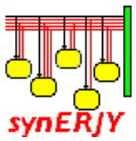


synERJY 5.1

Using the Programming Environment

Reinhard Budde
Axel Poigné
Karl-Heinz Sylla

Fraunhofer Institut

Autonome intelligente Systeme

Fraunhofer AiS

February 6, 2007

Preface

synERJY[1] is a programming language and a design environment for embedded reactive systems that combines two paradigms:

- *Object-oriented modelling* for a robust and flexible design.
- *Synchronous execution* for precise modelling of reactive behaviour.

Highlights are that

- *synERJY* provides a deep embedding of the reactive behaviour into the object-oriented data model.
- *synERJY* offers fine-grained integration of synchronous formalisms such as ESTEREL [1], LUSTRE [4], and STATECHARTS [3].[2]

The programming environment supports compilation, configuration, simulation, and testing, as well as verification by model checking. Behavioural descriptions may be edited in graphical or in textual form. Code generators for efficient and compact code in C and several hardware formats are available.

The *synERJY* language and its programming environment are developed at the Fraunhofer Institut Autonome Intelligente Systeme.[3] The programming environment is freely available with a public domain license.

Contact: {reinhard.budde,poigne,sylla}@ais.fraunhofer.de.

[1] *synERJY* may be read as **syn**chronous **e**mbedded **r**eactive **J**ava with a **y** for the sound.

[2] We recommend Halbwachs' book *Synchronous Programming of Reactive Systems* [2] as an excellent introduction.

[3] The development of *synERJY* has been partially supported by the ESPRIT LTR SYRF, "Synchronous formalisms", and the ESPRIT IIM Project CRISYS, "Critical Systems and Instrumentation".

Contents

1 Overview

The *synERJY* programming environment consists of three components.

- ***synERJY* Console**: the compiler bundled with a general control panel for the *synERJY* programming environment.
- ***synERJY* Simulator**: the step through simulation tool.
- ***synERJY* Graphic Editor**: the visual editor for hierarchical automata.

synERJY is a public domain software available at

`www.ais.fraunhofer.de/~ap/synERJY`

for Linux, Windows XP, and MacOS X. Target system generation depends on the availability of cross compilers. The present distribution supports system generation from Linux to the Hitachi Hi8 processor (used in the LEGO Mindstorms controller RCX) and some Atmel AVR micro controllers, MPC555, and some Texas Instrument DSPs. Example programs are included in the distribution.

2 Installation

Windows XP
: `synERJY.exe` installs the basic *synERJY* environment. Just run the executable and tick boxes as appropriate. Other set-up executables are provided for specific target architectures.

Linux
: Untar the distribution, and move the `synERJYlinux` distribution wherever appropriate. Then set the environment variable `SE_HOME` to .../synERJY.

Mac OSX
: Untar the distribution, and move the `synERJYdarwin` distribution wherever appropriate. Then set the environment variable `SE_HOME` to .../synERJY.

3 Getting Started

Running the environment. Copy an example program, e.g. `basic1.se` from the `target/host/examples` (resp. `target\host\examples`) directory of the *synERJY* distribution to some working directory. Preview the example file using your favourite editor[4]. Start the *synERJY* console

Windows XP
: by double-clicking on the file `basic1.se` in the directory `...\examples\Basic1`.

Linux
: by calling the command `synERJY` in a shell, and then loading the project `.../Basic1`.

Mac OSX X11 version
: as for Linux.

Loading a project. Use the File↦Load project menu entry[5] to start the load-project-dialog. Select the project `Basic1`. The project is loaded and the configuration class defined in the file `basic1.se` is compiled and added to the database of the compiler. Change the source file using your favourite editor. Now you may re-compile the project by using the File↦Reload menu entry. Even easier is to click the reload-button on the left hand side of the console window. Note, that the name `Rc` of the configuration class contained in the file `basic1.se` becomes the name of the configuration class in the console window. As you may guess, this class has a static method `main` to start the application.

Code generation for Simulation. Now you may want to generate code for simulation. *synERJY* can generate code to be used with the simulator only, if a C compiler for the operating hosting the *synERJY* environment is available. All different flavours of UNIX have `gcc`, some UNIXes have other C-compiler, too. For win32 systems either a licence for VisualStudio must be available, or public domain software must be installed which features a C-compiler, for instance `Cygwin` or `Mingw32/MSys`. The operating system MacOS X used for the Apple Macintosh has a `gcc` compiler. By either selecting the Make↦Build menu entry or by clicking the Build button the

[4] A syntax definition file for *synERJY* is provided for vim :-)

[5] Shortcuts are indicated in the menus by letters. We use `Ctrl-<Key>` for invocation.

compiler generates C code and then calls the C compiler to compile and link the object file against a small runtime library which is part of the *synERJY* distribution. If only C code is to be generated, use Make↦Compile. Then the Build button will be renamed to Compile with according behaviour. Using Make↦Build toggles again.

The C-code generated by the *synERJY* compiler is stored in the temporary files `se.sim.Rc.h` and `se.sim.Rc.c`. Remember, that `Rc` is the name of the configuration class from the file `basic1.se`. All temporary files are prefixed with `se.`. If you do not use filenames starting with this prefix by yourself, it is safe to remove temporary files by `rm se.*`.

Simulation By selecting either the Utils↦Simulator menu entry or by clicking the Simulator-button the simulator is started. You may trigger an instant by clicking the React button. By clicking an input signal it will be present in the next reaction. Make input signals present, trigger an instant, look at the source code and check whether that happened what you expected :-). You may edit the source and look for effects. Now we have stepped through one elementary debugging cycle. For generating code for a target micro-controller see section 4.5.

4 The synERJY Programming Environment

4.1 The Console

The *synERJY console* controls the compilation of *synERJY* programs and allows to call other components of the programming environment. The *synERJY* compiler maintains a database in which all classes from all files which have been loaded during the actual session are stored. *synERJY* programs can be composed from different files, of course. But: as the database indicates, there is no means for separately storing the compiler outcome of a single compilation (into a `.o` file e.g. Such a feature is usually called separate compilation). This is intentionally: The *synERJY* compiler does a lot of global optimisations before code is generated: for example, class hierarchy analysis to replace dynamic by static binding of methods calls, if possible. Thus a typical development cycle is: load a project, generate code, simulate, edit a file to remove an error, re-load that file only, generate code, simulate, and so on.

Fig. 4.1 is a screen-shot of the console window after loading a file and generating simulation code.

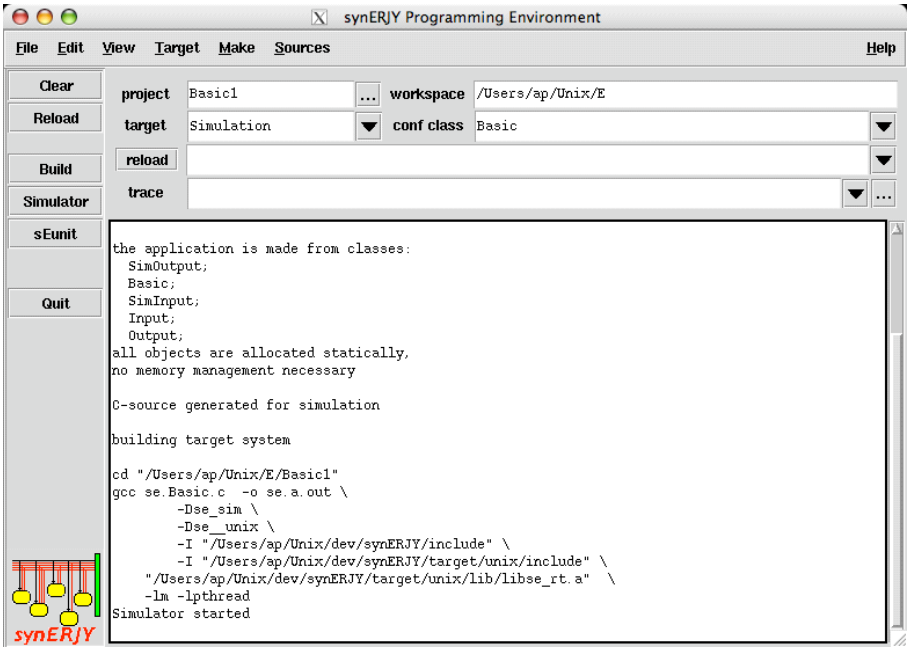


Figure 1: Screen-shot of the *synERJY* console

The main text widget displays the command history and the outputs from the compiler. Most menu entries, display widgets, and buttons should be self-explaining. Note that buttons are shortcuts for menu entries: *Clear* clears the main text widget, whereas *Reload* reloads the chosen project after resetting the internal compiler database, i.e. all classes from all files loaded during the actual session are removed.

