ORTD — The Open Realtime Dynamics Toolbox

Christian Klauer[1]
[1]Control Systems Group, Technische Universität Berlin
Kontakt: klauer@control.tu-berlin.de

May 2012

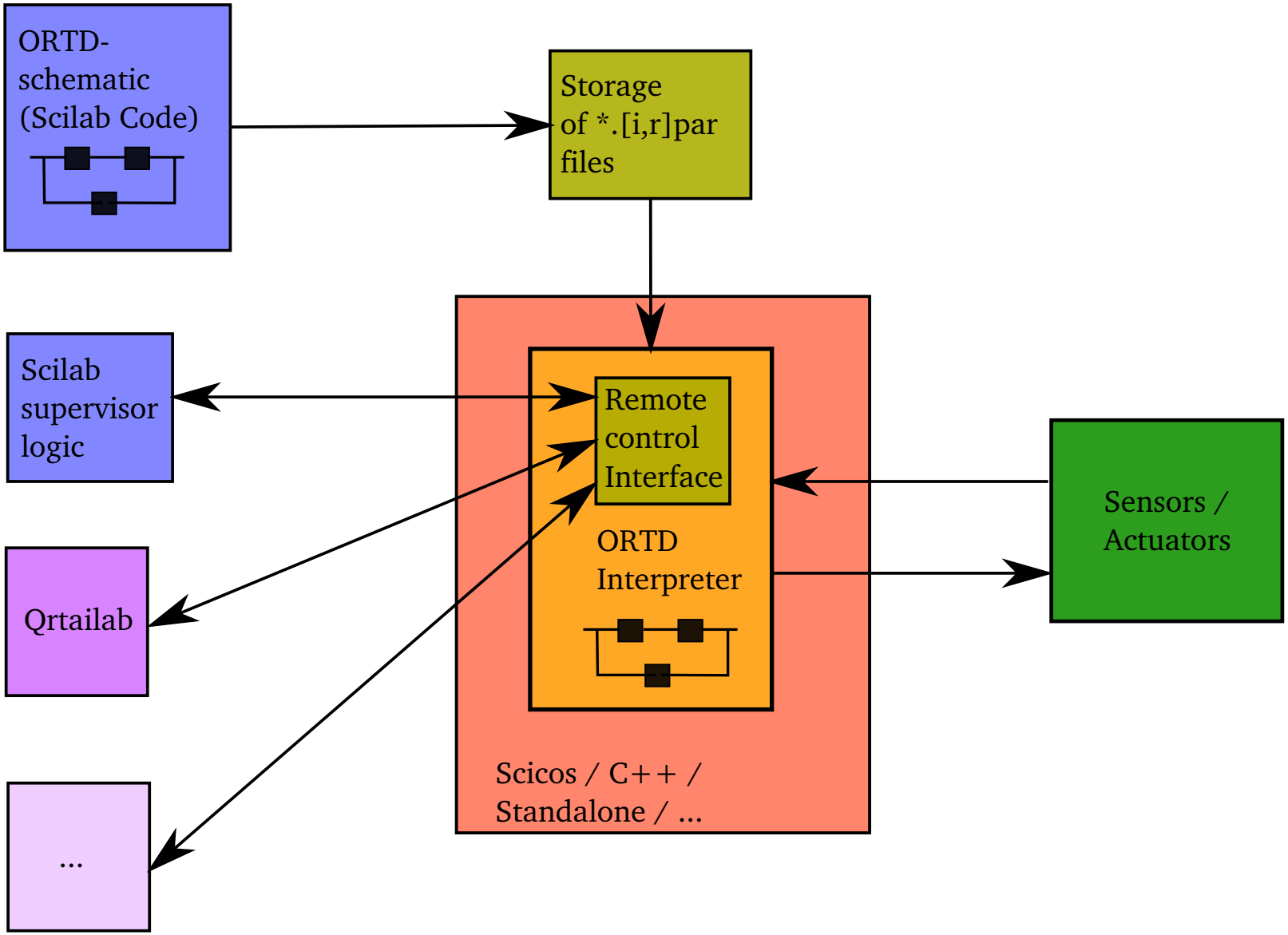
OpenRTDynamics is

- A simulator for time-discrete dynamical systems
- A Scilab-Toolbox for describing these systems in an block/signal based way.
- Additional modules (e.g. an remote control interface, state machines, parallelism using threads, math formula parsing, ...)

