughnut".

Controlled Nuclear Fusion for energy production

Understanding the application

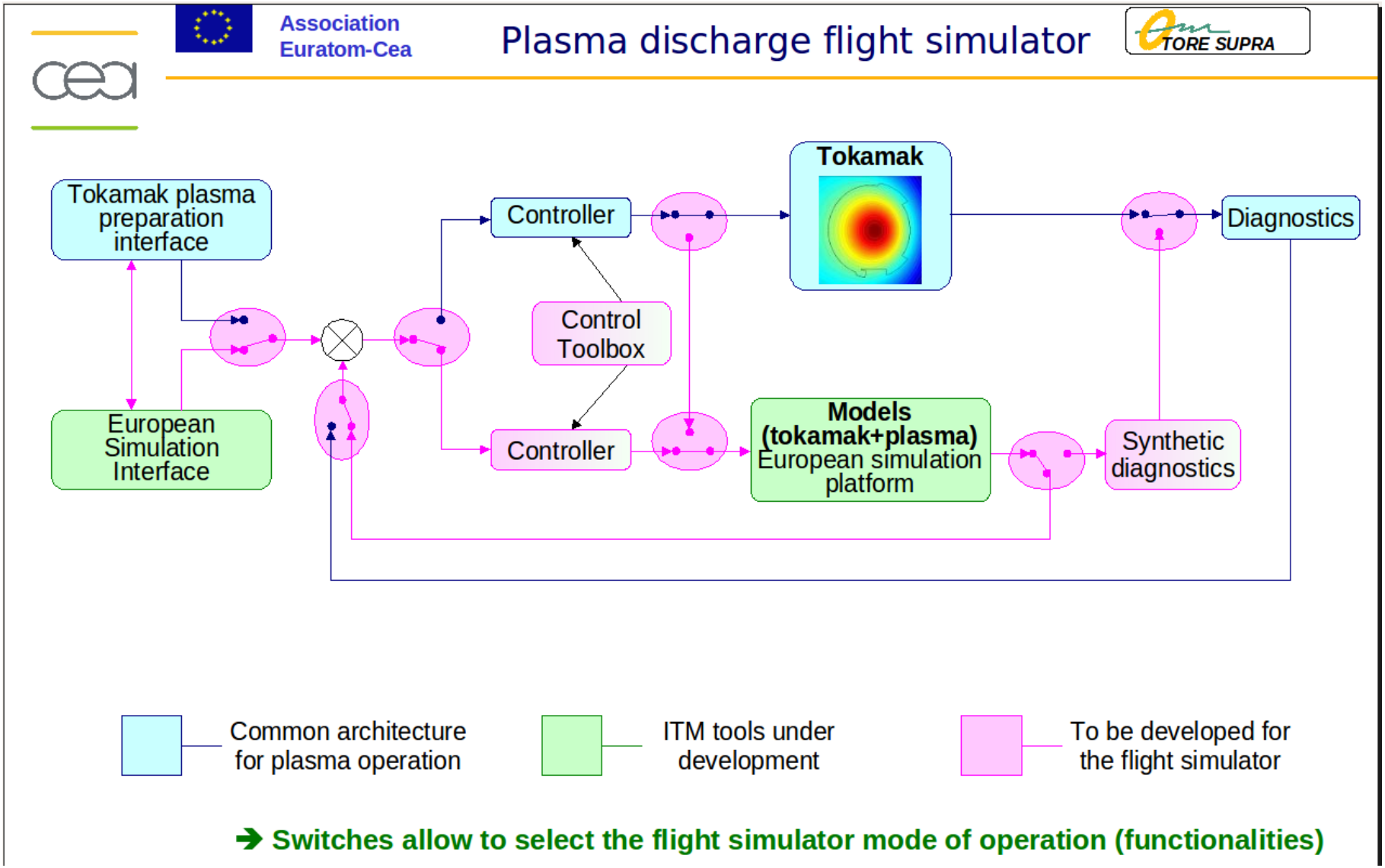
Integrate plasma models (“numeric plasma”) in ScicosLab

Convert Matworks applications to ScicosLab/Scicos

Integrate ScicosLab in Kepler (Java, Eclipse)

Design a Scicos plasma profile controller

Develop a suitable Code Gen for the control system (Scicos-ITM)


Association Euratom-Cea
Plasma discharge flight simulator
TORE SUPRA
Tokamak plasma preparation interface
European Simulation Interface
Controller
Control Toolbox
Controller
Tokamak
Diagnostics
Models (tokamak+plasma) European simulation platform
Synthetic diagnostics
Common architecture for plasma operation
ITM tools under development
To be developed for the flight simulator
➔ Switches allow to select the flight simulator mode of operation (functionalities)

ScicosLab and Kepler together

Kepler

Flexible integration of complex physical simulations developed using other platforms (FORTRAN, C, Matlab, R, etc.)