Build generates code according to the target. The target can be selected as a menu entry. The choice of targets depends on the platform. Targets may be the simulator (see Section 5) provided by the *synERJY* environment, an executable on the respective host, or on some micro processor or DSP boards. Target code is executed by pressing the *Simulator* button (which is renamed to *Run* in case of an executable, or to *Upload* if a processor board is the target).

To avoid reloading all files of a project while editing only one file, the respective file may be chosen in the combobox on the right with label `reload` which actually is a button. Pushing the button reloads the file.

Quit terminates the *synERJY* console.

4.2 Workspace and Projects

The *synERJY*-environment supports a light-weight project handling. A project consists of a directory containing a *project resource file* `_.seprj`. The project resource files specifies the *synERJY* files (`.se` files), the *synERJYcharts* files (`.sec` files), the trace files (`.setr` files), C files (`.setr` files), header files (`.h` files), and cclibs (`.lib` files) belonging to a project. Files may added and removed using the *project panel* Fig. 4.2.

The project panel pops up if the entry Project of the View menu is selected.

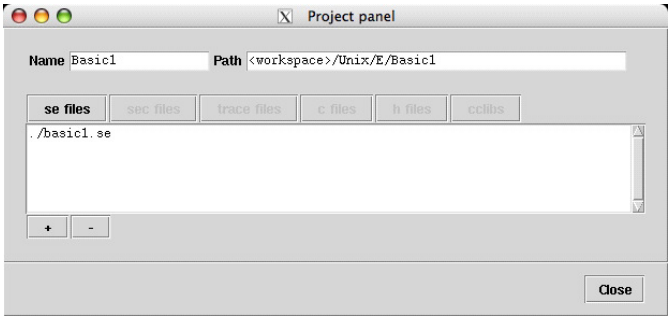


Figure 2: The synERGY project panel

All these files have a relative address within *synERJY*:

- relative to a project
- relative to the *synERJY* home directory
- relative to a workspace

In Fig. 4.2 the file `basic1.se` is defined relative to the project as indicated by the prefix "`./`" (resp. "`.\`" for Windows XP). If the file is addressed relative to the *synERJY* home directory the prefix will be "`<synERJY>/`", and if relative to the workspace the prefix will be "`<workspace>/`" (rsp. "`<synERJY>/`" and "`<workspace>/`" for Windows XP).

A *workspace* is just some other directory which can be chosen using the preference panel (see Section 4.3). Default workspace is the home directory of the respective user. It is required that a project is a sub-directory of the workspace directory.

The idea of relative addressing is to provide flexibility. Projects may be moved to an arbitrary position within a workspace if only relative addresses

are used. Similarly *synERJY* may be installed at an arbitrary place but one may still refer to *synERJY* files provided by the installation. Of course, files may be which are *not* in the project directory, the workspace, the *synERJY* home directory or its sub-directories, but at the price of using absolute addresses, which may make moving and porting projects more difficult.

Projects may be stored or opened, the latter by selecting the entry Open of the Project menu (or by double-clicking a project file in Windows.

A project resource file may be edited. Typical information stored is of the following form:

```
// settings for the compiler and the main environment
target Simulation;
set project kind = C-code;
load file = ./basic1.se;
set configuration class = Basic;
load trace file  = ./basic1.setr;
```

An overview of all commands is given in Section 4.6

4.3 Preferences

A few preferences concerning the workspace, and the size and weight of the fonts may be set using the *preference panel* (Fig. 4.3).

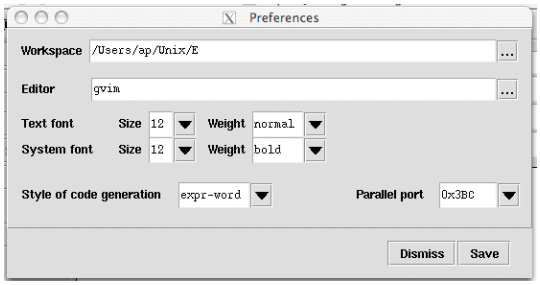


Figure 3: The synERGY preference panel

Preferences are stored in as a resource file `.serc` in the *synERJY* home directory. All the commands in Section 4.6 may be used within the resource file.

While most of the options are self-explaining, two of them need comments:

- *synERJY* allows to generate control code in different styles. The most readable one is `expr-word` where the control structures is given in terms a sequential circuit, i.e. Boolean equations representing the logic and the control register allocations. In the `expr-bit` each Boolean signal or register is coded as a bit in a `char` word. The last style encodes the control structure as a jump table, which is quite an efficient implementation for a micro controller.

- On Windows machines, the parallel port is often used flashing code to micro-controllers. The port name may vary. For convenience some common port names are provided to be set using the respective combobox.

4.4 Environment Variables

For some applications such as Matlab, Scilab, and the AVR toolset it is necessary to set environment variables that should refer to the application directories. Environment variables are set using the menu entries Edit↦Environment variables).

4.5 Building an Application

Often software is built from *synERJY* classes defined within one file only, and then is executed in simulation mode. Then it is sufficient to load the *synERJY* file, to build the binaries, and to call the simulator.

If the software consists several files including external C functions it usually is more convenient to define a project.

One should be aware that *synERJY* compilation usually consists of several steps:

- Loading of *synERJY* files builds the internal data structure, typechecks, and applies semantic checks on class level such as for causality and time races.

- The building step links the objects, applies semantic checks on the level of the application, and it generates the executable.

- The execution step runs the executable on the respective hardware.

Since building the executable often implies that first C code is generated, and then the executable using a cross compiler it is sometimes convenient only to generate C code. Hence the entry of the *Make* menu.

An experienced user may prefer to use Makefiles. These may be defined using batch commands as specified in the next section.

4.6 Commands

synERJY may be used in "batch mode" by using options `se -f <file>` or `se -b <file>` when calling the *synERJY* compiler. The file called should consist of a list of commands. The `-f` starts the interactive GUI after the commands have been executed, `-b` terminates after the commands have been processed.

Alternatively, the character `%` may be used as in

```
se -b ¨%set target = atmel; load file = can_test.se; make C-code;¨
```

to treat the `-f` keyword value as commands instead of a file name. These so-called immediate commands are an easy way to call the *synERJY* compiler from `make`.

