User Manual
Building Simulator (ERLANG)

SFB 501, project C1,
Thomas Deiß (`deiss@informatik.uni-kl.de`),
Thomas Hillenbrand

$Source: /home/C1/cvsroot/Simulator/Doc/Manual/simulator.tex,v $
$Revision: 1.11 $, $Date: 1998/07/30 07:16:08 $

The building simulator is an ERLANG[1]-program which provides

- basic functionality for discrete simulation,
- functionality for discrete simulation of continuous systems,
- a set of simulation objects useful to simulate physical effects in buildings,
- different means to access the state of the simulation objects throughout simulation.

The reader is assumed to have basic knowledge of Unix. This manual does **not** provide an introduction to ERLANG [1] nor to building simulation. For an overview on the design of the building simulator cf. [2].

1 Running the Building Simulator

1.1 Starting the Building Simulator

Let `$Simulator` and `$Configurations` denote the directories in which the compiled code of the building simulator and *building configurations* reside, respectively.

To start, ERLANG has to be told where the code of the building simulator resides and which *building configuration* shall be used in simulation. Telling ERLANG where the code resides can be done either by starting ERLANG in the directory `$Simulator` or by providing `$Simulator` as a command line option:

```
erl -pa $Simulator
```

To start the building simulator with a specific *building configuration* means to call the function `simulator:start($Conf,$Dir)`. The configuration loaded in this case is the file `$Dir/$Conf.cfg`. This can be abbreviated by calling the functions `simulator:start($Conf)` to load `$Configurations/$Conf/$Conf.cfg` or `simulator:start()` to load a default configuration. `$Conf` and `$Dir` can be file, resp. directory names enclosed in double-quotes. If `$Conf` is an *atom* according to ERLANG-syntax the double-quotes can be omitted.

These functions can be called from the ERLANG-shell or by using one of the following command-line options:

[1] ERLANG is a trademark of Ericsson Telecom AB

```
-s simulator start $Dir $Conf
-s simulator start $Conf
-s simulator start
-s simulator
```

The last two forms are equivalent. In the ERLANG-shell instead of `simulator:start` the function `r` can be used alternatively.

To stop ERLANG automatically after finishing simulation one can append the command-line option `-s erlang halt` to the call of ERLANG. The complete call can have the following form:[2]

```
erl -pa $Simulator -s simulator start $Dir $Conf -s erlang halt
```

1.2 Stopping the Building Simulator

A running ERLANG-process can be interrupted by pressing `^C` and then left by pressing `a`. If the ERLANG-shell is active calling the function `halt().` stops ERLANG and the building simulator. Note the period after the parentheses!

1.3 Configuration File

When starting the building simulator a configuration file is read which contains information on where to find a description of the *simulation objects* and on the *eventstreams* to be started. Its *file extension* is `.cfg`. The entries in the file consist of valid ERLANG-terms, their syntax is described in the subsequent sections.

The files accessed by the building simulator are depicted in figure 1. Solid lines denote dataflow, whereas the dotted lines denote that which files are actually used depend on the configuration file. The *file extensions* used are also shown in this diagram.

2 Nodes

The building simulator is able to start further ERLANG-nodes which are intended to perform input/output. The simulation itself is not distributed.

Input syntax in the configuration file is

```
'[ node, ' NodeName ',' HostName '].'
```

where `NodeName` is the name of a ERLANG-node and `HostName` is the name of a host.

It must be possible that a command on `HostName` can be executed and that the local and the remote host have access to the same file system.

[2] Such a command is provided for team3 members as `startgebsim.sh` in the directory `~team3/Basis-Projekt.BaX/bin`.

