Koala

Autonomous Diagnosis and Reconfiguration Tool
Documentation for Koala version 0.5
April 2003

Emmanuel Benazera

1 Introduction

1.1 Overview

KOALA is a model-based diagnosis engine that merges up-to-date modeling techniques based on both hybrid systems and qualitative physics, with powerful on-line diagnosis algorithms. The aim of this software has been to bring an integrated solution to the on-board diagnosis of complex systems. Originally oriented toward the research and development of a supervisor dedicated to autonomous satellites, it is able to model a large category of other devices. From the model, KOALA generates a C++ runtime able to run on the dedicated device. It uses commands and available observations to monitor the physical system behavior, tracking multiple trajectories if necessary, detecting faults, and diagnosing them on-the-fly. KOALA tracks the system even when in its faulty state.

1.2 Installing the software

1.2.1 Requirements

To compile KOALA you at least need the following programs:

- a version of `bison` and `flex` or compatible programs. It is known to work fine with respective versions `1.28` and `2.5.4`.
- a version of `gcc` such as `2.95.3` (`3.x` is ok).
- the GNU version of `make`, also known as `gmake` or `gnumake`.
- the `readline` library for the user interface.

and optionally:

- `gnuplot` or any other plotting utility to watch the results.
- a version of `dot` from the `graphviz` package.

  `http://www.research.att.com/sw/tools/graphviz/`. This is used for automatic generation of causal graphs in postscript format.
- a version of `gv` (ghostscript viewer) to view generated causal graphs.
- a version of the `bdd` package named BUDDY: `http://www.itu.dk/research/buddy/`.
- C/C++ code has comments such that a complete documentation can be automatically generated using `doxygen`: `http://www.doxygen.org/`

1.2.1.1 Compilation

Run the '`configure`' script at the root of KOALA's directory that will detect most of the required programs and create '`config.h`' and all '`Makefile`'. Then run the GNU version of `make`. When it's done, install in the KOALA's directory (by default not your machine local directory, and we recommand not to do this) with `make install`.

1.2.1.2 Setting up the repository

Some of the code generation modules in KOALA need to know about the koala repository (for linking to libraries). You must set up the shell variable `KOALADIR` so it points to your KOALA root directory:

```
csh type:
setenv KOALADIR /home/ebenazer/progs/koala
bash type:
export KOALADIR=/home/ebenazer/progs/koala
```

1.3 What you get

KOALA lets you design models of complex physical systems, whose descriptions can use both qualitative and quantitative constraints. A physical system has each of its components modeled before being assembled, in the same manner as in any object-oriented language. For this reason a large part of KOALA has been developed in C++ as the underlying formalism makes use of standard classes and objects. After the modeling is completed, KOALA's *compiler* compiles data, generates C++ code for each component as well as for the *runtime* process. Everything is then compiled into binaries, ready to run as a monitoring system. So what you get is in the following directories, from the KOALA root directory:

'`behav`' C++ classes for discrete components behavior description.

'`bin`' binaries directory with subdirectories for every architecture KOALA has been compiled for.

'`caen`' C/C++ code for continuous behavior description and model compilation.

'`compiler`'
C++ parsers, for variables, definition files and model files.

'`doc`' Code and user documentation.

'`exec`' C++ code for the *compiler* and the *runtime* user (basic) interface.

'`include`' *runtime* necessary headers.

'`itms`' C++ code for the Incremental Truth Maintenance System[1].

'`lib`' libraries for KOALA with subdirectories for your machines architectures.

'`model`' A few examples of models: a thermostat, satellite propulsion subsystem, valve driver, etc ...[2]

'`system`' C++ classes for scheduler and *runtime*.

'`tools`' some useful tools for modeling, mostly scripts.

---

[1] ref ?

[2] '`koala_models-<version>.tar.gz`' is a different package from the koala main sources package. See `http://www.laas.fr/~ebenazer/dloads.html`.

2 Langage Description

Reserved words:

`uses` importation definition files

`component` component declaration

`cin` command input declaration

`cout` command output declaration

`others` components importation

`cmdin` command input to a component

`shared` shared command

`vardef` variables declaration

`transitions` transitions declaration

2.1 Modeling a component

KOALA models a physical system as a set of concurrent components. Each component is described by a hybrid automaton and runs concurrently to the others, communicating through shared variables. This method allows the modeling of very large physical systems by making KOALA focuse on the commanded components. A component's model can combine *qualitative* and *quantitative* behavior description.

