CLS^2
Manual

December 11, 2012

Contents

1 Installation

1.1 Requirements

The CLS^2 project is based on the cmake build package [1].

packages required in ubuntu :
sudo apt-get install cmake

This should make compilation comfortable and platform independent. As cmake is very popular, most development environments will accept the project description of CLS^2 . In this way the code can be compiled and edited using comfortable and GUI-based environments. Examples are "Eclipse" [2], "qtcreator" [4] and "kdevelop" [3]. For the installation and compilation of the package with one of these environments please see the documentation of the respective tool.

The core code of CLS^2 is programmed in C++ under the ansii standard. Some tools require X11-development libraries or qt4 development libraries.

1.2 General installation procedure

The required steps for a manual (command line) out of source build:

```
1 cd CLSquare
2 mkdir build
3 cd build
4 cmake ..
5 make
6 make install
```

This procedure will install the binaries in the main `CLSquare` directory under the `bin` directory. When a file's content has changed, start at step 5. If files were added in some existing subdirectory, normally starting at step 4 will be sufficient.

To configure the package a number of options can be provided to the cmake command. These options are defined in the form `cmake .. -D<OPTION>=<VALUE>`. Some available options:

CMAKE_INSTALL_PREFIX Will control where the binaries will be installed (default is `CLSquare` main directory).
Example: `cmake .. -DCMAKE_INSTALL_PREFIX="/usr/local"`

WITHLIB_<library> Will toggle installation of optional libraries that are provided with CLS^2 . A list of available libraries and their installation status will be printed when running cmake.
Example: `cmake .. -DWITHLIB_n++=ON`

SUPPRESSi_<library> Will suppress installation of specific external libraries. By default, a fixed set of libraries is linked into CLS^2 if available. In cases where this causes problems (for instance, certain versions of libdc1394 and libusb are not compatible with each other) unneeded libraries can be turned off.
Example: `cmake .. -DSUPPRESS_dc1394=ON`

1.3 Execution and configuration

To use CLS^2 , the `CLSquare` binary is always run from the command line with an argument pointing to a valid configuration file: `./CLSquare <config.cls>`

Configuration files contain lists of *parameters*, attribute-value pairs, which are organized into *chapters*, strings with square brackets around them that denote blocks of parameters that apply to a specific module. An example could look like this:

```
1 [Main]
2 num_episodes = 1
3 cycles_per_episode = 10
4 plant = DemoPlant
```

```
5  controller = DemoController
6
7  [Controller]
8  max_action = 2.0
9
10 [Plant]
11 # the DemoPlant does not need any parameters
```

There is a wide variety of demo applications with configuration files located in the `CLSquare/demos` folder.

If Doxygen is installed on the host machine, descriptions of modules and their configuration parameters can be generated from the build folder by invoking the command `make doc`. After having been generated, the documentation can be accessed at `CLSquare/doc/html/index.html`.

2 Basic concepts

2.1 Definitions

action The action u_t is a vector of real values that describes the action applied to the dynamic system at time step t.

plant state The plant state x_t is a vector of real values that describes the internal state of a plant at time step t.

plant measurement The plant measurement y_t^1 is a vector of real values that describes the externally visible state of the plant in time step t.

system observation The system observation y_t^2 is a vector of real values that describes the part of the plant observation available for control generation in time step t.

cycle A cycle in CLS^2 is the transition from time step t to time step t+1.

episode An episode starts at time step 0 and contains a sequence of time steps where the modules of CLS^2 interact in a closed loop.

2.2 Modules

The functionality of CLS^2 is split into a number of main modules, the specific implementations of which can be easily exchanged.

Plant The plant module implements the behaviour of the system under investigation, how it behaves over time and the influence of actions applied to it. Its main functionality is to compute time discrete dynamic: $x_{t+1} = f(x_t, u_t)$ (potentially dependent on t and subject to noise). Examples:

- simple rule based system behaviour (e.g. CatAndMouse grid example)
- numerical simulations of dynamic systems (e.g. Acrobot example)
- external applications by means of communication (e.g. TCP communication to a Matlab simulation)
- real systems by communication with hardware

The plant also provides a distinct measurement: $y_t^1 = h_1(x_t)$, a function shared with the observer module.