The following settings/commands are supported:

```
// synERJY - resource and command file

// font used for menu entries and buttons
set system font weight    = normal; // normal and bold supported
set system font size      = 10;     // 8,10,12,14,16,18,20,22,24 supported

// font used for the console window
set font weight           = normal; // normal and bold supported
set font size             = 10;     // 8,10,12,14,16,18,20,22,24 supported

// font used for the simulator windows
set simulator font weight = normal; // normal and bold supported
set simulator font size   = 10;     // 8,10,12,14,16,18,20,22,24 supported

// font used for the graphic editor windows
set graphic font weight   = normal; // normal or bold supported
set graphic font size     = 10;     // 8,10,12,14,16,18,20,22,24 supported

// settings for the graphic editor
set graphic width     = 913;    // approximately a Din A4 page (on my machine)
set graphic height    = 765;    // ...
set graphic file      = name.sc;// default file-name for graphic-editor
set print size        = a5;     // [a4|a5] size for generated postscript

// settings for the compiler and main environment
set load file             = name.se; // for re-load, shown in console window
set configuration class   = Class;   // where main is, shown in console window
set workspace             = /home/user;
                                     // set the workspace
```

```
set editor                = gvim;    // editor used if edit invoked from menu
set build                 = make;    // only make supported
clear build;                         // reset build to internal method
target simulation;                   // executable for simulation
target Host;                         // executable for target host computer
target Simulink;                     // executable for Simulink
C-code target             = ...;     // executable for target platform ...
target VerilogSimulation;            // executable for Verilog simulation
Verilog target            = ...;     // executable for Verilog platform ...

set simulation directory  = sim;     // use separate dir for simulation code
clear simulation directory;          // no separate dir for simulation code
set target directory      = tgt;     // use separate dir for target code
clear target directory;              // no separate dir for target code

// commands for the compiler
make binary;                    // generate C-code, compile and link
make C-code;                    // generate C-code only
make build;                     // generate C-code, call make

// load project
set project kind = name.sepr    // load synERJY project
                                // kind = C-code | Verilog

// loading files
set sefile  = mmm.se;           // add synERJY file
set sefile += nnn.swe;          // add further synERJY files
clear sefile;                   // remove all synERJY files

set secfile  = mmm.sec;         // add synERJYcharts file
set secfile += nnn.sec;         // add further synERJYcharts files
clear secfile;                  // remove all synERJYcharts files

set trace file  = mmm.setr;     // add synERJY trace file
set trace file += nnn.setr;     // add further synERJY trace files
clear trace file;               // remove all synERJY trace files

set cfile  = "xxx.c";           // C libraries to add for linking (internal mode)
set cfile += "zzz.c";           // add further libraries
clear cfile;                    // remove all C libraries

set cclib  = "xxx.a yyyy.a";    // C libraries to add for linking (internal mode)
set cclib += "zzz.a";           // add further libraries
clear cclib;                    // remove all C libraries

set vfiles  = "xxx.v";          // Verilog file
set vfiles += "zzz.a";          // add further Verilog files
clear vfiles;                   // remove all Verilog files
```

```
clear database;                // ...
clear window;                  // ...
print               = "text";  // print text onto console window

load file           = name.se; // load file
set workspace       = "path"   // set the workspace path
set project path    = "path"   // set the project path

check configuration = Class;   // typecheck application build upon Class

// commands to set the target platform
target simulation;             // target is the build-in simulator
target Host;                   //       ... the host platform
target simulink;               //       ... Simulink
target Makefile;               //       ... as provided by a Makefile in the
                                           workspace directory
C-code target = name;          //       ... the platform denoted by name
target VerilogSim;             //       ... the Verilog simulation environment
                                       (using Cver and GTKwave)
Verilog target = name;        //       ... the Verilog platform denoted by name


// Miscellaneous
execute graphic editor;        // ...
execute editor;                // see "set editor=..."
execute file = name;           // read command file name
execute file = "%commands";    // interpret immediate commands
execute file = "!command";     // execute system command (e.g. date)
execute file = "<command";     // read commands from system command (be careful)
set parallel port = "hex"      // set the parallel Port for windows
set code style    = "style"    // set the code style
                               // expr-word, expr-bit,jmp-sty
quit;                          // ...

// commands for the simulator
load configuration file   = name.sim; // load setting (binary,windows) as stored
save configuration file   = name.sim // store setting (binary,windows)
show object browser;                 // ...
show trace browser;                  // ...
show configuration signal = all;     // show all signals (incl. local)

// environment variables
set Matlab directory = path // path of the Matlab directory
set Scilab directory = path // path of the Scilab directory
set AVR directory    = path // path of the AVR directory
```

4.7 Unit Test

synERJY programs can be tested using traces. A trace specifies the input and output signals at instants, the format of an instant being

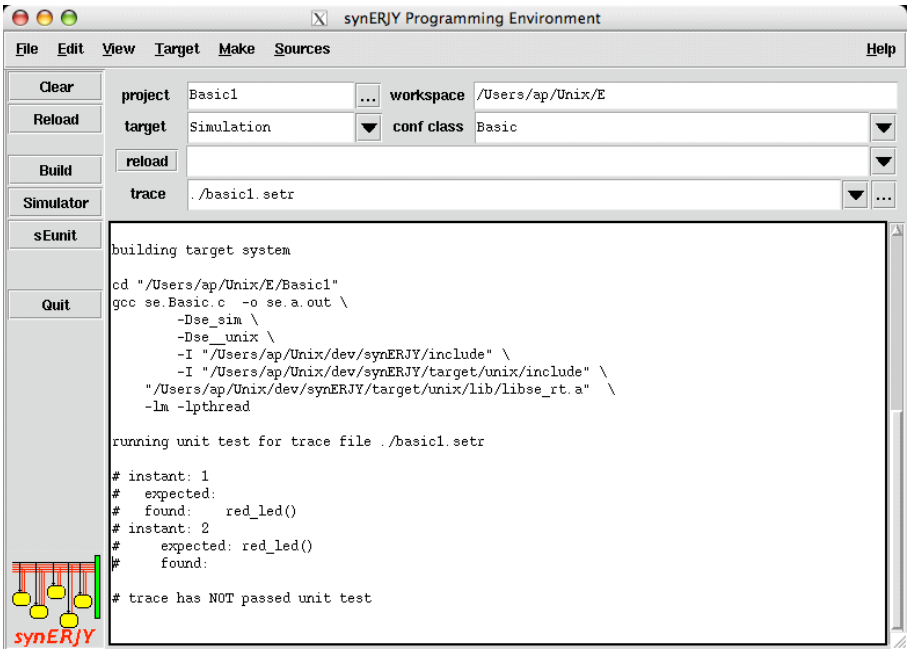
$$input_1 \ \ldots \ input_m \texttt{ -> } ouput_1 \ \ldots \ ouput_n\texttt{;}$$

Inputs and outputs are of the form *signal_name*() for pure signals, and *signal_name*(*signal_value*) for valued signals.

A trace file may be generated using the simulator (cf. 5) or may be edited by hand. Each trace file consists of one trace. A trace file generated by the simulator may show additional commands such as the command `break` which sets a break point for running the simulator. Comments are line-wise starting with `//`.

A unit test may be comprised of several traces. A trace file are either loaded using the menu entry File↦Load trace file, or by adding it to a project. The latter are stored as part of a project, the others are temporary: they will be discarded when resetting or closing the console window.

A unit test is executed by pressing the button `sEunit`. The button is active only if trace files are present. If the test passes the button flickers in green, if it fails in red. Additionally a message is displayed in the console window, e.g.



The message states at what instants conflicts arise between the expected behaviour and the behaviour found.