2.1.1 The definition file

A definition file '`.def`' describes a unique component type. The name of the file MUST be the name of the type of the component described (as classes in Java). The definition file is divided into five ordered parts: *importations*, *component*, *variables*, *constraints*, *transitions*. The main structure of the file is as follows:

```
Component "defname" { ... }
vardef { ... }
constraints { ... }
transitions { ... }
```

2.1.2 Components inclusions

Similarly to common programming languages, it is possible to *include* definition files into other components' models. Such components types are included for use with `uses` followed by the list of included types:

```
uses vdu, valve;
```

where the types must correspond to existing definition files '`vdu.def`' and '`valve.def`'.

2.1.3 Component declaration

This section declares the component, with its definition name (`defname`, equivalent to C++ class name), and if necessary, its inputs, outputs and imported types.

```
Component "defname" {
cin: <inputs>;
cout: <outputs>;
others: <type1> var1, ..., <typeN> varN;
}
```

The arguments to `others` define the *constructor* of the component: when instanciated (see Section 2.2 [Modeling a plant], page 12) the arguments have to be passed in the same order as they have been declared, because their respective types are verified.

2.1.3.1 Inputs/Outputs declaration and importation

The syntax for input and output commands is slightly more complex than the components importation (see Section 2.1.3.2 [Importing components], page 5) as the commands require a special treatment. A distinction is made between command variables that are defined inside the current component file, and the importation of a command from another component. For practical reasons, it is not necessary to import a component to use its command output.

A drawback to this approach is that the type and existence of command values are not checked when imported and this might lead to the use of non-existent commands in the model. Defining a command variable:

```
CommandVar <var_name> (default, val1, ..., valn)
```

where `(default, val1, ..., valn)` are the discrete values the command variable ranges over. The first argument (here `default`) is the value the variable is initialized with.

Importing a command variable is done by using the reserved word `shared`:

```
shared (default, val1, ..., valn) c-
```

`c-` keyword tells the model compiler that the imported command has no other values permitted than `(val1, ..., valn)`. This is for safety at runtime, and at this point it is no need turning this option off.

When importing commands, it is possible to have them incorporate the component's name (the component's name is defined when it is instanciated, see Section 2.2 [Modeling a plant], page 12) easily:

```
Component "c";
...
shared (default, %val1, ..., %valn) c-
```

where `%` will be replaced by the compiler with the component name. For example, with `comp1` the name of a component, the value for the shared command would be `comp1_val1`. We recommand you refer to the component declaration example (see Section 2.1.3.3 [Component declaration example], page 5) while writing your own model.

2.1.3.2 Importing components

Importing other components:

```
uses valve, vdu;
...
others: valve v1, valve v2, vdu vd;
```

The arguments to `others` define the *constructor* of the component: when instanciated (see Section 2.2 [Modeling a plant], page 12) the arguments have to be passed in the same order as they have been declared, because they have types. Also make sure component types are included in your current definition file (`uses` keyword).

2.1.3.3 Component declaration example

The following example declares two components in respective files '`valve_cont.def`', '`valve.def`', where '`valve_cont`' is a valve controller, and '`valve`' is the mechanical valve. The declaration for the controller:

```
Component "valve_cont";
{
cin: CommandVar cmdin (none, v_open, v_close);
cout: CommandVar cmdout (none, v_open, v_close);
}
```

and for the valve:

```
uses tank1, tank2;

Component "valve";
{
cin: shared (none, %open, %close) c-;
cout:
others: tank1 t1, tank2 t2;
}
```

In this example, the valve controller possibly takes three commands as inputs (`cmdin`), and may produce them in output (`cmdout`). The valve shares the controller's output. It may seem strange that the controller's commands declaration bears the name of the valve, `v`, but this has prooved useful in practice. Finally note that the valve imports the definition for two different models of tanks.

2.1.4 Variables

Variables are declared in the `vardef { }` section of the definition file. There are different *types* of discrete variables, `ModeVar`, `ObsVar` and `DependentVar`, and of continuous variables, `input`, `measure` and `parameter`.

2.1.4.1 Discrete variables

Discrete variables define the *discrete state* of a component, and are declared as:

```
<type> var_name (default, val2, ..., valn, unknown);
```

A component needs at least one variable of type `ModeVar`, whose values model the functional *modes* of the component's automaton.