Strategic choice of ITM

ScicosLab / Scicos

Dedicated platform for design and simulation of complex control systems

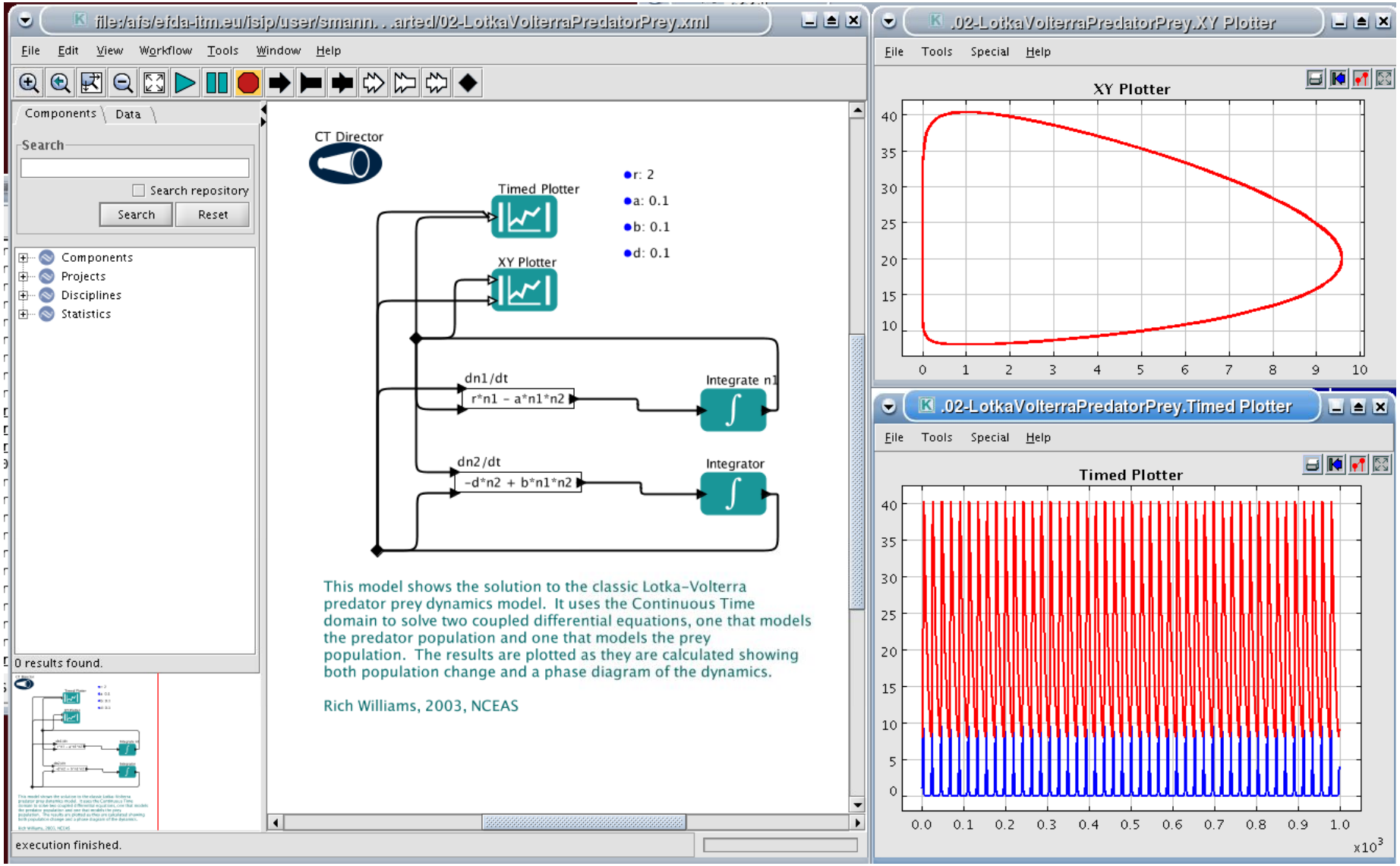
Built in code generation capabilities for simulation and application on embedded systems

Simulink, ScicosLab, Kepler

	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
	The Bad	*The Ugly*	*The good*