In the actual case the *synERJY* program is

```
class Basic {
    static final time timing = 250msec;
    Sensor button  = new Sensor(new SimInput());
    Signal red_led = new Signal(new SimOutput());

    public Basic () {
        active {
          loop {
              await ?button;
              emit  red_led;
              next;
          };
        };
    };

    public static void main (String[] args) {
       while (instant() == 0) {};
    };
}
```

(cf. `user_basic.se`) with the trace loaded being

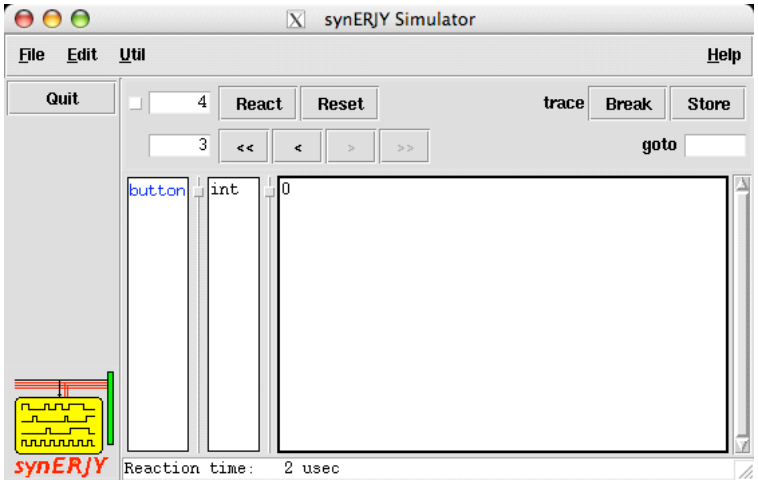
```
button() -> red_led();
button() -> ;
-> red_led();
->;
```

5 The *synERJY* Simulation Environment

The simulation environment consists of several components which allow to trace a programs control flow and signals. It provides recording and replay facilities for traces. The simulator can be started either from the *synERJY* console(see Section 4) or stand-alone, e.g. by typing the shell command `sim`.

5.1 The Simulator Console Window.

Introduction. The simulator runs as a process on its own. The simulator console window is the top-level window for interacting with the simulator.

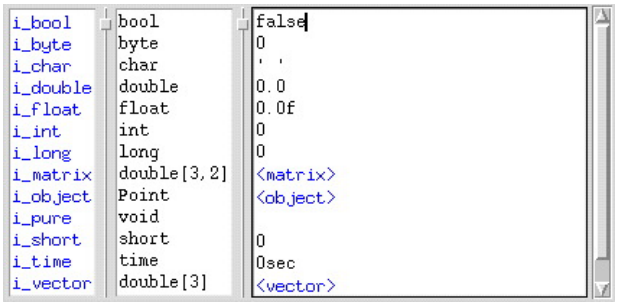


The simulator console window allows to step through an *synERJY* application, displaying instant by instant. The two main functionalities are:

- defining the status and values of input signals, and
- running a program step by step.

Text widget at the bottom displays system messages such as the reaction time which is the time used by the simulated *synERJY* program (not including the time needed for simulation). Execution time, of course, depend on the target machine, hence the reaction time shown specifies the time used if the program is run on the same machine that executes the simulation. It has little bearing for other target machines, except that the ration of reaction times for different instants indicates which computations might be costly.

Defining the status and values of input signals. The input widget



of the simulator console window three subwidgets:

- the left widget displays the signal names and indicates by colour whether a signal is present or absent. They are displayed in blue if absent and in red if present. The user can determine their status via the left mouse button. A simple click on a signal name toggles presence and absence (red and blue). The signal will be present for only one instant. A click with additionally the shift button pressed will turn a signal name green; this means that presence will persist. A simple click will turn the green to red.

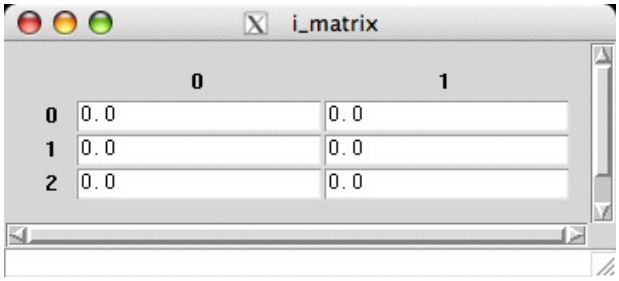
  The signal value used for the next presence of the signal must be typed into a text widget. Hitting the return key sets the signal to present while checking whether the corresponding values are well-formed. Alternatively, the signal can be set present using the toggle described above, also checking the well-formedness of values.

  In case of errors direct feedback is given in the entry line on the bottom of the widgets

- the middle widget displays the signal types.

- the right widget displays the signal values. It is a restricted text widget for specifying the values. If the return key is hit, the values are checked for being well formed and the respective signal is set to red. Errors will be indicated in the error message entry at the bottom of the simulator console window.

For structured values such as vectors, matrices, or objects only the kind is indicated as active text. Clicking on the signal name opens an input panel such as



Values can be set as suggested above for simple signals.

Running the Simulator in Single Step Mode. The upper widgets of the simulator console window

consists of several buttons:

- The React button causes execution of a single step.
- The Reset button resets the simulator.

These are the main actions for driving the simulator. One might step backward and forward through the instants already executed by the following buttons:

- Button < : one instant backward
- Button > : one instant forward
- Button << : to the very first instant
- Button >> : to the very last instant

Inserting a positive integer in the `goto` widget, and hitting the return key, brings you to the respective instant.

Terminology. We distinguish between three kinds of instants:

- the *next instant*: the next instant to be executed. The number of the present instant is shown in the box in the upper left corner.
- the *previous instant*: the last instant executed.

- the *displayed instant*: the instant that has been chosen by stepping backwards or forwards or by using the goto widget. The number of the respective instant is shown below. Just after a reaction the number of the previous instant is displayed.

There are two more buttons for recording traces.

- The Break button sets a break in the trace. The button is a toggle `Break`↔`Unset`: if a break is set it may be unset.
- The Store button stores the present trace into a file.

Traces are stored for a replay: the stored sequence of the signals presences and their values can be extracted automatically from a file and used as input for another test run. Replaying is initiated from the Utils↦Open trace file menu entry. An opened trace file will result in a slightly modified simulator console window.

The small box on the left of the simulator console window turns to red. The additional button

- The Run button executes all of the activated trace files up to the next break (the end of a trace file defines a break by default).

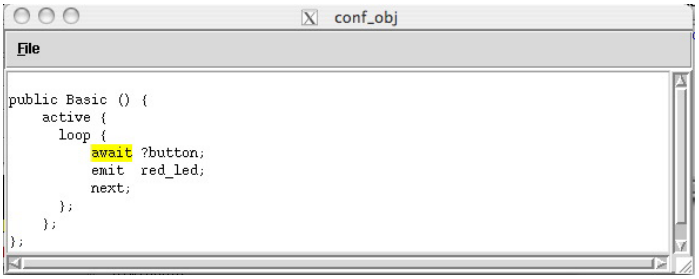
5.2 The Object Browser.

The object browser is the main interface for organising the configuration of the *synERJY* simulator. It displays the object hierarchy. A "+" in front of an object name indicates that there are sub-objects to the respective object. Clicking with the left mouse on "+" opens the hierarchy which can be closed again by clicking on "-". For each object, a *pop-up menu* is displayed when clicking the left mouse button on its name. There are three entries:

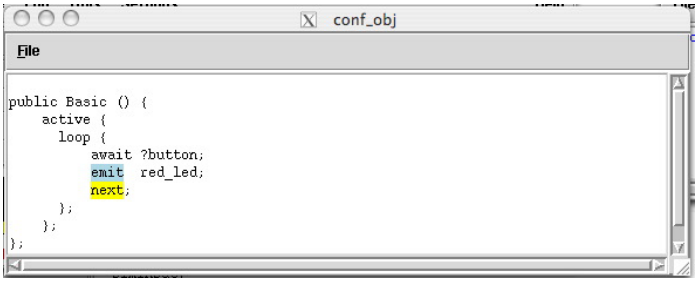
- **Show Class** displays the respective class.
- **Show Animation** opens the animator for the respective object.
- **Show Traces** calls the Trace Browser that displays the trace of the signals and attributes of the respective object.

5.3 The Animator.

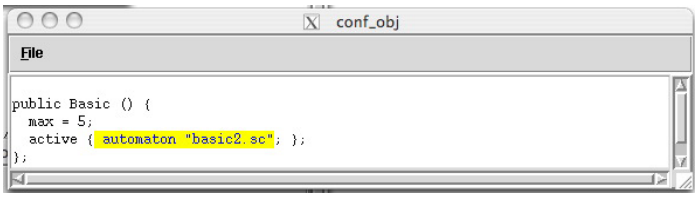
The animator highlights the halting points and the emitted signals of the reactive code. Selecting "Show Animation" opens a text window displaying the constructor of the respective object.