Compared to other systems it features

- A new way of defining schematics, which enables well structured code that is easy to maintain as projects get bigger.
- Multiple nested-schematics: Switching around sub-schematics and resetting them (like a restart).
- Because of an interpretation algorithm, schematics can be exchanged online.

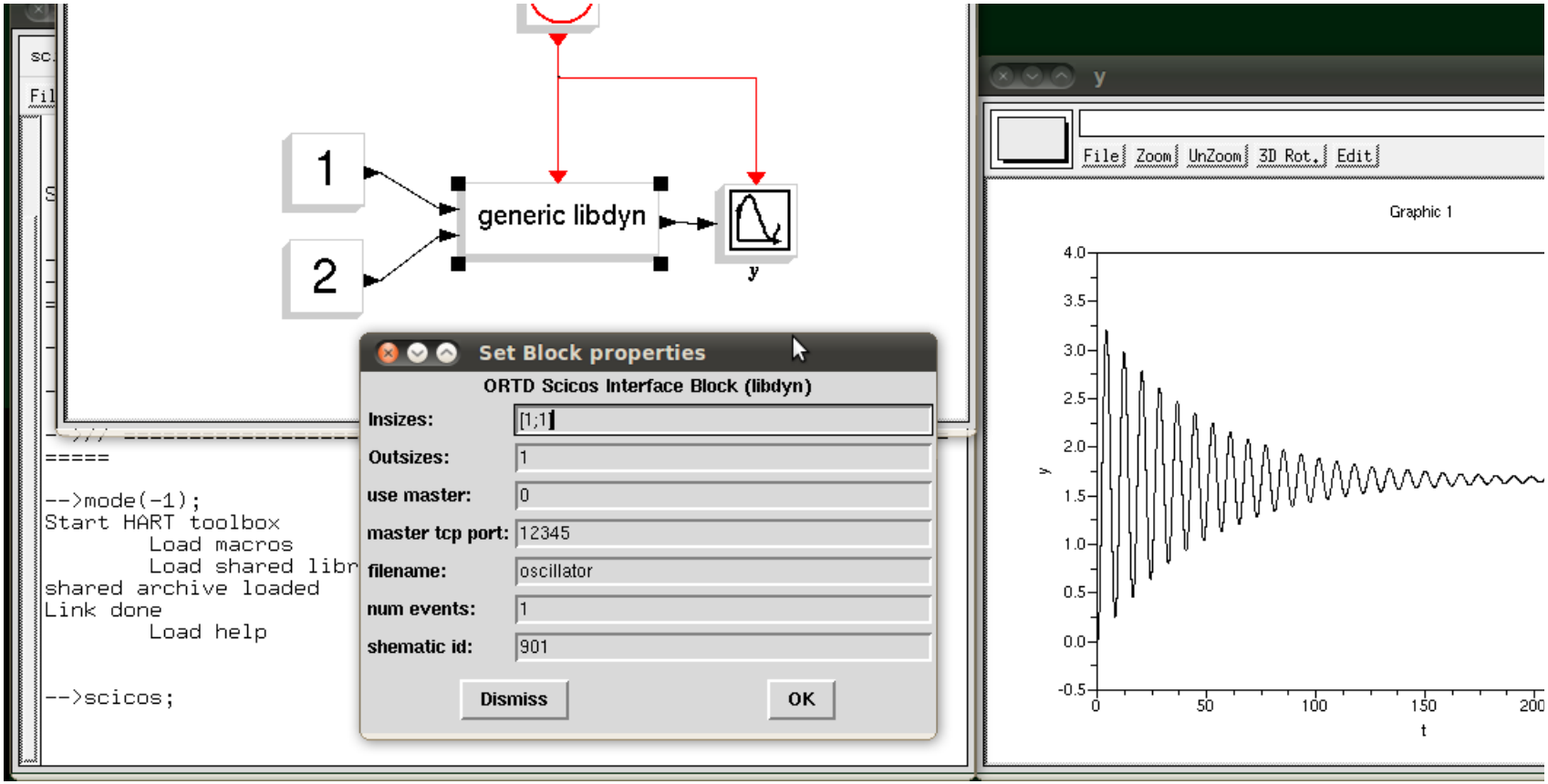
General Overview



The interpreter can be interfaced via

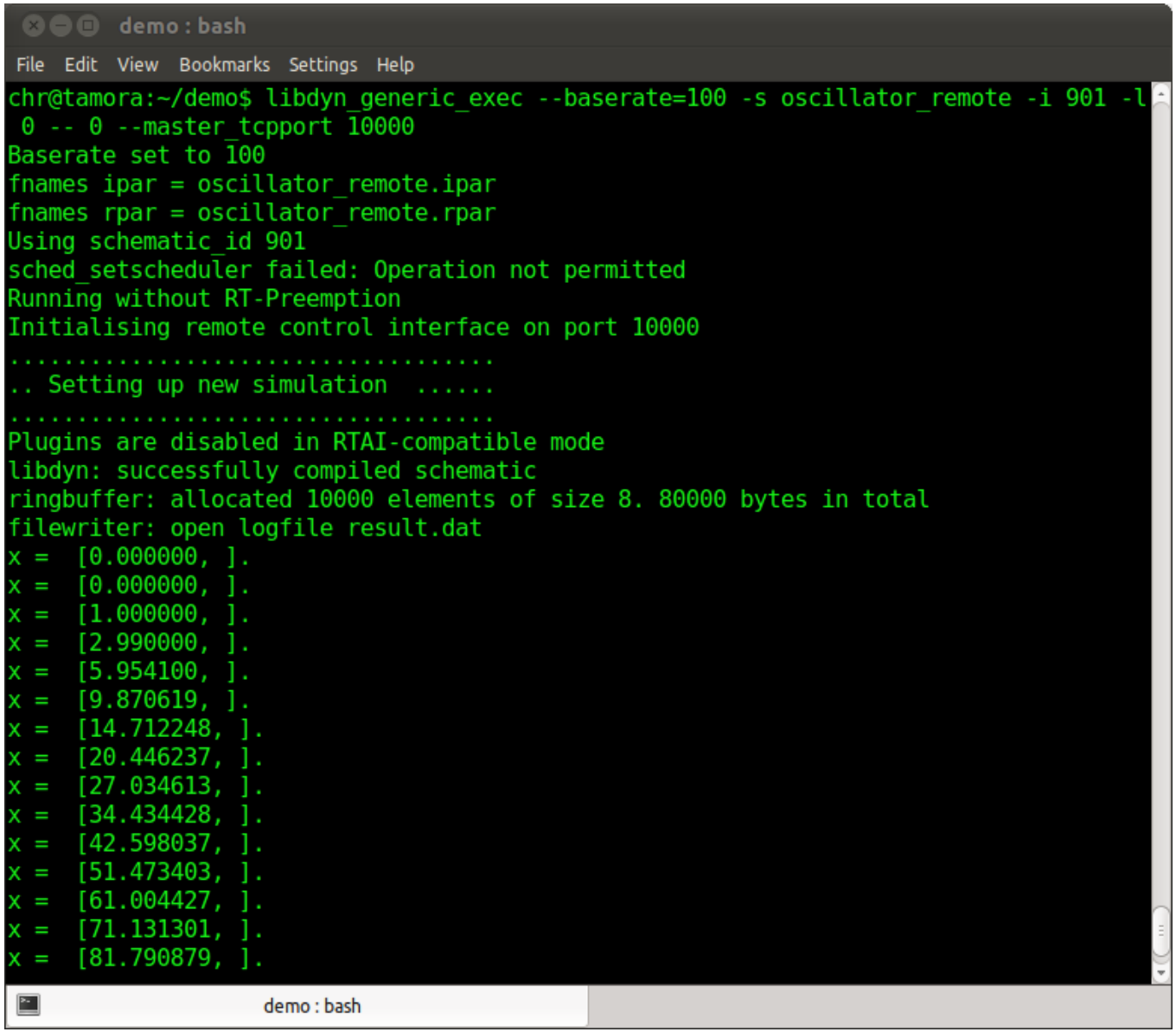
- The provided Scicos-Block
- The executable `libdyn_generic_exec`
- Third party C++ Code by linking to the shared library `libortd.so`

Interface to Scicos



- Specification of the in- and output port sizes along the name of the schematic to load.
- Multiple interface blocks within one Scicos diagram are possible

Standalone interface



- Simulation mode or real-time execution with RT-Preempt scheduling or using soft RT.

How schematics are defined:

- **Signals** are represented by a special Scilab variable type.
- **Blocks** are created by calls to special Scilab functions. They can take input signals and can give back new signal variables.