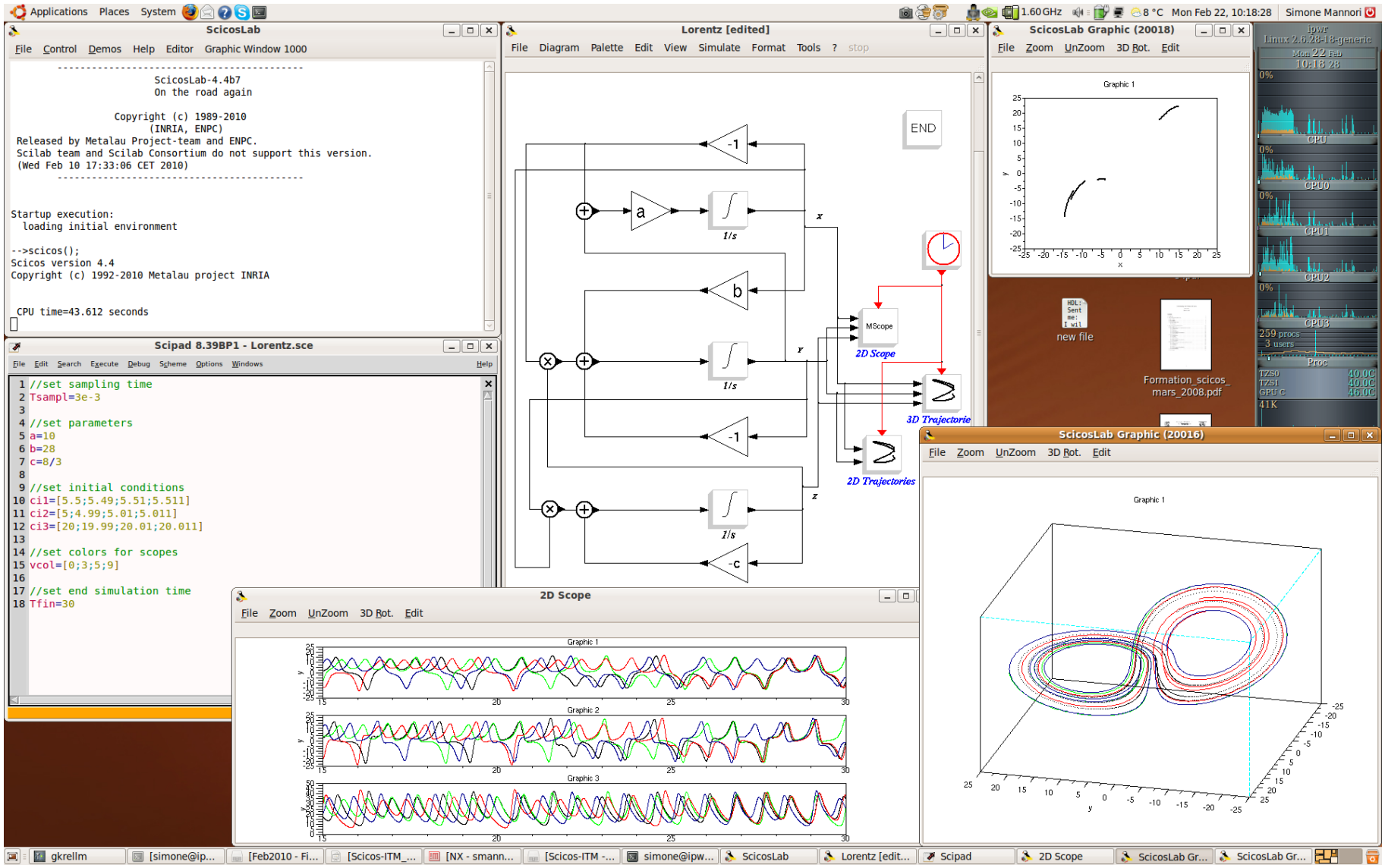
Kepler

Advanced Scicos



ScicosLab

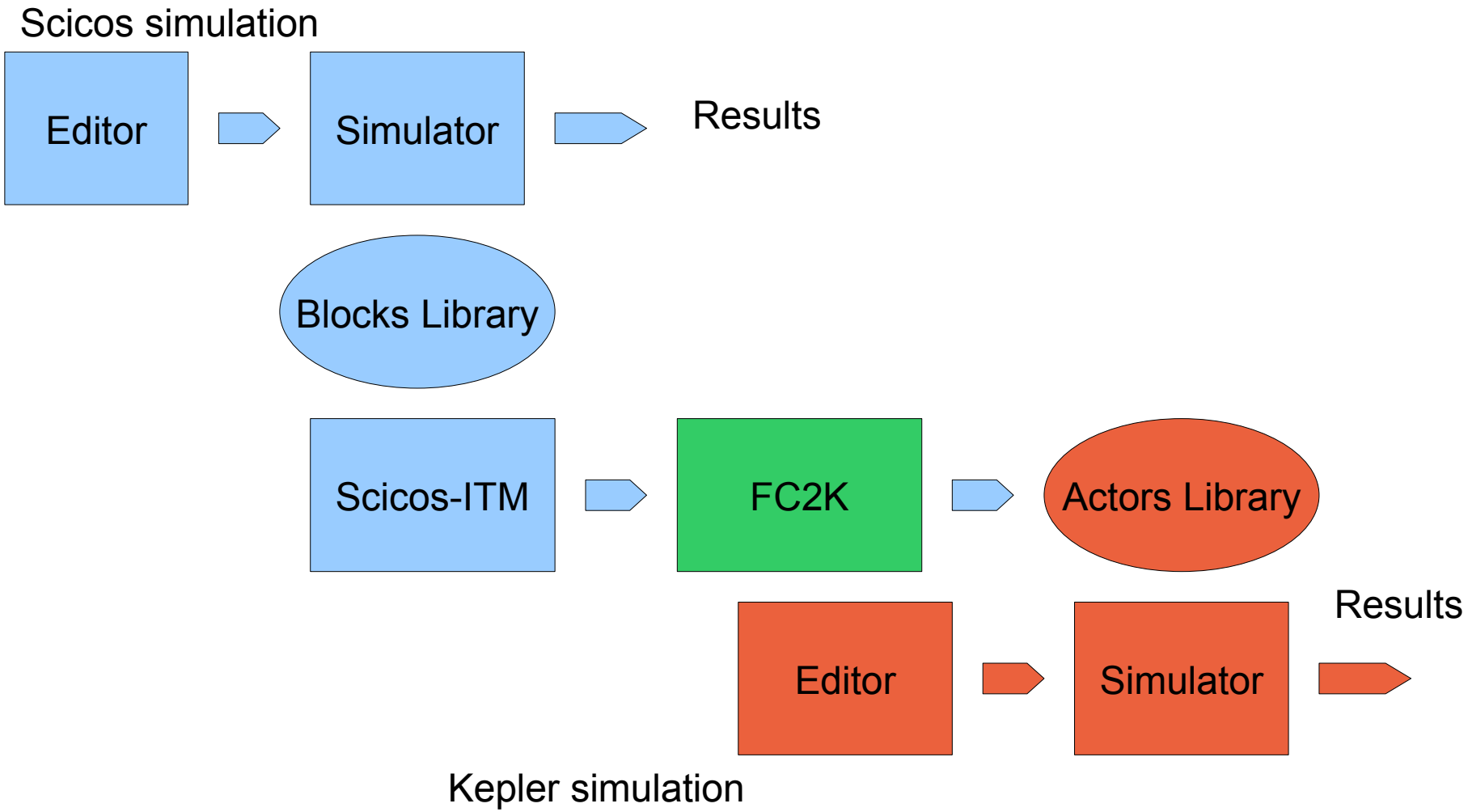
Advanced Scicos



Scicos-ITM How-To

1. Design the control system in ScicosLab
2. Prepare the controller for Code Generation
3. Generate the code using Scicos-ITM
4. Generate a Kepler actor using FC2K
5. Insert the actor inside the Kepler work flow
6. Run the Kepler simulation
7. Don't worry. Be happy :-)

From ScicosLab to Kepler



The system: the floating apple

- I have an apple at y(0)=y0=1.0m.
- At t=0 I drop the apple, and the apple falls down.
- I'm not satisfied: I'd like to see the apple floating at a reference height (ref=0.5m).
- I need a controller to implement a closed loop feedback system.