Note that the first value of the set will be the default value of the variable at initialization (for more information on how KOALA helps building the initialization state, see Section 4.2.3.1 [init], page 17). Moreover, to be ready for diagnosis, a `ModeVar` variable must have one of its values equal to `unknown`, otherwise the compiler will issue a warning.

`ObsVar` is the type of discrete observable variables. `DependentVar` is the type for all other variables except modes.

2.1.4.2 Continuous variables

Continous variables define the *continuous state* of a component and are declared with the keyword `variable` or `parameter` if they are constant:

```
variable var1, ..., varn;
input var1, var6;
measure var2, var3;
display var5;
parameter p1=3, p2 in [0.3, 0.5], ..., pn=0.1;
```

Inputs (`input`) variables are continuous commands to the system, and measures (`measure`) are available observations. Declared parameters (it is also possible to use values in the model without declaring any parameter for them) support uncertainty as they can be defined as reals or intervals. Intervals are declared as follows:

```
p in [pmin, pmax]
```

Infinite values are accepted and written `+oo, -oo`. Continuous variables you may want to display are declared with `display`. Note that the unique output file will be in directory '`disp`' (see Section 2.3 [Files], page 12).

2.1.4.3 Conditional variables

Conditional variables are specific variables that link the continuous state of the hybrid model to its discrete state. They are declared as other discrete variables but their type is `CondVar`, and their values are inequalities.

```
CondVar var_name ((var <|<=|>|>= expr1), ..., (var <|<=|>|>= exprN));
```

where `var` is any continuous variable (see Section 2.1.4.2 [Continuous variables], page 6). `expr1, ..., exprN` are *expressions*, i.e. any function of other continuous variables (but not `var`). Expressions may be parameters, or comprehend addition (`+`), substraction (`-`), division (`/`), product (`*`) and exponent (`^`). Parenthesis can be used to separate subparts of an expression.

The purpose of conditional variable is to partition the continuous space according to the *conditions* that are present in the model. For example transitions (see Section 2.1.8 [Transitions declaration], page 9) have guards that may be conditions over the continuous state. You should have no problem expressing any condition among variables in a form such that only a single variable appears in the left member of the inequality.

2.1.5 Constraints declaration

There are two types of constraints in KOALA to describe a component's behavior: *qualitative* and *quantitative* constraints. Qualitative constraints are expressed in propositional logic over the discrete variables. Quantitative constraints express the continuous differential relations between continuous variables. Constraints are defined in the `constraints { }` area of the definition file.

2.1.6 Qualitative constraints

2.1.7 Continuous constraints

Continuous constraints may be linear static relations among continuous variables or linear differential equations[1]. The equations are not entered as is in the '`.def`' file. KOALA uses a causal-graph of influences among variables instead. An influence is a causal relation between two variables `var1` and `var2`, and has an underlying discrete-time operational equation, according to a sampling period T_s . The operational equation is built from the static relations and transfer functions of differential equations (see *ref* for more details about the overall process, and see Section 2.1.7.6 [Continuous constraints examples], page 9 before you write your own model). There are different types of built-in influences. In the following, `N` is a number and all parameters (such as `gain`, ...) can take intervals as their values.

2.1.7.1 Constant influence

```
iN: c1 -C-> var1: gain=p1;
```

declares a *constant* influence, which equation is of the form $var1(k+1) = p1 * c1$,

2.1.7.2 Static influence

```
iN: var1 -S-> var2: gain=p1, delay=p2;
```

declares a *static* influence expresses a continuous relation of the form

$$var2(t) = p1 * var1(t - p2)$$

[1] The limitation to a linear model will be soon overcome in the forthcoming versions of the software.

thus its operational discrete-time equation is

$$var2(k+1) = p1 * var1(k+1-d)$$

where d is the smallest integer greater or equal to $\frac{p2}{T_s}$.