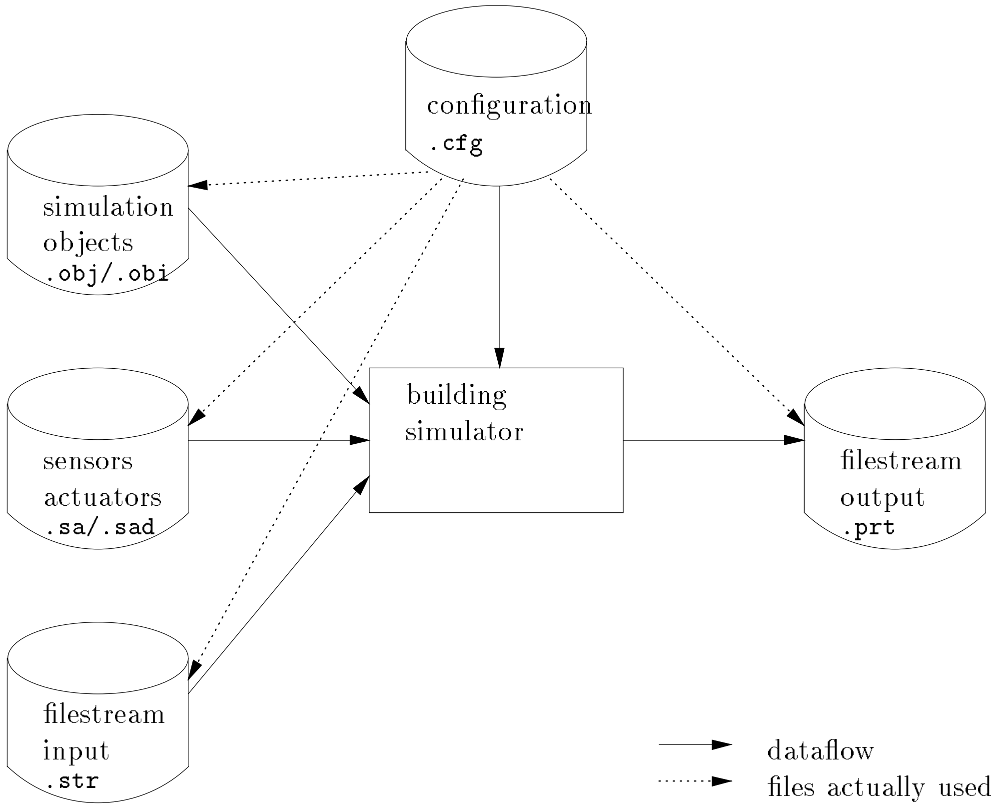

Figure 1: Files used

3 Controlling the Simulation

The building simulator communicates with its environment via so-called eventstreams. A distinguished one is used to control the simulation itself, it is called the *controlstream*. It can be used to

- start/stop/resume simulation

  A running simulation can be interrupted, either at a specified point of time, on request of a user, or because some internal problems in the building simulator occured.

- quit the building simulator,

- influence the relation between real and simulation time

  The building simulator can be run as fast as possible — *loose* — or run in a mode — *fixed* — in which it tries to maintain a relation between real and simulation time. This means that it can run n seconds simulation time passing in 1 second real-time. A slack of δ seconds simulation time is allowed before the building simulator takes responsive action. Possible reactions are to issue a warning or to stop the simulation.

- set breakpoints, i.e. specify points of simulation time where the simulation has to stop.

3.1 Entry in Configuration File

Each *configuration* must contain exactly one definition of a *controlstream*. This must be an entry of the form

```
'[ controlstream,' Window | Silent '].'

Window        'window' [',' End [',' Mode [',' Breaks [',' run ] ] ] ]

End           'infinity' | Seconds | Compound
Compound      '{' [ [ Hours ',' ] Minutes ',' ] Seconds ',' Milliseconds'}'
Hours         <non-negative integer>
Minutes       <non-negative integer>
Seconds       <non-negative integer>
Milliseconds  <non-negative integer>

Mode          'loose' | 'fixed' |
              '[' fixed [',' Cont [',' Factor [',' Slack ] ] ] ']'
Cont          'cont' | 'stop'
Factor        <positive integer>
Slack         <positive integer>

Breaks        '[]' | '[' (Seconds | Compound) (',' (Seconds | Compound))* ']'

Silent        'silent' [',' End [',' Mode ] ]
```

`End` describes when the simulation shall stop, i.e. after the given amount of simulation time. `Mode` describes whether the building simulator runs with a fixed binding between real and simulation time or as fast as possible. `Cont` describes whether the building simulator shall issue a warning or stop simulator when simulation time is more than `Slack` seconds simulation time behind. `Breaks` can be used to define breakpoints. If `Window` is used, a graphical interface is started up which can be used to change these values online and throughout an ongoing simulation: if `Silent` is used no graphical interface is started. In the last case there is no possibility to influence the running simulator. When `run` is given in the description of `Window`, the simulation starts automatically.