Open Loop "PLANT" model

The "PLANT" : a free falling apple

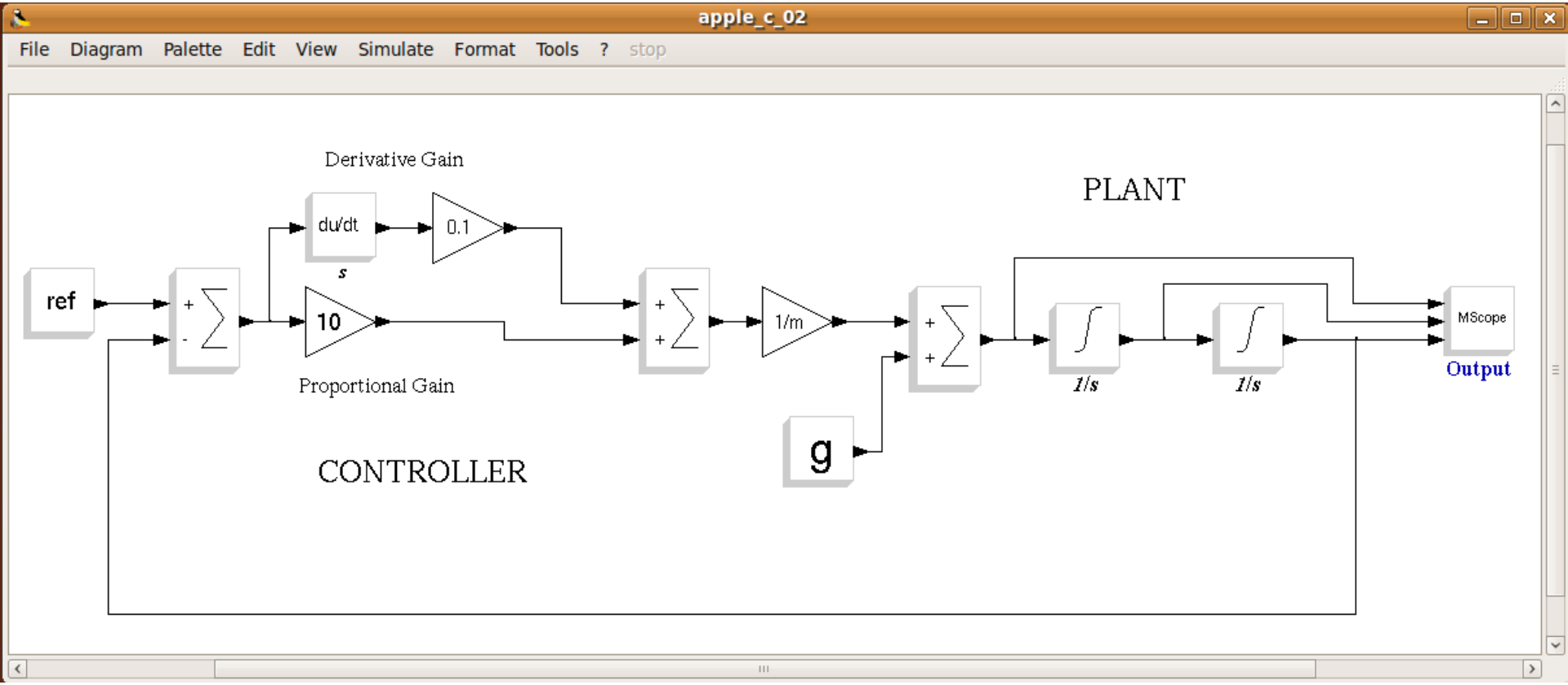
$$F = G\frac{m_a m_T}{r^2}$$

$$F = m\,a \qquad a = \frac{F}{m} \qquad a_a = G\frac{m_T}{r^2};\ g = -9.81\,m/s^2$$

$$v(t) = v_0 + \int a(t)\,dt \qquad y(t) = y_0 + \int v(t)\,dt$$

$$y(t) = y_0 + \frac{1}{2}g\,t^2$$

Time continuous controller



Equivalent discrete system (plant + controller)

Using Euler, we create a discrete equivalent system

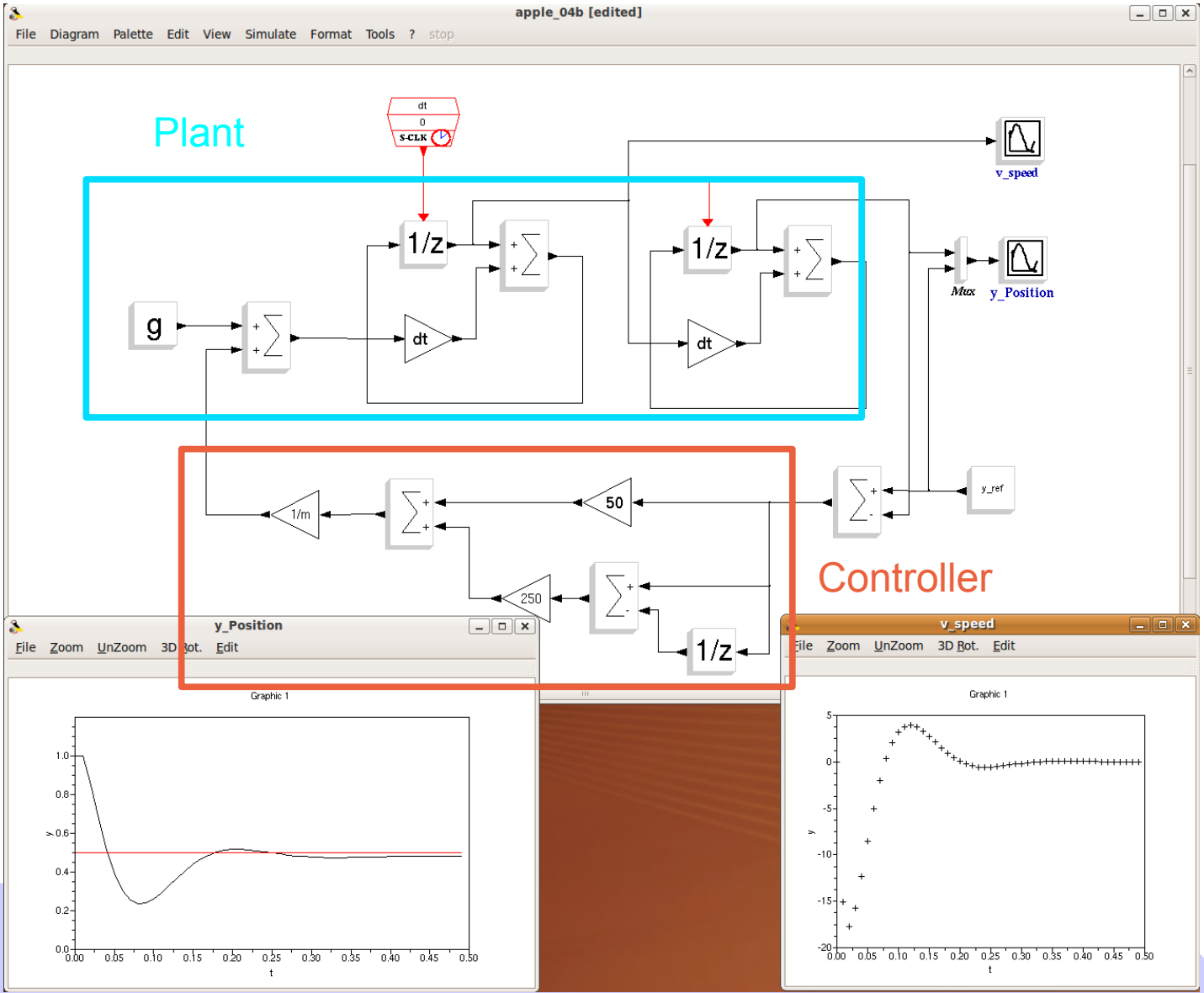
$$v(t) = v_0 + \int a(t)\,dt$$; integral (differential) equation