Observer The observer provides an intermediary between the plant and the controller. In the same manner that the plant can compute a measurement from the state, the observer generates a system observation from the plant measurement: $y_t^2 = h_2(y_t^1)$

> **Observations vs. measurements :**
> *While both provide the same function, using an observer is generally preferable to plant measurements, as the former can be reused with different plants; the measurement should be $y_t^1 = x_t$. Even so, plant measurements can be generated for the sake of backward compatibility and for plant-specific operations.*

Controller The controller module receives information of the recent observation of the plant and has to compute an action. Its main functionality can be summarised as a function g: $u_t = g(y_t^2)$

Reward These modules implement the evaluation of system states for learning.

Input The input module defines the initial plant state of an episode at time step 0: x_0

Output The output module logs the information of all episodes and cycles.

Statistics In the statistics module the control behaviour can be evaluated over the episodes.

3 Advanced concepts

3.1 Cyclic execution of external commands

CLS^2 offers the possibility to execute an external command between episodes. The command can be specified as a string in the Main section of the configuration file: `call_cmd = <string>`.

```
1 [Main]
2 call_cmd_freq = <int> # (default 0)
3 call_cmd = <string>   # (default CallCmd)
```

Execution command and parameters :
The command is executed and provided with some parameters:
$<call_cmd> <episode_number> <total_number_cycles> <total_time>$

The frequency of execution is controlled by an integer parameter in the Main section of the configuration file: `call_cmd_freq = <int>`. This frequency controls the number of episodes after the command is executed. A frequency $<= 0$ will disable this feature (default), no command is called.

3.1.1 Intended use

The feature of the cyclic external command call can be used for general purposes. The intended use is for learning controllers that interact with the plant in a training procedure for a certain number of episodes. To test the quality of the recent controller during the learning procedure every n episodes another CLS^2 instance is called with a test setup and the recent controller policy. With the statistics information of these test runs we can collect an evaluation of the training procedure (e.g. without exploration or with a number of certain starting states and conditions).

An exemplary use case:

- define the learning procedure in a configuration file called `train.cls`
- in this file specify the `call_cmd` to execute a script file called `test.bash`, e.g. every episode
- define the test procedure in a file called `test.cls`, with own definition for statistic/output result files, e.g. statistics will be saved in `test.stat`
- the bash file could look like:

```
1 #!/bin/bash
2 ALLTEST_STAT="test.all.stat"
3 if [ $1 -eq 0 ]; then rm -f $ALLTEST_STAT;
4 fi
5 echo "Do testing after training episode: "$1
6 ./CLSquare test.cls
7 echo -n   $1 $2 $3 >> $ALLTEST_STAT
8 cat test.stat >> $ALLTEST_STAT
```

- Explanation: After every episode of the main instance of CLS^2 the `test.bash` is called with the desribed command line parameters. The `test.bash` will call another instance of CLS^2 with the `test.cls` configuration and concat the statistics in a file `test.all.stat`. To evaluate the learning behaviour this file can be used.

3.1.2 Plant specific behaviour

In special cases when we call another instance of CLS^2 from a running instance the plant module of the original program instance has to release some resources to allow the new instance to run. To allow these resource release and allocation a plant can reimplement the functions:

- `virtual void Plant::notify_suspend_for_aux_call_cmd() {return true;}`
  Called after the episode stopped, before the external command is executed.

- `virtual void Plant::notify_return_from_aux_call_cmd() {return true;}`
  Called after the external command has finished, before the next episode starts.