2.1.7.3 First-order dynamic influence

```
iN: var1 -D-> var2: gain=p1, delay=p2, resptime=p3;
```

declares a *first-order dynamic* influence where `resptime` is the *response time* of the system. The continuous equation that links both variables is of the form

$$a_1 * \dot{var}2(t) + a_0 * var2(t) = b_0 * var1(t - p2)$$

The operational equation is thus of the form

$$var2(k+1) = \alpha * var2(k) + p1 * (1-\alpha) * var1(k-d)$$

where $p1 = \frac{b_0}{a_0}$ must be evaluated properly by the user, and $\alpha = exp(-3\frac{T_s}{p3})$.

```
iN: var1 -I-> var2: gain=p1, delay=p2;
```

is an *integral* influence,

2.1.7.4 Second-order dynamic influence

There are three ways of declaring a *second-order dynamic* influence.

```
iN: var1 -D2-> var2: gain=p1, xi=p2, w=p3, delay=p7;
```

and

```
iN: var1 -D2-> var2: gain=p1, n1=p2, n2=p3, a0=p4, a1=p5, a2=p6, delay=p7;
```

are equivalent. The continuous equation is of the form

$$a2 * \ddot{var}2(t) + a1 * \dot{var}2(t) + a0 * var2(t) = n1 * \dot{var}1(t - p7) + n0 * var1(t - p7)$$

`xi` and `w` are the respective *damping ratio* and the *undamped natural frequency* of the second-order system. Otherwise, the general form of the transfer function attached to the continuous equation is of the form

$$\frac{var2(s)}{var1(s)} = \frac{n0 + n1 * s}{a2 * s^2 + a1 * s + a0}$$

Finally, the special case of having $a1 = a0 = n0 = 0$ corresponds to the transfer function

$$\frac{var2(s)}{var1(s)} = \frac{K}{s^2}$$

and is declared with as follows

```
iN: var1 -D2-> var2: gain=p1, xi=+oo, delay=p7;
```

2.1.7.5 Continuous variables initialization

```
init var1=val0, ..., varn=valn0;
```

where `vali` is any number.

2.1.7.6 Continuous constraints examples

Any dynamic influence (first or second order) is first materialized by an intermediate variable, which stands for its associated *marginal influence* (this step is not necessary for *static* and *constant* influences). The combination is then performed by adding up all the marginal influences by means of *static* influences.

2.1.8 Transitions declaration

Transitions model changes among `ModeVar` variables. Transitions are declared in the `transitions { ...}` area of the definition file. Basically a transition is made of *premisses*, conjunction of statements of the form `var="val"`, where at least one variable is a `ModeVar` variable. The conjunction of other variables is said to be the *guard* of the transition. The last element of a transition is the *post effect*, that is a conjunction of statements of the form `var="val"`. Additionnaly, a *mapping function* may be defined (see Section 2.1.8.2 [Mapping functions], page 9). When its premisses are fulfilled in an hybrid state, a transition is said to be *enabled*.

2.1.8.1 Transitions

`N` being an integer number, three transitions may be declared.

Nominal transition (nominal behavior):

```
tN: {Nominal (var1="vali") & ... & (varn="valj")
                Next (var1="valk") & ... & (varq="vall") p;
        <mapping function>}
```

Idle transition:

```
Idle (var1="vali") & ... & (varn="valj") Next (var1="valk") p;
```

Fault transition (transition to a fault mode):

```
tN: {Fail (var1="vali") & ... & (varn="valj")
                Next (var1="valk") & ... & (varq="vall") p;
        <mapping function>}
```

`p` is a real number, written as a decimal `x.x`, that is the probability of occurence attached to the transition, when *enabled*.

2.1.8.2 Mapping functions

The mapping function is attached to a transition transfers the hybrid state of an automaton from a mode to another. It inializes the declared values of some variables in the new mode, and lets other variable values unchanged. It is declared inside a `Nominal` or `Fail` transition as a sucession of conditioned variables assignments:

```
cond1: var1=vali, ..., varp=expr;
...
condn: var1=vali, ..., varp=expr;
```

where `expr` can be any mathematical expression, including standard operations on any variable, parameter or number (see Section 2.1.4.3 [Conditional variables], page 6). `condj` is empty or any logical conjunction or disjunction over discrete variable instances (see Section 2.1.6 [Qualitative constraints], page 7).