$$v_{k+1} = v_k + a_k\, delta$$; difference equation

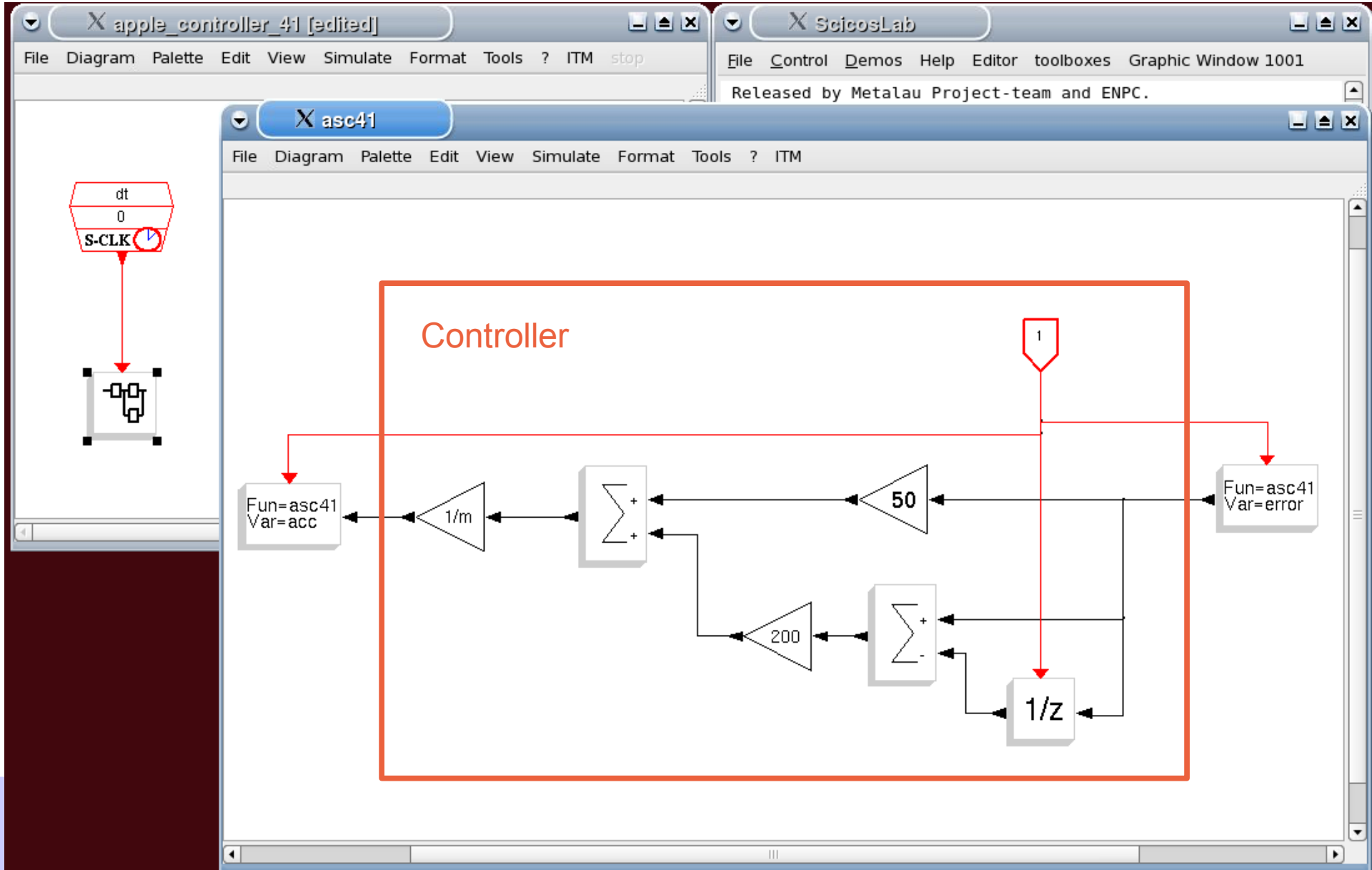
$$v_k = v(kT_s)$$; time sampling

$$delta = T_s$$; discrete time = sampling time