An Example:

- A linear combination of two signals ($y = u_1 - u_2$) would look like:

```
[sim, y] = ld_add(sim, defaultevents, list(u1, u2), [ 1, -1 ] );
```

- A time-discrete transfer function is implemented like this:

```
[sim, y] = ld_ztf(sim, defaultevents, u, (1-0.2)/(z-0.2) );
```

Please Note:

- For all calculations the toolbox functions have to be used. **Not possible:** `y = u1 - u2`

Some more explanation:

```
[sim, y] = ld_add(sim, defaultevents, list(u1, u2), [ 1, -1 ] );
```

- The variable `sim` is used to emulate Object-Orientated behaviour in Scilab.
- `defaultevents` defines a set of events that will be forwarded to the block. (Commonly set to `0`)

For Help:

- A list of blocks is available through the Scilab help browser, e.g. try `help ld_add`.

Definition: Within Scilab by writing a function that describes the blocks and connections:

```
// This is the main top level schematic
function [sim, outlist]=schematic_fn(sim, inlist)
   u1 = inlist(1); // Simulation input #1
   u2 = inlist(2); // Simulation input #2

  // sum up two inputs
  [sim,out] = ld_add(sim, defaultevents, list(u1, u2), [1, 1] );

  // save result to file
  [sim, save0] = ld_dumptoiofile(sim, defaultevents, ...
       "result.dat", out);

  // output of schematic
  outlist = list(out); // Simulation output #1
endfunction
```

- It takes the simulation object `sim` as well as a list of in- and outputs.

Generation: A set of function calls trigger evaluation of the functions describing the schematic.

```
defaultevents = [0]; // main event

// set-up schematic by calling the user defined
// function "schematic_fn"
insizes = [1,1]; outsizes=[1];
[sim_container_irpar, sim]=libdyn_setup_schematic(schematic_fn, ...
                           insizes, outsizes);

// Initialise a new parameter set
parlist = new_irparam_set();

// pack simulations into irpar container with id = 901
parlist = new_irparam_container(parlist, sim_container_irpar, 901);

// irparam set is complete convert to vectors
par = combine_irparam(parlist);

// save vectors to a file
save_irparam(par, 'simple demo.ipar', 'simple demo.rpar');
```

- The schematic is saved to disk by `save_irparam`.

A working example – part 3

Execution:

- This Scilab-Script will generate two files `simple_demo.ipar` and `simple_demo.rpar`, which contain a encoded definition of the whole schematic.
- These file are then loaded by the provided interpreter library and executed.

Superblocks: definition

Superblocks are introduced by writing a new Scilab function.

```
function [sim, y]=ld_mute(sim, ev, u, cntrl, mutewhengreaterzero)
    [sim, zero] = ld_const(sim, ev, 0);

    if (mutewhengreaterzero == %T) then // parametrised functionality
      [sim,y] = ld_switch2to1(sim, ev, cntrl, zero, u);
    else
      [sim,y] = ld_switch2to1(sim, ev, cntrl, u, zero);
    end
endfunction
```

- This example describes a superblock, which has two inputs `u` and `cntrl` and one output `y`.
- `mutewhengreaterzero` describes an parameter.
- **NOTE**: With the if / else construction a superblock can have different behaviour depending on a parameter! (This enables great possibilities for creating reusable code)

Superblocks: Usage

Once defined, the superblock can be used like any other ORTD-Block:

```
[sim, y] = ld_mute( sim, ev, u=input, cntrl=csig, ...
                    mutewhengreaterzero=%T )
```

How to implement feedback?

- A dummy signal is required, which can be used to connect a real block:

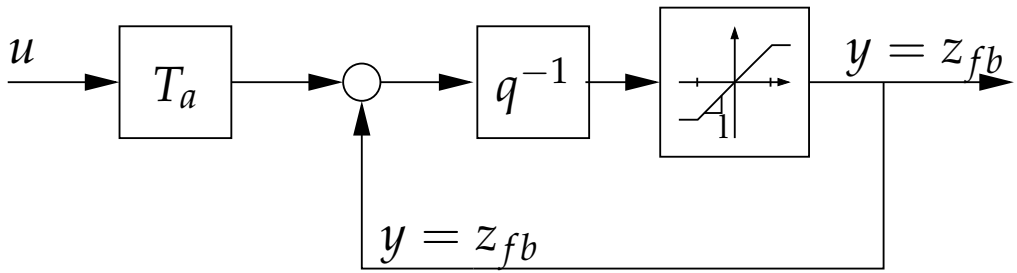
```
[sim, feedback] = libdyn_new_feedback(sim);
```