3.2 Graphical Interface to Controlstream

The meaning of the buttons and scales should be obvious. To enter breakpoints the focus of the window system has to be set to the corresponding input form. The required input syntax is in the configuration file is:

```
[ [ [ Days ',' ] Hours ',' ] Minutes ',' ] Seconds
```

4 Describing Simulation Objects

The *simulation objects* are described in a file together with possible connections between them. Each entry has to be a valid ERLANG-term. The detailed syntax of the description of the simulation objects and connections can be found in the corresponding header files, which are `$Simulator/def_*.hrl`.

When starting the building simulator the objects determined by the configuration file entry

```
'[ objects,' Filename '].'
```

are read. If necessary they are converted into an internal format. When converting syntactical and semantical checks are performed.

5 Accessing Simulation Objects

The state of the *simulation objects* can be controlled and monitored. It is possible to retrieve the value of each variable and each parameter of each simulation object. The value of each independent variable can be set from the environment while a simulation is running.

The communication between the building simulator and its environment are established by *eventstreams* which generate the necessary commands and convert the output of the building simulator into a user readable format.

The following kinds of eventstreams are available:

- a clock,
- file access to read and write values and commands from and respectively onto file,
- sockets, which provide ordinary Unix sockets and an abstraction of sensors and actuators.

The names of the eventstreams must be unique.

5.1 Accessing Time(s)

By the entry

```
'[ simclock,' Name '].'
```

in the configuration file a graphical display is started which displays three different times:

Simulation Day, Elapsed Simulation Time: Simulation time passed since the building simulator started. Simulation time starts on day 1, at 0:00 00.

Wall Clock Time: Time in the real world.

True Simulation Time: True time in the building simulation, this time is e.g. used to compute the position of the sun.

All times are given in the format hh:mm ss.

The different times are updated about every second, but due to the limitations of the ERLANG timing concept, there is some imprecision and the updates are not synchronous. Do not be worried that real-time binding is violated if simulation time and wall clock time do not change synchronously, the real-time binding is maintained nevertheless.

5.2 Sockets

This kind of *eventstreams* provides access to a running simulation via sockets. Access can be from other programs as well as via the command `telnet`. `telnet` can be used in either line or character mode.

Via sockets it is **not** possible to send commands to be executed at a specific time.

Sockets provide a model of sensors and actuators corresponding to the testroom of AG Zimmermann, see [3]. Several connections can be established simultaneously on each socket and several sockets can be started with different sets of sensors/actuators.

There are four different subtypes of these eventstreams:

cache: maintains a cache of values, hence responses from sensors will be faster than otherwise. Digital sensors start in eventmode.

nocache: each access to a sensor is translated to a single access of the corresponding variable. No sensor starts in eventmode.

fault: to be used with one of the previous types. Provides a simple error model of sensors and actuators. Fault classes supported are:

  ok: the sensor/actuator is ok (default)

  silent: no reaction on input via the socket takes place, nor are any values sent out.

  notavailable: reaction as accessing a non-available sensor/actuator takes place

  constant: The next value sent out or in is stored and used from then on as constant input or output.

  Sensor/actuator names are as for the previous types. The fault class can be changed by setting one of the values `ok`, `silent`, `notavailable`, and `constant` .

inspect: no mapping of sensors/actuators to variables need to be given. Instead sensor/actuator names have the form `Object'/'Variable` which can be translated trivially.

Sockets are declared in the configuration file using the following syntax:

```
'[ socket, ' Node ',' Host ',' Ports ',' Sensors '].'

Node      'localnode' | NodeName
Host      'localhost' | HostName

Ports     '[' Port ( ',' Port )* ']'
Port      '{' Name ',' Number ',' Type '}'
Name      <name>
Number    <portnumber>
Type      '[' 'cache' | 'nocache' | 'fault' | 'inspect' ']'

Sensors   '[' FileName (',' FileName)* ']'
FileName  <filename>
```

Starting sockets may take several seconds. At startup it is checked whether the tables providing the relation between sensor/actuator names and variables have to be reparsed and stored in an internal format.