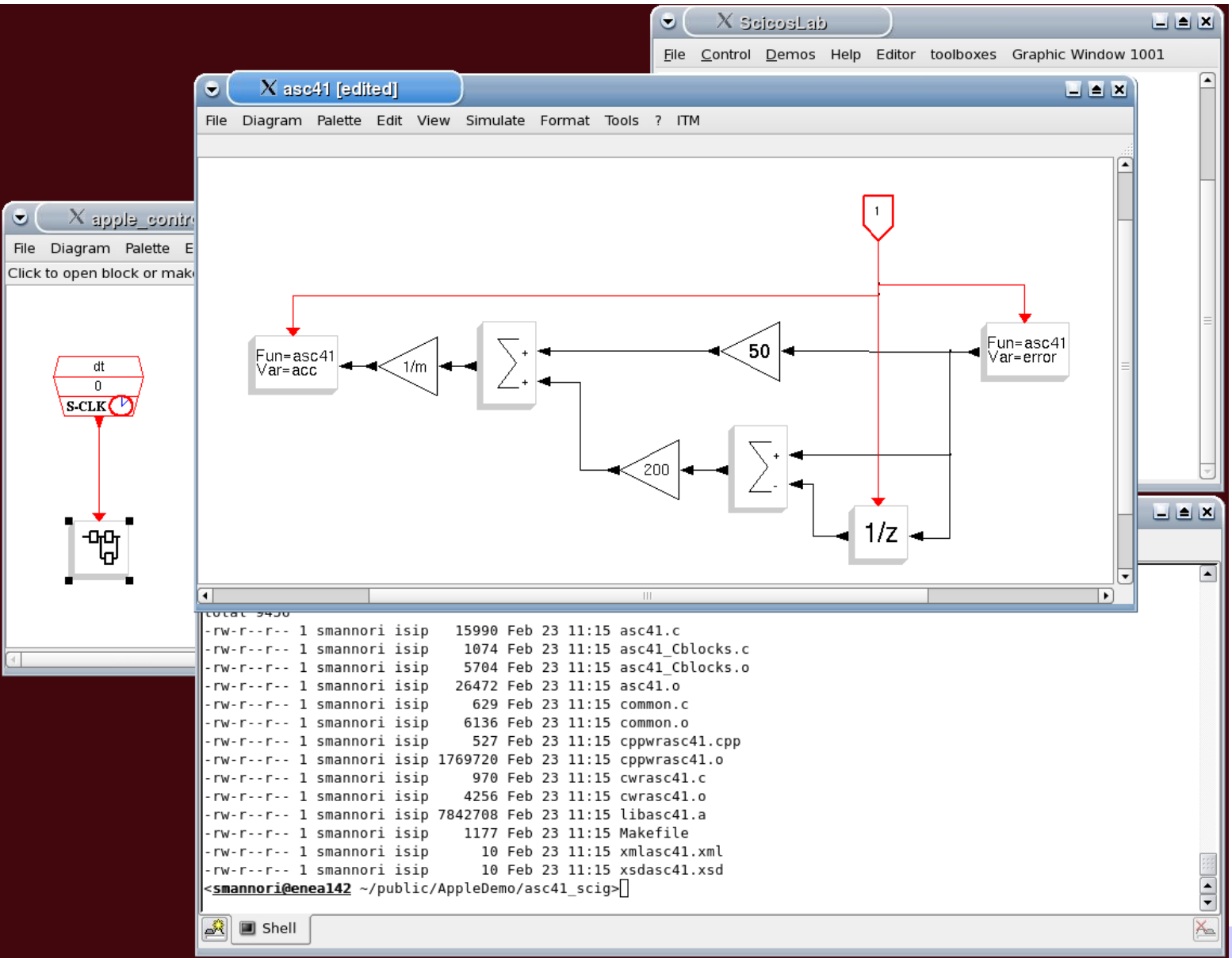
1. Design the control system in ScicosLab



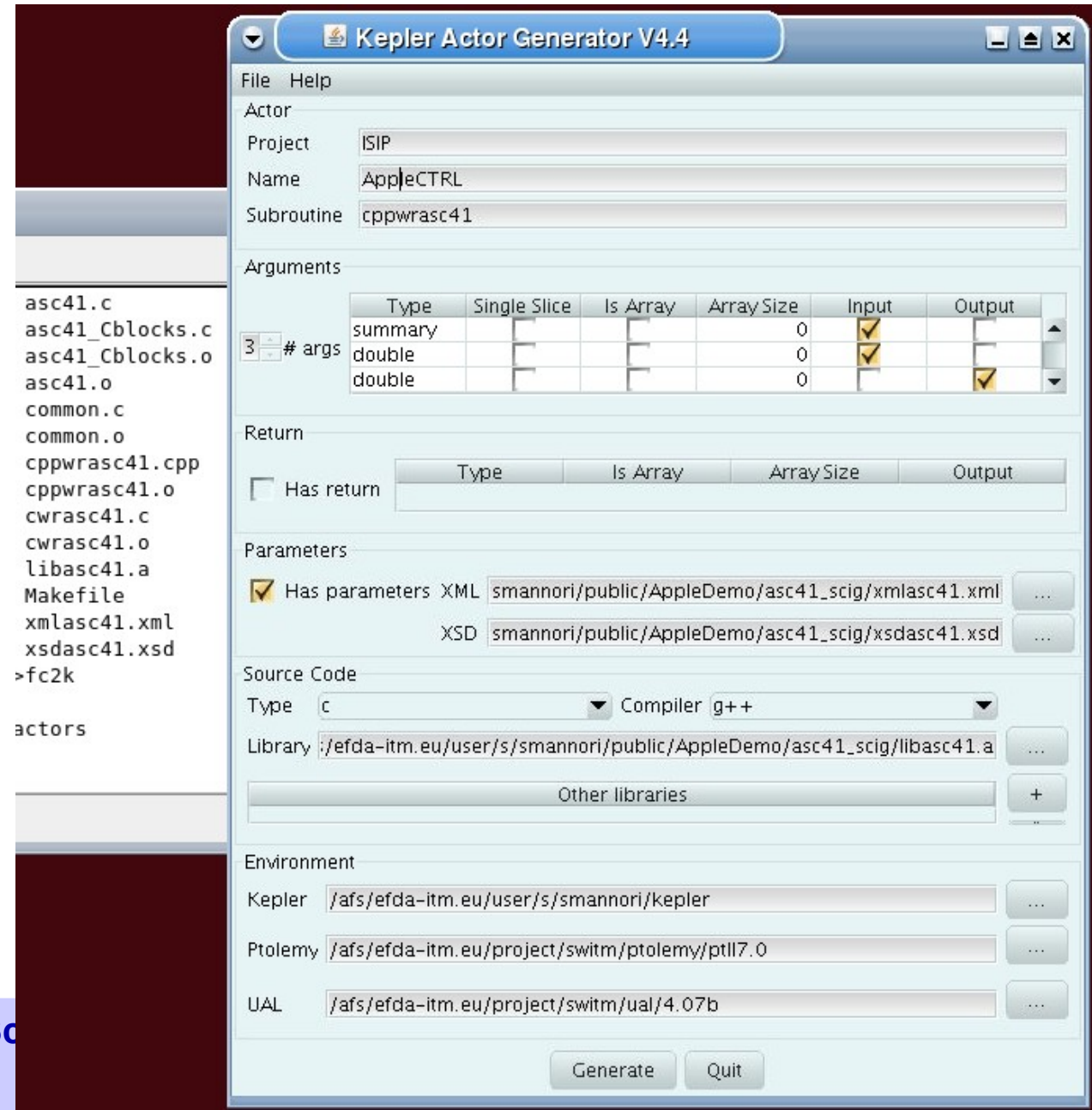
2. Prepare the controller in Scicos for code generation