Example :
The plant is controlling a real hardware system, connected over the serial port that was opend by the init procedure of the plant and will be closed by the deinit procedure of the plant. When a external command executes another instance of CLS[2] the port cannot be opened by this new instance as the first instance has opened it already. In the plant we can implement `void MyPlant::notify_suspend_for_aux_call_cmd()` *to close the port while* `void MyPlant::notify_return_from_aux_call_cmd()` *is implemented to open it again.*

3.2 Reference Inputs

The concept of reference inputs was introduced to influence the plant during the episode by external inputs. This input represents external and changing variables of the plant behaviour for the dynamic of the plant, the measurement or the observations of the plant state. The input module will set these variables in every time step t, based on some implemented and configurable modes for each dimension, e.g.:

- random values in a certain range, changing every n-th cycle
- cyclic standard functions (sine, cosine) with configurable frequency, amplitude and phase
- linear interpolation of points given in a file
- TCP client to a GUI-tool, where sliders can be generated by string definitions in the config file

To use the concept of reference inputs, you have to reimplement an extended init method in the plant, instead of the standard method to give the dimesnion of the reference input variables (0 will disable this feature):

```
1 virtual bool Plant::init(
2   int& plant_state_dim, int& measurement_dim,
3   int& action_dim, int& reference_input_dim,
4   double& delta_t, const char* fname=0);
```

3.2.1 Exemplary use cases

- A reactor where the action has to compensate for some concetration difference, where the concentrations are dependant of the chemical reaction (internal plant dynamics) but also on the effluent of the reractor fluid (externally controlled and given by one of the reference input variables).
- A simulated dc motor tht should be robust aginst the change of the driven mass (given externally by one of the reference input variables).
- A controller for changing set-points, where the changing set-point is modelled as external changing reference inputs. The plant can compute the control deviation (placed in the measurement vector) dependant on the recent plant state and the external reference input.

Note on set-point controllers :
It would be rather natural for the implementation to give the reference input to the get_action method of the controller to compute an appropriate output action. Nevertheless this would make the interfaces rather complicated. The intended use is to provide these set-points to the plant by the reference inputs and compute some control deviation in the plant measurement. The controller receives a system observation vector and has to compute its action based on it.

3.2.2 Impact on other modules

The use of reference inputs is defined by the plant and has some affect on other modules:

- Input: The reference inputs has to be defined, using the implemented modes in the config file.
- Controller: No impact, will receive the observation of the plant state without any notice about reference inputs.
- Statistics: notified about the reference inputs, recent implementation will not care.
- Output: You can switch on the logging of reference inputs in the configuration file.

3.3 Pipes

It is possible to influence the flow of CLS^2 from outside the program using pipes. To do so, a configuration option has to be set in the main chapter: `interactive_mode=true`. Once activated, arbitrary commands can be echoed to the location `/tmp/pipe2cls`.

By default, CLS^2 supports the commands listed below. In addition, specific modules may react to additional commands by reimplementing the function `notify_command_string()`.

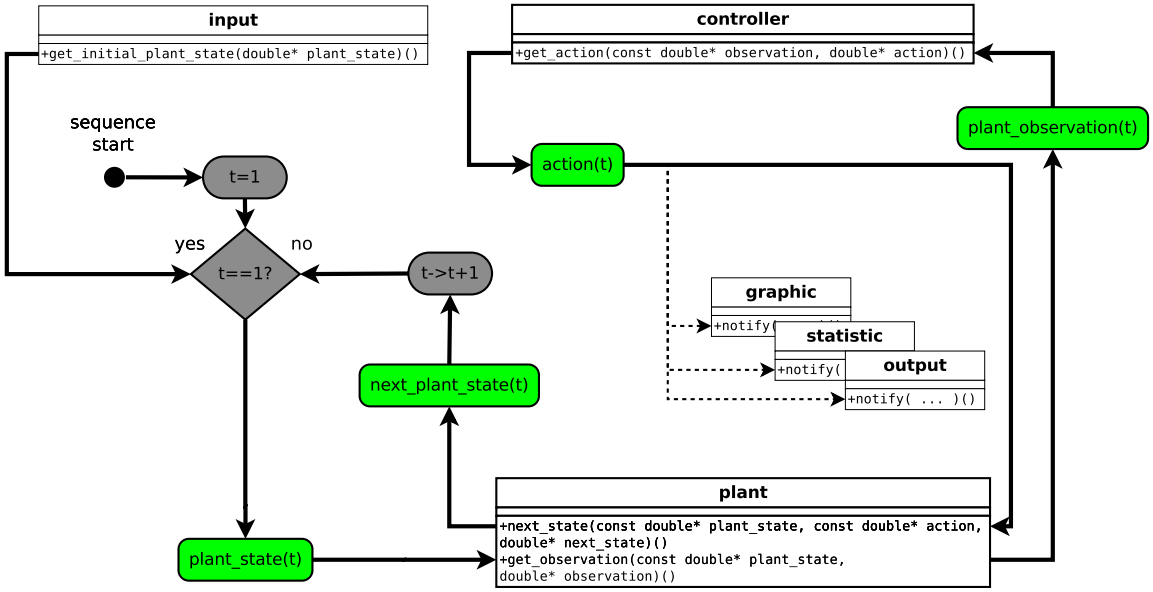
- `stop` terminated the program
- `pause` pauses the program until `start` is called
- `start` resumes the program after `pause` has been called
- `plant\_cmd $<$command$>$` passes the command <command> on to the plant module
- `controller\_cmd $<$command$>$` passes the command <command> on to the controller module
- `graphic\_cmd $<$command$>$` passes the command <command> on to the graphic module