An internal error can occur when several of these files are reparsed simultaneously. Restarting the building simulator until all files are parsed usually helps.

Sensors and Actuators are described as

```
Sensor     '[ sensor,' Name ', {' Object ',' Variable '}, Unit ',' Type '].'
Type       'analog' | 'digital'

Actuator   '[ actuator,' Name ', {' Object ',' Variable '},' IntRep ',' ExtRep '].'
IntRep     Unit |
           'threepoint' | 'openclose' | 'realzeroone' |
           'float' | 'time' | 'name'
ExtRep     'trigger' | 'threepoint' | 'twopoint' |
           'realzeroone' | 'percentfloat' | 'string'
```

The `Type` of a sensor determines whether it can be run in eventmode (digital) or not (analog). The `Unit` of sensors determines how values are converted before sending them on the socket. The internal and external representation information of actuators is used to convert inputs from the socket.

5.3 Files

By the entries

```
'[ file_shell ,' Name ',' Filename '].'
'[ file_file ,' Name ',' Filename ',' Filename '].'
```

in the configuration file eventstreams are started which can read commands to access the state of the simulation objects from file and and which can write output produced by the objects to the shell (`file_shell`) or onto file (`file_file`).

The input and output formats are the same.

An entry in a file has the following format, see below for periodic input:

```
's:io(' Time ',' Object ',' Variable ',' Command ').'

Time       'now' | Seconds |Compound
Compound   as for controlstream

Object     <name>
Variable   <name>

Command    '{' Value ',' Unit '}' | Number | 'get' | 'mon' | 'nom'
```

`now` is used as a special time, meaning to execute the command as soon as possible. Explicit times given are in simulation time, not in true time.

<name>s can be given as ERLANG-atoms or as strings enclosed in quotes. `Value` and `Unit` must be an appropriate value and a unit, respectively, for the variable accessed. The unit of each variable is given in the header file of the corresponding object, cf. `$Simulator/def_*.hrl`. Numerical values can be input without unit, but this is not suggested.

`get` is used to retrieve the current value of the variable once, `mon` is used to retrieve the value of the variable at each future change, i.e. to monitor the variable. `nom` is used to stop monitoring.

Setting and retrieving variables can be done periodically, too. Input syntax is:

```
's:io_period(' Time ',' Object ',' Variable ',' Command ',' Period').'

Period  'no_longer' | Seconds
```

The command is repeated with the given period, `no_longer` is used to stop further execution of a command accessing the same variable in the same way.

6 Examples

The following examples are all taken from the simulation of the same building, but the examples are neither complete descriptions of the simulation objects and sensors/actuators, nor does the filestream input match the output.

6.1 Configuration File

```
[ objects, "team3_install1" ].
[ aggregates, "team3_install1" ].

[ simclock, "SimTime" ].

%%% clouding
[ file_shell, clouding, "clouding" ].

%%% input is from testcaseNset.str
%%% monitoring commands are from testcaseNget.str
%%% monitoring is to file and shell
%%% output is to testcaseN.prt

[ file_shell, testin, "testcase7set"].
[ file_shell, testout, "testcase7get"].
[ file_file, testfile, "testcase7get", "testcase7"].

[ controlstream, window, {168,00,00,00}, [fixed, cont, 1], []  ].

%%% for controlsystem
[socket, localnode, localhost,
 [ {sock_cs, 3000, [cache] } ,{sock_fault, 2999, [fault]} ],
```

```
 ["controlsystem"]
].

%%% for a user to test controlsystem
[socket, localnode, localhost,
 [ {sock_user, 3002, [nocache] } ],
 ["usersensor", "useractuator"]
].

%%% for developers
[socket, localnode, localhost, [ {sock_inspect, 3004, [inspect] } ], [] ].
```