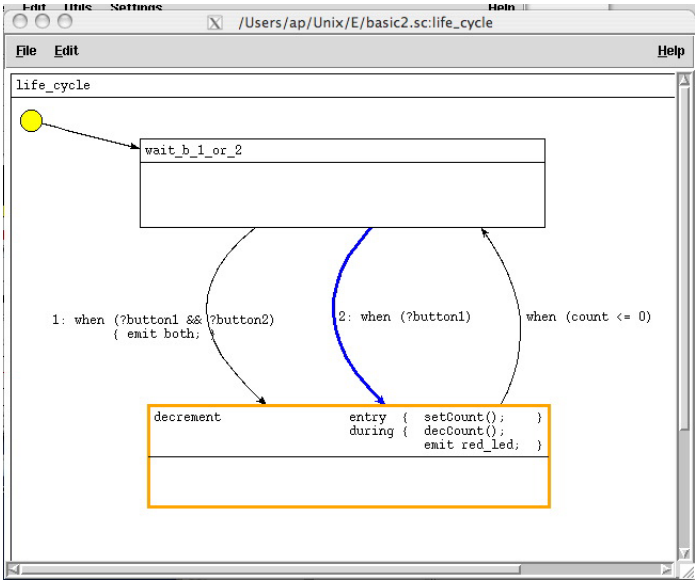
The halting points are marked by a yellow background, the signals emitted by a light-blue background.



Within such code there may be references to reactive reactive or graphical presentations of automata. These are indicated by a blue foreground as in

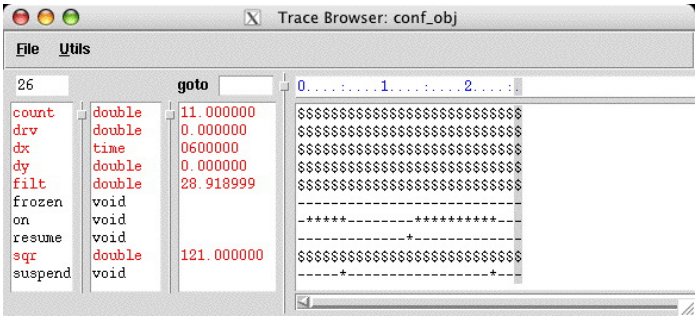


The yellow background specifies that the automaton is active. The active state in such an automaton is highlighted in orange while the transition taken again is highlighted in blue as in



5.4 The Trace Browser.

The trace browser displays the status and the value of all the signals of an object, and the values of all attributes.



The panel displays from the left:

- *name*: A signal is highlighted in red if it is present at the displayed instant (the displayed instant is indicated in the upper left widget)
- *type*
- *value*
- *instant number*: the small window indicates by special characters and numbers the instants displayed (modulo 100).

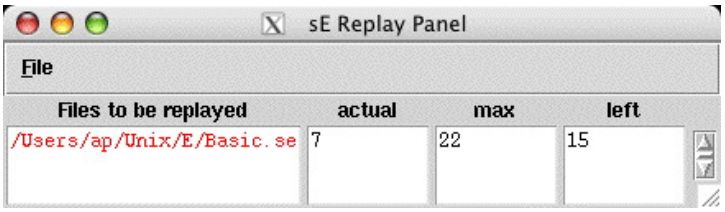
By clicking with left mouse button, the instant displayed under the mouse cursor becomes the displayed instant. By marking a region of instants (by clicking and dragging the left mouse button), all the values of this region will be displayed in a separate window, a so called *Value Sample Panel* (see below).

- The *field traces*. The lower right large widget indicates absence of a signal ba a "-". Presence of a pure signal is indicated by "", and that of a valued signal by by "$". For Boolean-valued signals we use "t" and false to indicate the value in case that the signal is present. The highlighted bar indicates the displayed instant.

Inserting a positive integer in the goto widget, and hitting the return key, moves to the respective instant.

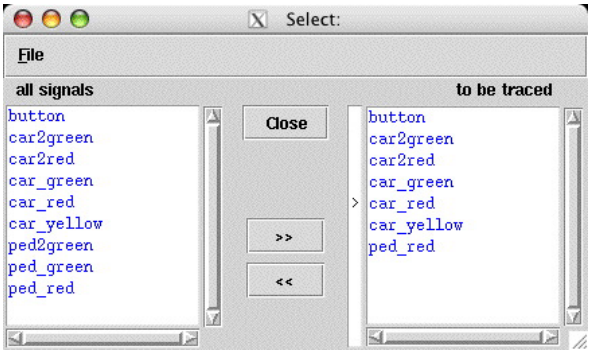
5.5 The replay panel

allows to organise the replay of trace files. Several trace files may be active. Trace files are (de-) activated or deleted using a pop-up menu. The menu is opened by clicking the left button with the cursor being over the file name.



5.6 The Signal Choice Panel

The signal choice panel allows to change the set of signals and attributes displayed by a trace browser.

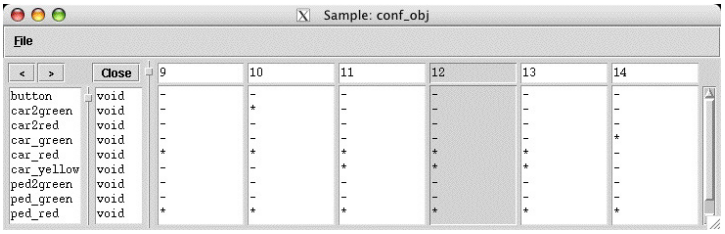


Marking a sequence of signals with the mouse on the right hand side and pushing the button << eliminates the signals and attributes. Marking on the left hand side and pushing the button >> adds starting from the position indicated by the marker >.

The marker > can be changed by clicking on the intended position in its widget.

5.7 The Value Sample Panel

displays all the values of signals in a specified region of instants. It is opened from the trace browser. The highlighting indicates the displayed instant.



The window behaves like a sliding window: by pushing the buttons < and > the displayed region moves forward or backward.

5.8 Error panel

An error panel pops up in case of some error.

6 *synERJYcharts* - The Graphic Editor

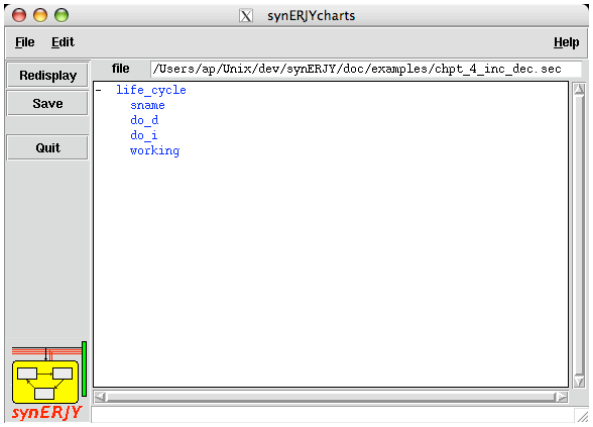
*synERJY*charts are graphical presentations of hierarchical state machine. These can be visualised and edited using the graphical editor. *synERJY*charts are stored in a binary format in files, suffixed with `.sec`.

The graphic editor for *synERJY*charts may be started

- from the *synERJY* console window,
- by typing the command `ge` in a shell (for Unix systems only), or
- by double clicking the icon of a `.sec` file (for Windows only).