- Later in the ongoing code, the loop is closed via `libdyn_close_loop`, which means `feedback` is assigned to a real signal `y`:

```
[sim] = libdyn_close_loop(sim, y, feedback);
```

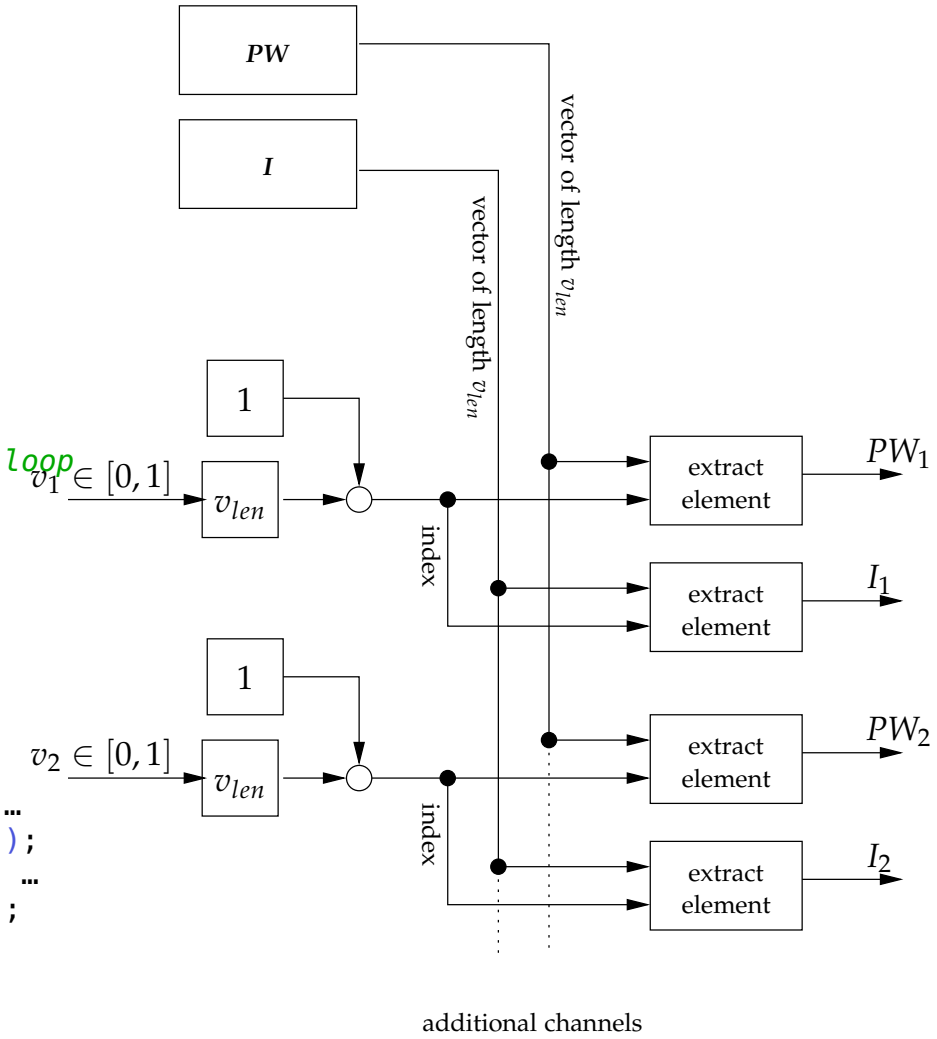
Feedback Loops: An example

```
function [sim, y]=limited_int(sim, ev, u, min__, max__, Ta)
  // Implements a time discrete integrator with saturation
  // of the output between min__ and max__
  //
  // u * - input
  // y * - output
  //
  // y(k+1) = sat( y(k) + Ta*u , min__, max__ )

    [sim, u__] = ld_gain(sim, ev, u, Ta);

    // create z_fb, because it is not available by now
    [sim,z_fb] = libdyn_new_feedback(sim);

    // do something with z_fb
    [sim, sum_] = ld_sum(sim, ev, list(u__, z_fb), 1, 1);
    [sim, tmp] = ld_ztf(sim, ev, sum_, 1/z);

    // Now y becomes available
    [sim, y] = ld_sat(sim, ev, tmp, min__, max__);

    // assign z_fb = y
    [sim] = libdyn_close_loop(sim, y, z_fb);
endfunction
```