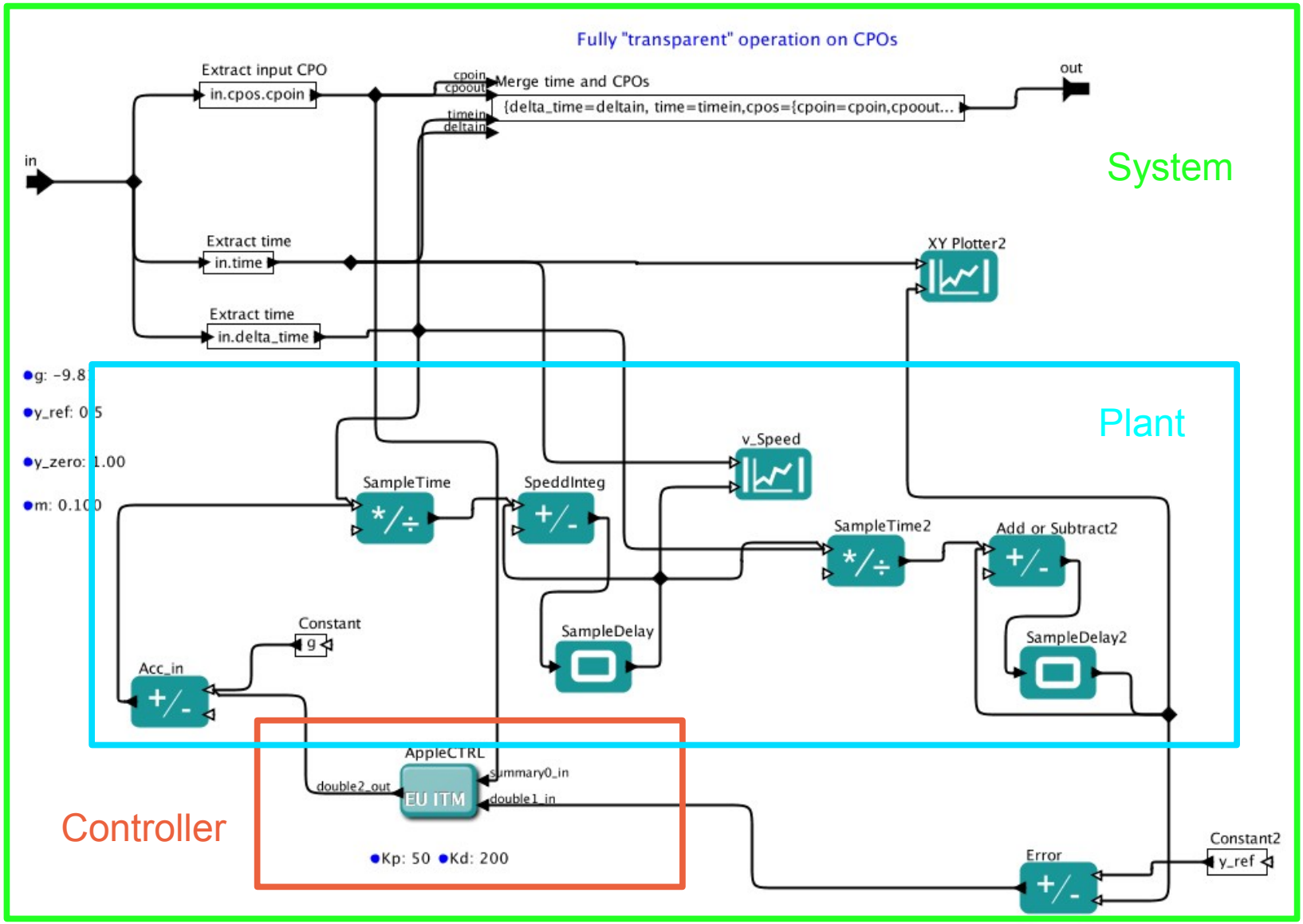
3. Generate the code using Scicos-ITM



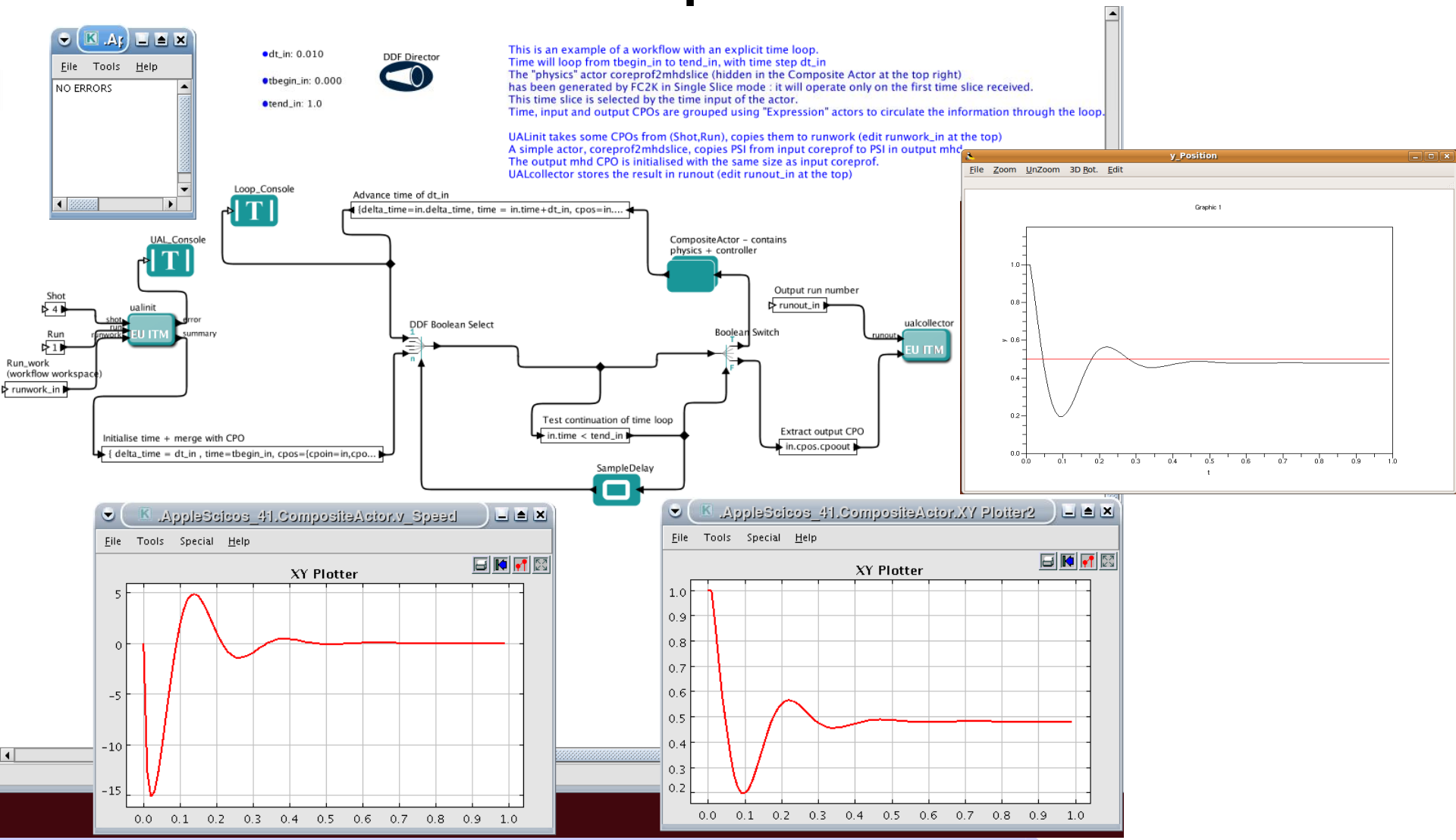
4. Generate a Kepler actor using FC2K



5. Insert the actor inside the work flow