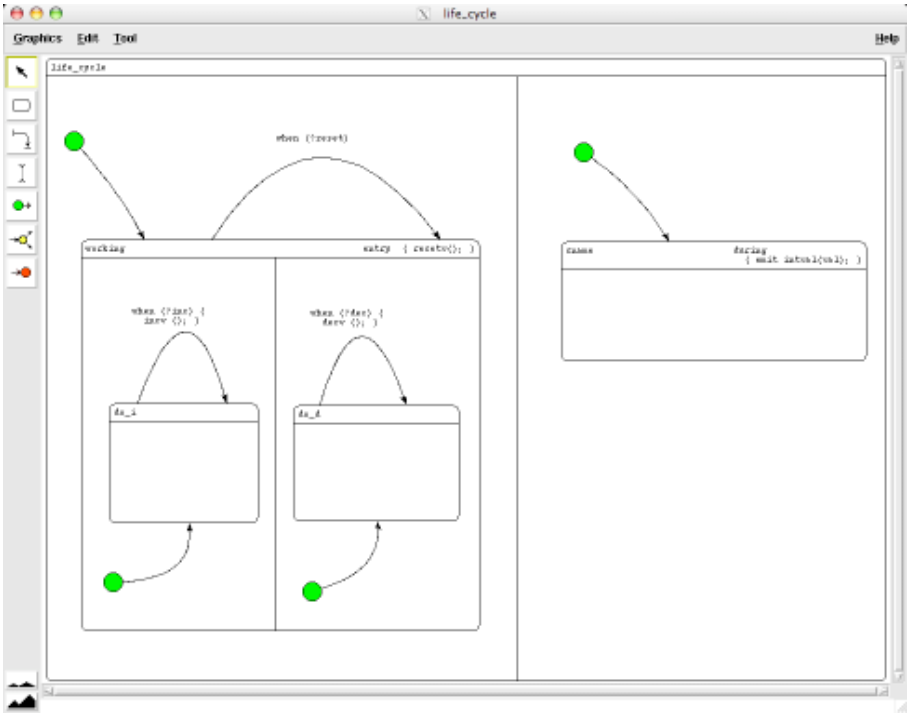
6.1 The Graphic Editor Console Window

The graphic editor console window displays the hierarchy of states if a graphics file is loaded.



6.2 The Graphic Editing Window

Editing a graphic. A graphic consists of graphic objects such as states, transitions, initial nodes and conditional nodes. For each kind, there is a specific tool to create and manipulate a graphic object. Editing takes place in a graphic edit window.



The main widget is the drawing widget. On the left there are the tool button. The entry line at the bottom is an *info window*. It displays information as well as error messages.

All graphic objects in a window have a number, the "graphic object id". Entering a number in the info window and hitting the return key, highlights the respective object. Note, that *error messages* of the *synERJY* Console may refer to graphic object id's.

Tools. The tools create and modify graphic objects on the drawing widget.

Point Tool: select, move, and resize graphic objects.

The *State Tool*: create persistent states.

Transition Tool: create transitions.

And Tool; create new and-states for a selected (super-) state.

 *Init Tool*: create initial transient states.

 *Condition Tool*: create conditional transient states.

 *Exit Tool*: creates termina transient states.

Creation. Graphic objects are created by pressing the left mouse button B1, moving, and releasing it. Moving determines the size. For transitions, the cursor must be on a state (source state) before pressing B1. Dragging the cursor to the target state and releasing B1 will create a transition. Some constraints are checked for minimal sizes of state-boxes or minimal distances of transitions to edges of state-boxes, etc. Remarks and errors are reported in the info-window.

Selection. A single or a group of graphic objects is selected either by clicking B1 on an object or by clicking B1 on the graphic window and then dragging with B1. All graphic objects contained in the rectangle spread by the dragging are selected. Adding to a selection is done by pressing the Shift-key and clicking B1.

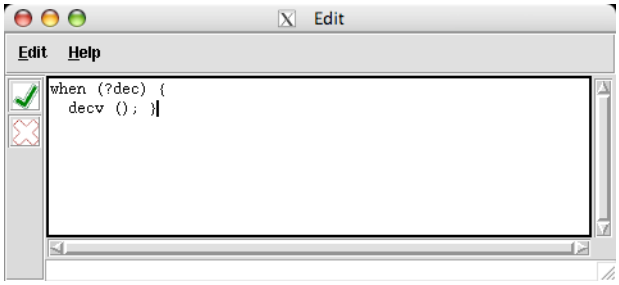
Selection of graphic objects generates handles for manipulating the object. In case of transitions, the red handles refer to the arrow of the transition, the yellow handle refers to its label.

Moving. A single or a group of graphic objects may be moved only after selecting them. B1 is pressed, moved towards the desired direction of the move and then released. Visual feedback is only given after B1 is released. Repeat if you want to improve the move.

Resizing. Resizing is possible only if a single graphic object is selected. B1 is pressed very close to one of the handles of the graphic object and released at the desired place. Visual feedback is only given after B1 is released. Repeat to improve the resize.

If a transition is resized, the source and target handle, the side of the state-box to which it is attached, the bending of the transition and the placement of the label may be modified.

Editing text. All text in the graphics can be edited. Double click B1 on the text. The *Text Edit Window* will be opened.



This simple text editor can be used for changes. Pressing the *Accept* button checks whether the text is syntactically correct according to its position. If accepted the text is copied to the graphics, otherwise an error message is displayed in the the line below the text widget. If the Text Edit Window is closed by pressing the button *Reject* the old text is kept in the graphics.

Refining a state. Any state may have sub-states (If the state has sub-states, it is called a super-state). Few sub-states including their transitions may be put into the box indicating the place of the super-state. If a super-state has many sub-states, it is recommended to put them into a separate window, the *refinement window*. Double clicking on a state opens the refinement window. This window contains the complete substructure of the super-state (often called the and-or-decomposition). The refinement may be cancelled using the `File->Undo refinement` in a refinement window. But note that all existing substates in the refined window will be lost.

Actions. There are several action which may be defined textually in a state.

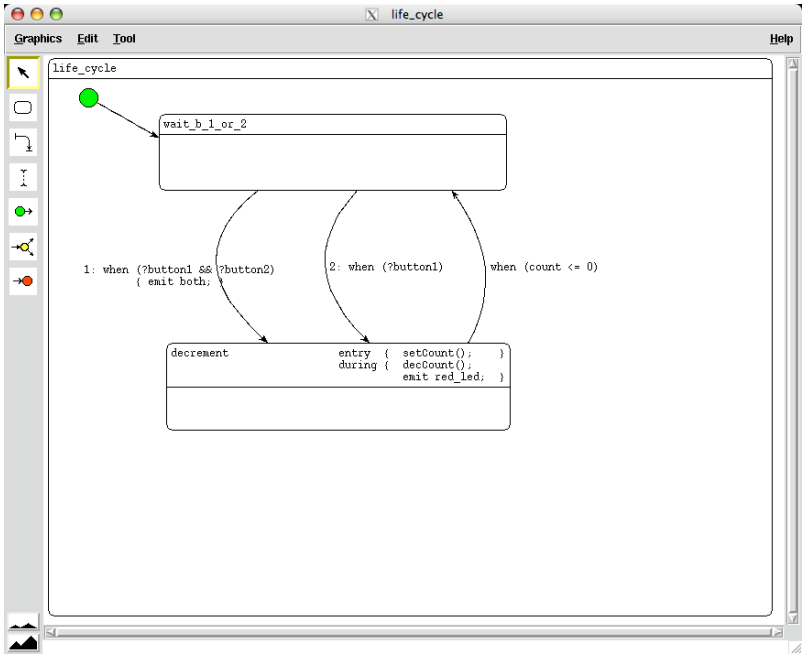
`entry` {*`action`*} - an instantaneous action executed when entering a state.

`exit` {*`action`*} - an instantaneous action executed when exiting a state.