2.1.9 Definition files examples

The following example is a thermostat system that heats a room. Both the room and the thermostat are modeled by two hybrid automatons of respective modes `open`, `closed`, `unknown` and `on`, `off`, `stuck_on`, `stuck_off`, `unknown`. The room temperature $R.x$ is influenced by the temperature of a source $R.xdq$ according to a first-order differential equation which accounts for the room characteristics $R.Q\_c$ (closed) and $R.Qo$ (open). The actions that move the room from one mode to another are modeled as observed single discrete commands $R.cmd = open$ and $R.cmd = close$. The thermostat switches according to the room temperature $R.x$ (it should be in its *on* mode when the temperature $R.x <= T.m$ to warm up the room, and back to its *off* mode when $R.x >= T.M$ to cool it down). $R.x$ is hence influenced by the heater setting temperature $T.h$ (in mode *on*) or by the outside temperature $R.xext$ (in mode *off*). The temperature variation $T.xd$ is observed through a sensor with additive normally distributed noise $xdnoi$ and internally integrated to $R.x$ with an uncertain factor $T.a$. Initially, $R.x = R.xext$, the room is *closed* and the thermostat is *on*.

```
uses room;

Component "therm" {
cin:
cout:
others: room r;
}

vardef {
ModeVar mode(off, on, stuck_on, stuck_off, unknown);
parameter h=20, M=17, m=10;
}

constraints {
i1: (mode="on") | (mode="stuck_on") h -C-> r.xdq;
}

transitions {
t1 : { Nominal (mode="off") & (r.c1="c10") Next (mode="on") 0.99;
r.x=m;}
t2 : { Nominal (mode="on") & (r.c1="c12") Next (mode="off") 0.99;
r.x=M;}
Idle (mode="on") & (r.c1!="c12") Next (mode="on") 0.999;
Idle (mode="off") & (r.c1!="c10") Next (mode="off") 0.999;
```

```
Idle (mode="unknown") Next (mode="unknown") 1.0;
Idle (mode="stuck_off") Next (mode="stuck_off") 1.0;
Idle (mode="stuck_on") Next (mode="stuck_on") 1.0;
tf1: {Fail (mode="off") & (r.c1="c10") Next (mode="stuck_off") 0.01;}
tf2: {Fail (mode="on") & (r.c1="c12") Next (mode="stuck_on") 0.01;}
tf3: {Fail (mode="off") Next (mode="unknown") 0.001;}
tf4: {Fail (mode="on") Next (mode="unknown") 0.001;}
}

uses therm;

Component "room" {
cin: CommandVar cmdin (none, open, close);
cout:
others: therm t;
}

vardef {
ModeVar mode (closed, open, unknown);
variable xd, x, xdnoi, xdq;
parameter xext=4, Qc in [0.01,0.05], Qo=0.0002,
        m=10, M=17, un=1, a in [0.95, 1.05];
CondVar c1 ((x<m), (x>=m & x<M), (x>=M));
input xdnoi;
measure xd;
display xd, x;
}

constraints {
i1: (t.mode="off") | (t.mode="stuck_off") xext -C-> xdq;
i2: (mode="open") | (mode="closed") xd -I-> x: gain=a, delay=0;
i3: (mode="open") | (mode="closed") xdnoi -S-> xd: gain=1, delay=0;
i4: (mode="closed") xdq -S-> xd: gain=Qc, delay=0;
i5: (mode="open") xdq -S-> xd: gain=Qo, delay=0;
i6: (mode="open") | (mode="closed") x -S-> xdq: gain=-1, delay=1;

init x=4, xd=0;
}

transitions {
t1: {Nominal (mode="closed") & (cmdin="open") Next (mode="open") 0.99;}
t2: {Nominal (mode="open") & (cmdin="close") Next (mode="closed") 0.99;}
Idle (mode="closed") Next (mode="closed") 0.99;
Idle (mode="open") Next (mode="open") 0.99;
Idle (mode="unknown") Next (mode="unknown") 1.0;
tf1: {Fail (mode="open") Next (mode="unknown") 0.01;}
tf2: {Fail (mode="closed") Next (mode="unknown") 0.01;}
}
```

Note that cross-using the components poses no problem.

2.2 Modeling a plant

Once each component has its generic description in the form of '`.def`' files, the model is assembled in a '`.mdl`' file by including all necessary component types and instanciating them as needed. The '`.mdl`' file has three parts: *component types inclusion*, *components declaration* and *components instanciation.*

Component types are included in the same manner as components in Section 2.1.2 [Components inclusions], page 3 with the `uses` keyword.