Examples are available in CLSquare/demos/PipeDemo.

> **Note on pausing :**
> *The inbuilt command* pause *to interrupt the program is to be used with caution, as it only stops the main CLS^2 loop. While the plant is notified of the pause request as well, some plant modules are incapable on stopping immediately (causing incorrect state transitions to be observed) or at all (potentially causing damage to the hardware). Please check the capabilities of the plant when using hardware before pausing via pipe.*

4 Information and Progam Flow

4.1 The basic information flow



4.2 End of sequence

In general the end of an episode is reached when the required number of steps has been computed. However there are some special cases:

- The plant module can terminate an episode by returning false in `Plant::next_state( ... )`. This means that a state or measurement is a terminal state. This terminal state can be a failure state (leaving working range) or even a terminal state where the task is solved and the (simulated) dynamic is stopped.
- The controller ends the recent episode by returning false in `Controller::get_action( ... )`.

Real plants :
Real plants can not be stopped without taking care for the state of the real system. To prevent crashes and to lead the system into a safe state please implement safety behaviours in `Plant::notify_episode_stops()`.

5 Programming Concepts

5.1 Adding a module

To develop your own module for CLS^2 , you merely have to create a class that inherits from one of the base module types. A very minimal example for a controller module could look like the following:

```
// src/controller/ExampleController/example.h
#ifndef _EXAMPLECONTROLLER_H_
#define _EXAMPLECONTROLLER_H_
#include "controller.h"
class ExampleController : public Controller {
  virtual bool get_action(const double* x, double* u);
}
#endif
```

```
// src/controller/ExampleController/example.cpp
#include "examplecontroller.h"
bool ExampleController::get_action(const double* x, double* u)
  {u[0] = 1; return true;}
REGISTER_CONTROLLER(ExampleController, "Always returns 0.")
```

```
// src/controller/ExampleController/Sources.cmake
INCLUDE_DIRECTORIES(${CURRENT_LIST_DIR})
LIST(APPEND controller_srcs ${CURRENT_LIST_DIR}/example.cpp)
```

The file `Sources.cmake` tells cmake to actually compile your code. Signing your module up with CLS^2 is done by the macro `REGISTER_<module_type> (<class> <description>)`, which should provide a short description what the module does.

5.2 Message handling

The file global.h supports several kinds of message output macros to help to organize code and keep structure of messages. The basic macros are
`EOUT( ... )` for error messages,
`WOUT( <LEVEL>, ... )` for warnings,
`IOUT( ... )` for information handling and
`DOUT( ... )` for debug messages (only when compiled with DEBUG flag).