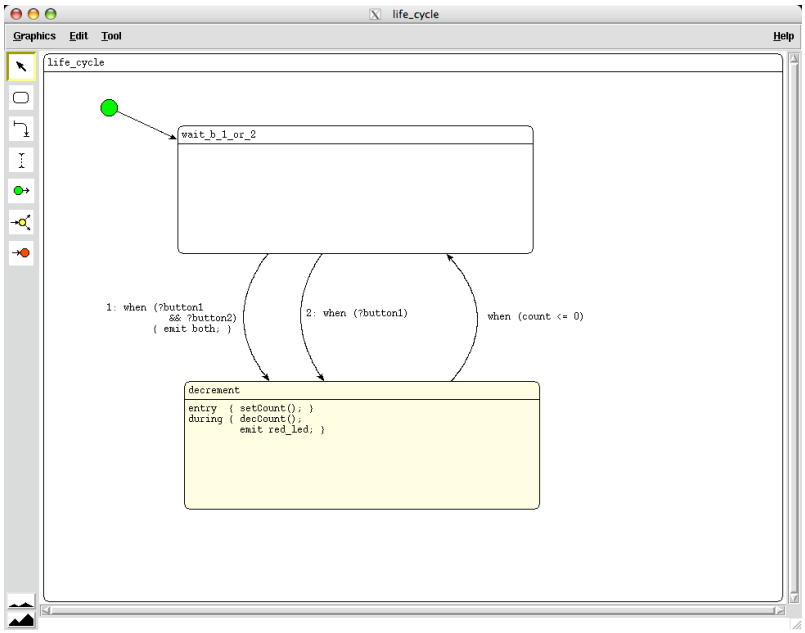
`during` {*`action`*} - an instantaneous action which is executed while being in a state.

`do` {*`action`*} - an action the execution of which starts when entering a state, and terminates when exiting a state.

The action are specified in the right upper corner of a state box, e.g.



Textual states. Sometimes the it is convenient to consider states with textual actions being exclusive. In that case the perspective may be switched in that the actions are displayed beneath the separating line, e.g.



The switch is achieved using a the context menu for state actions (see below).
Context menus. Clicking the right mouse button opens several context menus. Click on the

canvas - to switch the tool, or to paste a state

state name - to change the font size and weight

state action - to change the font size and weight, or the switch to a textual state

transition text - to change the font size and weight

For instance, there is a context menu

Known problems.

- Attaching transitions to state-boxes is not too smooth. This may sometimes be irritating in particular for small state boxes.
- Reducing the size of a state-box is rejected if the transitions attached become disconnected. The rule to detect this, however, is sometimes obscure. To overcome this problem, you may attach transitions temporarily to other states, resize the state, and re-attach the transitions.

7 Model-based Design

7.1 An Upshot of Model-based Design

Model-based design distinguishes between the environment (or plant) and the controller that is embedded into the environment. Both environment and and controller may range from being simulation software to the real hardware on the environment side and embedded controllers on the controller side.

The typical design cycle proceeds in several steps:

Model-in-the-Loop Both environment and controller are coded in a simulation tool, for instance in Matlab/Simulink [5] or Scilab/Scicos [6].

Software-in-the-Loop The controller is replaced by a binary (e.g. an Sfunction in case of Simulink, or a new block in case of Scicos).

Virtual-Silicon-in-the-Loop The controller runs on an instruction set simulator that interacts with the simulation environment.

Processor-in-the-Loop The controller runs on the target hardware that interacts with the simulation environment.

Rapid Prototyping The controller runs on a PC/workstation that interacts with the real environment.

Rapid Prototyping on Target System The controller now runs prototypical ECU.

Hardware-in-the-Loop The controller runs on a production ECU but in a HIL system.

Final Product The controller now runs production ECU.

synERJY supports Software-in-the-Loop in that it generates executables for Matlab's Simulink and Scilab's Scicos. The same code (minus instrumentation) may run on the supported hardware emulators or target hardware.

7.2 Generating Sfunctions for Simulink.

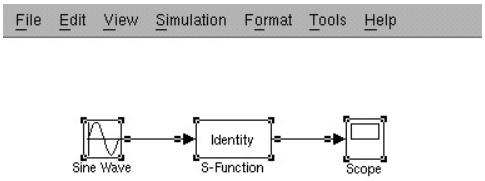
synERJY programs are integrated in Simulink diagrams using Sfunctions: for building an Sfunction

- the target must be set to Simulink in the *synERJY* console (Target↦Simulink).

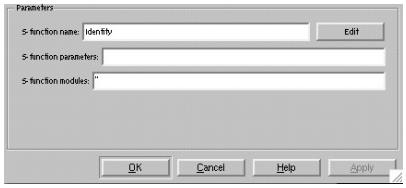
- the environment variable for the Matlab directory must be set using the *environment variable panel* (Edit↦Environment variables).

The name of the generated Sfunction is that of the configuration class.

To use this Sfunction in a Simulink diagram, insert an Sfunction block,



open its property list, and enter the name of the configuration class.



For proper operation it is recommended to start Simulink from within the respective *synERJY* project directory.

Parameters are supported. Parameters may be of all the types supported by Simulink. Type constraints may be needed: e.g. `0` will not automatically considered as being an integer, but one needs to specify its type, for instance `int32(0)` to be of *synERJY* type `int`.

The *timing constant* `timing` determines sampling time of the Sfunction block. If the timing constant equals `0sec` the sampling time of the block will be inherited by the environment.

Exceptions are raised by *synERJY* will be shown as Simulink exceptions.

Feedtroughs, i.e. signals the output of which at an instant depends on the input of some sensor at the very same instant, are detected and flagged as such, hence may give rise to "algebraic loops".

Warnings

- The method main is ignored when a SIMULINK S-Function is generated. Inspect the method body to check whether this may cause problems (often initialisation code can be moved to the constructor of the configuration class).

- On Windows, Matlab has the unpleasant property that it "blocks" the *dll*'s used, i.e. when a *dll* is used for an S-function, one cannot generate a new version without quitting *all* of Matlab. The error message related to this phenomenon are cryptic. Typically, it states that the command `-e` cannot be found.

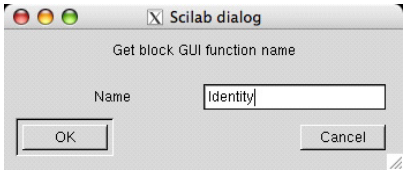
7.3 Generating Blocks for Scicos.

synERJY programs are integrated in Scicos diagrams by adding a new block to a Scicos diagram.

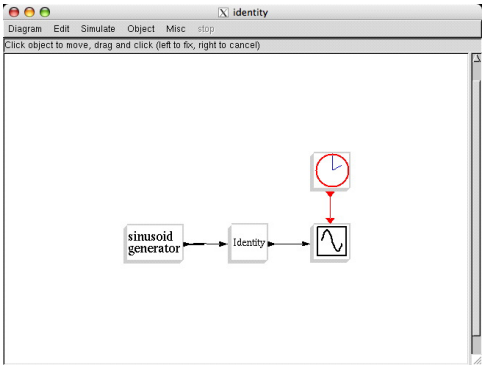
To generate all the files needed for a scicos block push the `Build` button, but only after

- the target has been set to Simulink in the *synERJY* console (Target↦Scicos), and
- the environment variable for the Scilab directory has been set using the *environment variable panel*

Then use the menu entry Edit↦Add new block of Scicos. Again the name of the configuration class should be entered.



The added block is incorporated in the graphics as in



(where the links have been set additionally).

Add the graphics to the the project (using the tab `Scicos models` in the project panel. Then push the `Run` button. Then Scilab is started executing a hidden file `.scilab` which loads the necessary libraries and the chosen Scicos model automatically.

Note that for development of the synERJY-based block Scicos diagrams should be saved as text files (using the extension `.cosf`. Then the last version generated is used for updating the diagram. If the binary version of the diagram is stored (using the extension `.cos`) the block(s) generated using *synERJY* are not updated, but the version of the *synERJY* block is kept that was available when the block as been added as a new block. This tends to a constant source of confusion when new code has been generated by *synERJY* but no effect can be seen.