Each component is declared as follows:

```
<type1> var1;
<type1> var2;
...
<typek> varn;
```

Beware, `<>` braces are required ! Components variables `vari` are then instanciated according to their respective models ('`.def`' file).

```
vari ("name", cmd_in, cmd_out, others1, ..., othersn);
```

where `others1, ..., othersn` are the component variables in the same order as called in the '`vari.def`' file. `cmd_in, cmd_out` are link to other component variables commands input and output. They must be called as attribute of components: `varj.cmdout` for example.

2.2.1 Model example

Here is the '`.mdl`' file for the thermostat system.

```
uses therm, room;

<room> R1;
<therm> T1;

T1 ("thermostat", 0, 0, R1);
R1 ("room", 0, 0, T1);
```

2.3 Files

Models in KOALA uses special directories and files must be respected. To create these directories, you must execute the macro '`bin/koalamkdirs`'.

Reserved directories:

'`disp`' results from running a runtime. Ready for display.

'`fig`'

'`inp`' repository of input files

'`tmp`' compilation garbage

'`C`' model generated C++ code

'`bin`'

‘obs’ repository of observations files

‘runtime’ supervisor runtime generated C++ code

‘stats’

3 The Compiler

Basically, KOALA is a compiler. It takes a model (‘`.mdl`’ file) in entry and builds a C++ runtime that is the model-based supervisor to run on the physical system. The process is as follows: a model is *loaded*, *compiled*, and *executed*. Once the runtime has been executed, more commands are available (see Chapter 4 [The Runtime], page 17).

3.1 Model Loading

Here is how KOALA is invoked:

```
koala (-hyb,disc) (appname.mdl)
```

The `-hyb` option is the default option that indicates KOALA it is going to deal with hybrid component representations (i.e. there are models with continuous constraints). The `-disc` option indicates KOALA it is going to deal with discrete representations only (i.e. qualitative constraints only). The ‘`appname.mdl`’ is an optional model file argument for KOALA to load it. When starting KOALA without any argument, a hybrid model can later be loaded with command `load` (see Section 3.2.1.1 [unload], page 14).

3.2 Compiler’s user interface commands

The compiler’s interface offers the basic features of a shell such as the history of commands. `help` brings on the command use.

3.2.1 Model operations

3.2.1.1 load, reload, unload

Purpose: The command `load` loads a system model into the COMPILER environment with syntactic and semantic errors checking

Syntax: load | reload <appname.mdl>

The `reload` relaunch KOALA and thus lose the history of commands.

3.2.1.2 compile

Purpose: The command `compile` compiles a *loaded* model by performing numerous operations on data and generating C++ operational code to be used by the runtime.

Syntax: compile (<frequency>)

If the frequency is not set, its default value is 1.

3.2.1.3 execute

Purpose: The command `execute` executes a *compiled* model’s runtime and enter the execution menu (see Section 4.2 [Runtime commands], page 17).

Syntax: execute <inputfile> <obsfile> <outfile> <scenfile>

For more information about parameters of `execute`, see Section 2.3 [Files], page 12.

3.2.2 Miscellanenous commands

3.2.2.1 quit

3.2.2.2 help

3.2.2.3 system

3.2.2.4 command-line

3.2.3 Printing data

3.2.3.1 print

Data structures of a loaded model may be printed.

`dic` print continuous variables

`inf` print influences

`cond` conditions defined in discrete variables (CondVar)

`co` list of components in the model

`co + name component variables, constraints and transitions`

`toms` all boolean propositions in the model (compiled model only)

`gra` causal graph (compiled model only)

`upd` variables update sequence

`glo` global constraints

`cdc` conditions as constraints

`init` initializations

3.2.3.2 graph

Purpose: Generates a visual postscript representation of the causal graph of influences (cg), or the discrete automaton of a component (comp_name).

Syntax: graph cg | <comp_name>

3.3 Example

3.4 Options

3.4.1 hybrid, discrete or continuous

3.4.2 log

3.4.3 envelopes optimization

3.4.4 generated code optimization

3.4.5 diagnosis of exogeneous commands

3.4.6 continuous and discrete time windows

3.4.7 prime implicants compilation

4 The Runtime

4.1 Inputs, Commands and Observations

4.2 Runtime's user interface commands

4.2.1 Printing data

4.2.2 plotting the results

4.2.3 Runtime operations

4.2.3.1 init

4.2.3.2 simul

4.2.3.3 detect

4.3 Reading from sensors

4.4 Other tools

Concept Index

C

D

E

I

K

L

M

O

Q

S

T

V

Short Contents

Table of Contents