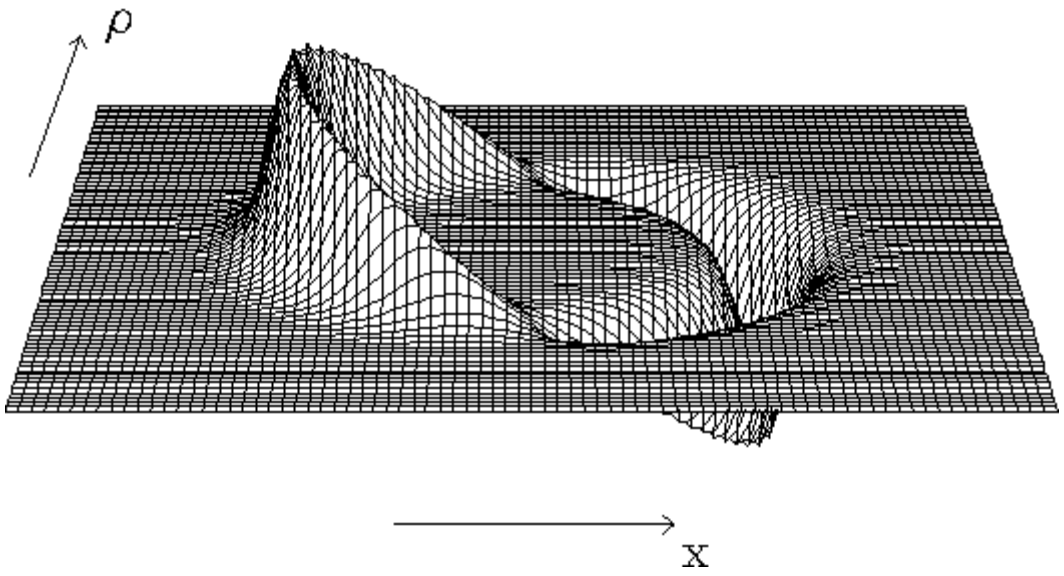
6. Run the Kepler simulation



Why you need a fusion power plant ?

Controlled Nuclear Fusion ...

$$\vartheta = -\alpha \ \mathrm{Tr}(\mathrm{K})$$



Controlled Nuclear Fusion as Warp Engine Starter