On Windows, for instance double clicking on one of the Scilab files in the respective project directory would do the job. Remember that Scicos must be started within the *synERJY* project directory to load the libraries related to *synERJY*.

The support of the *timing* is rather crude in that there are two distinct cases only:

- either the timing constant is `0sec`, then the sample time of the block will be inherited.
- Otherwise one actuator input is provided. Then according to the paradigms of Scicos the sample time can be set using a clock source.

Feedtroughs are detected. However, in contrast to Simulink, the whole block is flagged to have a feedthrough, which is somewhat more restrictive than in Simulink, but is a restriction of Scicos..

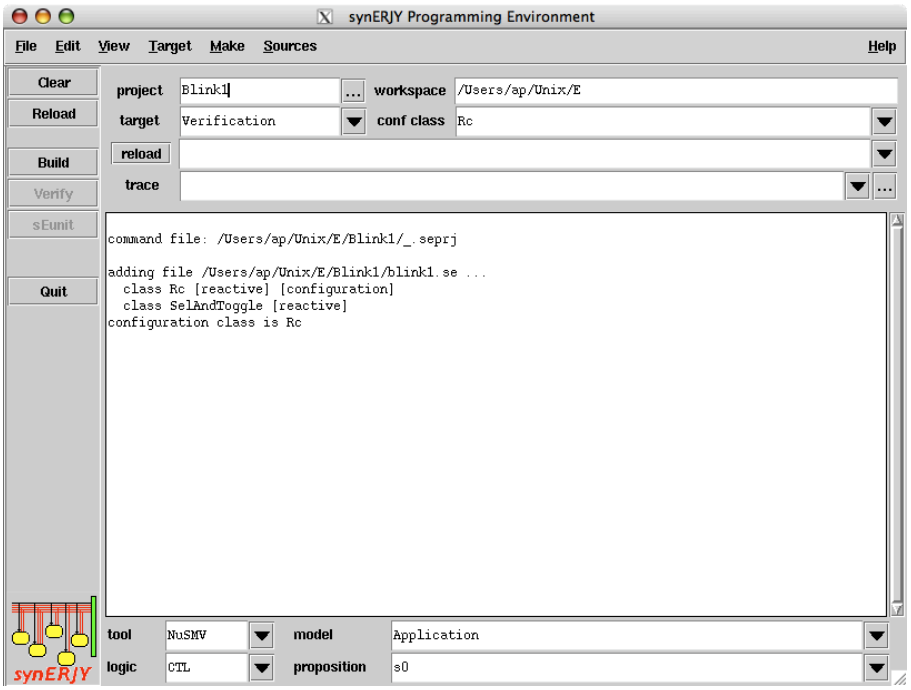
synERJY exception are supported, exception are printed using `sciprint`. They show in the Scilab window.

Parameters can used. Here the Scicos graphical interface is more convenient than the corresponding interface for Sfunctions in that each parameter has a separate entry.

Warning The method main is ignored when a Scicos block is generated. Inspect the method body to check whether this may cause problems (often initialisation code can be moved to the constructor of the configuration class).

8 The *synERJY* Verification Environment

synERJY supports verification, at present on the basis of NuSMV (`http://nusmv.irst.itc.it/`). We use the example `Blink1` (in directory `$(SE_HOME/target/examples`) in order to demonstrate the interface. When loaded choose target `Verification`, and the *synERJY*-console will change to



At the bottom you see the tool that may be used for verification (at presen t only NuSMV), the logic which may be used to specify the propositions, the scope of the verification (either the full application, or propositions with regard to a specific object), and one may choose the proposition to prove using labels. Pressing the `Build` button generates the model, and then pressing the `Verification` button starts the verification. Prerequisite is that the path to NuSMV has been defined as environment variable.

Warning: This has only been tested for a view examples on Unix systems (actually for OS X). This binding is of alpha Status.

9 How to Make *synERJY* Available for a New Micro-controller

This description shows the steps of building the runtime environment for a new micro-controller.

- ***Choose a name for the new type of micro controller***. In this description we refer to this name by `mcuname`.
- ***Describe how to install the tools necessary to build applications***.
  Typically these tools are:
  - a C-compiler and its development environment,
  - an upload tool, to install software into a target system.
- ***Create a a subdirectory-tree to*** `$SE_HOME/target`.

```
cd $SE_HOME/target
mkdir mcuname
cd mcuname
mkdir src
mkdir include
mkdir lib
mkdir lib/src
mkdir examples
```

The directories in `$SE_HOME/target/mcuname/` should hold the following files (compare, e.g., `$SE_HOME/target/AT90s8515/`):

```
src/                    *.se files to utilise the installation
                        Typically the directory holds these
                        files with basic target classes.

   Mcu.se               synERJY-classes that provide access
                        kernel of the micro controller unit.
                        These classes may
                          - define constants
                         - basic facilities like interrupt
                           enable/disable
                         - storage access
```

```
                          - addresses of registers,
                            peripheral units

   StdReader.se           serial input of a character stream
   StdBufferdReader.se    line buffered serial input
   StdWriter.se           serial output of a character stream
   StdBufferdWriter.se    line buffered serial output
   Realtime.se            access of the basic realtime clock

include/                  target specific C-header files,
                             makefile-include-files ...

lib/libsert.a             the target specific runtime library

lib/src/                  sources to build the runtime library

examples/                 synERJY classes as examples,
                             for instance to show
                             the facilities of the classes in
                             src/*.se
```

- ***Provide Basic functions and C-macros for the runtime environment***

  a) *Functions for the Timing of the Synchronous Engine.*
  Prototypes of these functions are defined in

  `$SE_HOME/include/se_realtm.h`.

  They should be implemented as a single C-module in the file

  `$SE_HOME/target/lib/src/se_realtm.c`.

  b) *Functions and Macros for Using IOstreams for Input and Output.*
  Typically these are facilities implemented by

  – `<stdio.h>` on operating systems, or
  – using asynchronous serial units (e.g. UART) of micro-controllers

  Use

  `$SE_HOME/target/mcuname/include/target.h`

  to define C-function-prototypes and -macros, that are to be included in order to access the implementation of the basic target classes. Since the interfaces are defined by *synERJY* types, the file target.h must #include ¡se_types.h¿

c) *Provide the Implementation for Each Function as Declared in* `target.h`

in a individual C-file of the directory

$SE_HOME/target/mcuname/lib/src/

Thus function is compiled into an individual C-module. Linkage will only insert function implementations into the target binary, which are actually used in the application.

- ***Provide a*** `Makefile` ***to Build the Runtime Library***

  The makefile is

  `$SE_HOME/target/mcuname/lib/Makefile`

  The first rule of this makefile must produce the runtime library The makefile is

  `$SE_HOME/target/mcuname/lib/libse.rt.a`

- ***Provide a*** `Makefile` ***for Writing Makefiles for Applications***

  The makefile is

  `$SE_HOME/target/include/Makefile.inc`

  This makefile should provide definitions to access

  – the C-compiler, linker, and other tools,
  – the runtime libraries of the C-development environment,
  – other tools like the upload program, and
  – the specific *synERJY* runtime library

  Typically default rules for compilation, linking and uploading are defined. These definitions should support writing of application specific makefiles.

- ***Give application examples, that also test the proper installation***

  There are two example applications in

  $SE_HOME/target/mcuname/examples/

  McuTiming.se

  Adapt these examples to the micro-controller and its environment. Compile and run them to check the proper installation.

References

[1] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[2] N. Halbwachs. *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, Dordrecht, 1993.

[3] D. Harel. Statecharts: A visual approach to complex systems, *Science of Computer Programming*, 8:231–274, 1987.

[4] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[5] `www.mathworks.com`

[6] `www.scilab.org`