6.2 Simulation Objects

```
[ location_time, -0.1356, 0.8631, -0.2618, 3600, 36000 ].
%%%         initial day of year (jun 21)
[ calendar, 172 ].
[ sun_position, 80, 0.4090 ].

[ weather, 15 ].
[ outdoor_brightness, 120000 ].

[ kernel, kernel ].

[ occupied_area, playground ].
[ person, person_0, playground ].
[ person, person_1, playground ].

[ occupied_area, occ_staircase_east ].
[ supervised_area, sup_staircase_east, occ_staircase_east ].
```

6.3 Sensors/Actuators

```
[ sensor, localtime, {location_time, local_time}, 's[d]', analog ].
[ sensor, truetime,  {location_time, true_time}, 's[d]', analog ].
[ actuator, cloudiness, {weather, cloudiness}, float, percentfloat ].
[ sensor, kernelfactor, {kernel, factor}, intnonneg, analog ].
[ actuator, person_0_loc, {person_0, location}, name, string ].
[ actuator, person_0_acc, {person_0, active}, 'active//inactive', twopoint ].
```

6.4 Filestream Input

Note that the name of an occupied_area given as new value must be a string, not an ERLANG-atom.

```
s:io( now, light_32_416_wall, performance_rate, get).
s:io( now, light_32_416_wall, performance_rate, mon).
```

```
s:io( now, light_32_416_window, performance_rate, get).
s:io( now, light_32_416_window, performance_rate, mon).

s:io( 240, person_4, location, {"occ_staircase_east", object}).
s:io( 241, door_32_4_1, aperture_state, {opened, 'opened//closed'}).
s:io( 242, person_4, location, {"occ_hallway_H1_1", object}).
s:io( 243, person_4, location, {"occ_hallway_H1_2", object}).
s:io( 244, person_4, location, {"occ_hallway_H1_3", object}).

s:io_period( 300, person_6, location, {"occ_room_32_429", object}, 610).

s:io_period( 360, person_6, location, {"occ_hallway_H1_4", object}, 610).

s:io_period( 2630, person_6, location, {"occ_room_32_429", object}, no_longer).
s:io_period( 2630, person_6, location, {"occ_hallway_H1_4", object}, no_longer).
```

6.5 Filestream Output

```
io({0,5,20,0}, la_32_416_wall, current_brightness, {173.878,lux}).
io({0,6,0,0}, light_32_416_window, performance_rate, {50.0000,'%'}).
io({0,6,0,0}, la_32_416_desk, current_brightness, {220.000,lux}).
io({0,6,0,0}, la_32_416_window, current_brightness, {231.837,lux}).
```

7 Glossary

building configuration: A building configuration is a file describing which *simulation objects* are to be loaded and which *eventstreams* are started.

controlstream: Special *eventstream* used to monitor and control the building simulator itself.

eventstream: Eventstreams are used by the building simulator to communicate with its environment. Eventstreams are used to influence and to monitor the state of *simulation objects*. The *controlstream* is a special eventstream used to control and to monitor the building simulator itself.

file extension: The following file extensions are used:

configuration	.cfg
object description	.obj
object description (compiled)	.obi
aggregate description	.agr
aggregate description (compiled)	.agi
sensor/actuator description	.sa
sensor/actuator description (compiled)	.sad
filestream (input)	.str
filestream (output)	.prt

The aggregate descriptions are obsolete in the current version and are retained only for backwards compatibility.

All files containing compiled descriptions must be writable by the user of the building simulator.

simulation object: Simulation objects encapsulate the actual simulation. They can be updated at specific times and they can propagate changes of variables to other simulation objects. They can be influenced via the *eventstreams* by sending new values and they can make the values of their variables visible via the *eventstreams* either on demand or on change of the values.

References

[1] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 2nd edition, 1996.

[2] Thomas Deiß and Thomas Hillenbrand. A case study on the use of SDL. SFB 501 Bericht 03/1997, Sonderforschungsbereich 501, Fachbereich Informatik, Universität Kaiserslautern, 1997.

[3] Andreas Metzger, Stefan Queins, and Bernd Schürmann. Installation eines Testraums zur Gebäudeautomatisierung und deren Schnittstelle. SFB 501 Bericht 13/96, Sonderforschungsbereich 501, Fachbereich Informatik, Universität Kaiserslautern, 1996. In German.