Example: Lookup table for multiple channels

```
function [sim, I_list, PW_list]=ld_charge_cntrl_multich(sim, ev, v_list, Nch)
//
// Charge control for multiple channels using the same lookup table
//

   // Get table data and their lengths
   tabPW_ = CHCNTL.tabPW;
   tabI_ = CHCNTL.tabI;
   vlen = length(tabI_);

   // The vectors containing the tables
   [sim,TABI] = ld_constvec(sim, ev, tabI_ );
   [sim,TABPW] = ld_constvec(sim, ev, tabPW_ );

   // init output lists
   I_list = list(); PW_list = list();

   // loop
   for i=1:Nch // Create blocks for each channel by a for loop

      // extract normalised stimulation for channel i
      v = v_list(i);

      // calc index
      [sim, index] = ld_gain(sim, ev, v, vlen);
      [sim, index] = ld_add_ofs(sim, ev, index, 1);

      // look up the values
      [sim, I] = ld_extract_element(sim, ev, TABI, index, …
                                                 vecsize=vlen );
      [sim, PW] = ld_extract_element(sim, ev, invec=TABPW, …
                              pointer=index, vecsize=vlen );

      // store signals
      I_list($+1) = I; PW_list($+1) = PW;
   end
endfunction
```



Principle:

- Each state is represented by a whole sub-schematic. Thus, a state machine contains multiple of them.
- Only one schematic is active at once, representing the active state.
- Every time the active sub-schematic can cause switching to another state.

Extra features:

- All blocks within a schematic that is going to be inactive are reset.
- Possibility to share a memory among all schematics for realising global states (counters, ...)

State machines: implementation

A function (Superblock) is evaluate once for each state. Differentiation among schematics is achieved by a `select` statement

```
function [sim, outlist, active_state, x_global_kp1, userdata]=state_mainfn(sim, ...
                                    inlist, x_global, state, statename, userdata)
  printf("defining state %s (#%d) ... userdata(1)=%s\ n", statename, state, userdata(1) );

  // define names for the first event in the simulation
  events = 0;

  // demultiplex x_global
  [sim, x_global] = ld_demux(sim, events, vecsize=4, invec=x_global);

  // sample data fot output
  [sim, outdata1] = ld_constvec(sim, events, vec=[1200]);

  select state
    case 1 // state 1
      // wait 10 simulation steps and then switch to state 2
      [sim, active_state] = ld_steps(sim, events, activation_simsteps=[10], values=[-1,2]);
      [sim, x_global(1)] = ld_add_ofs(sim, events, x_global(1), 1); // increase counter 1 by 1
    case 2 // state 2
      // wait 10 simulation steps and then switch to state 3
      [sim, active_state] = ld_steps(sim, events, activation_simsteps=[10], values=[-1,3]);
      [sim, x_global(2)] = ld_add_ofs(sim, events, x_global(2), 1); // increase counter 2 by 1
    case 3 // state 3
      // wait 10 simulation steps and then switch to state 1
      [sim, active_state] = ld_steps(sim, events, activation_simsteps=[10], values=[-1,1]);
      [sim, x_global(3)] = ld_add_ofs(sim, events, x_global(3), 1); // increase counter 3 by 1
  end

  // multiplex the new global states
  [sim, x_global_kp1] = ld_mux(sim, events, vecsize=4, inlist=x_global);

  // the user defined output signals of this nested simulation
  outlist = list(outdata1);
endfunction
```

Examples for advanced features like

- State machines (`modules/nested`)
- Simulations running in threads (`modules/nested`)
- Mathematical formula parsing (`modules/muparser`)
- Vectors handling blocks (`modules/basic_ldblocks`)
- Calling Scilab from the simulation (`modules/scilab`)
- Remote control interface (`modules/rt_server`)
- Starting external processes (`modules/ext_process`)
- Timer for simulations running in threads (pending) (`modules/synchronisation`)
- Scicos to ORTD block wrapper (`modules/scicos_blocks`)

can be found within the directories `modules/*/demo`. Often they are ready to run and can be started by executing a simple shell script.