In general the macros can be called everywhere in the code and accept strings and pipeable string components. Examples:

```
IOUT("HelloWorld");
WOUT(10, "Careful...");
EOUT("Uuups " << my_variable);
```

These Macros will accompany the output with an identifier for the kind of message, the class name and function name where they where called. The implementation of the macros is based on ostreams that can be used in the code to output raw text without any additional information.

```
ESTREAM << "HelloErrorStream " << 123 << someFunc.str() << "\n";
```

There are additional macros and streams, e.g. conditioned versions with verbosity level or warning outputs (see the file `global.h`). It is possible to reimplement these macros and their functionality.

Attention :
Please try to use these macros in the whole code base and your own modules. A future release may contain an advanced Message handler that will use these macros as the basic interface.

5.3 Error handling

For a convenient error handling during initialization phase of modules please use the combination of `EOUT( ... )` to describe the error that occured and a return value of false to show that an error has occured. An error in initialization phase will stop the program immediately without further calls.

In other cases please use the macro `CLSERR( ... )` defined in the file `global.h` that can be called anywhere in the code. As argument the macro accepts strings or pipeable string components. Examples:

```
1 CLSERR("An error occurred, please fix this");
2 CLSERR("Out of range: " << my_variable );
```

This macro is implemented with an exception that is catched in the mainloop. With the help of the macro the exception mechanism is hidden for the user. Other implementations of the macro are possible. The macro CLSERR(...) helps to signal the mainloop that the program has to be terminated. The mainloop will do some deinit calls where it is necessary. For example when using a real hardware plant the plant has to be controlled into a safe state before the program can exit.

Attention :
Please try to avoid exit() calls in the code. Using exit() calls could damage real hardware plants.

5.4 Initialization

All modules call a function called `init()`, albeit with differing arguments, which needs to be implemented. Any arguments that are not declared constant are expected to be set by the respective module, particularly the state dimension by the plant and the action dimension by the controller.

Those that are passed the argument *chapter* should use it when reading config files; for instance, a value parser should be created as `ValueParser vp(fname,chapter==0?"Controller",chapter)`.

Attention :
Please avoid hardcoding chapters like `ValueParser vp(fname,"Controller")`, since for such modules proper behavior cannot be guaranteed when used by meta-modules.

5.5 Adding libraries

If your module requires specific libraries, you should use the provided macros to check if they have been properly installed and to print a standardized error message if not. A typical `Sources.cmake` file with dependencies should look like this:

```
1 CHECK_INTERNAL(ExampleModule n++)
2 CHECK_EXTERNAL(ExampleModule gtest OPTIONAL)
3 IF(NOT MISSING_DEPENDENCY)
4   # register the module's source files
5 ENDIF()
```

You will notice the distinction between internal libraries, which are included in the CLS^2 package and installed automatically, and external ones, which are installed through your operating system's package management or by hand. Available internal and external libraries are listed when you run cmake in the `build` folder.

If you want to use a library in your module that is not included with CLS^2 by default, you can add it yourself. The procedure to do so differs between internal and external libraries.

For external libraries, you mostly just have to point cmake to their location. To do so, you will need a Find-script that determines the relevant paths; scripts for common libraries are provided by cmake itself, but for others the script `Find<libname>.cmake` needs to be written by hand and placed in `CLSquare/cmake/`. Once you have this script it needs to be registered with the project in

CLSquare/cmake/CMakeLists.txt by calling a few macros and setting some variables; the process is detailed in the file itself.

Compiler flags :
Module registration is not only used for checks in cmake, but also defines compiler variables of the form FOUND_<libname> that allow checking a library's presence from within C-code.

In contrast, internal libraries are treated very much like modules and placed in their own subfolders in CLSquare/src/lib. Files are added to the project and the library is created in the file CLSquare/src/lib/<libname>/CMakeLists.txt; there are several examples available to show you how this is done. Just like modules, libraries should check for the presence of other libraries by using the CHECK_-macros. Unlike modules, they also need to come with a file CMakeOptions.txt that provides meta-information to cmake that is required to generate the option to turn a library on or off. Such a file should look like this:

```
1 SET ( CURRENT_LIB_NAME ExampleLibrary )
2 SET ( CURRENT_LIB_DEFAULT OFF )
3 SET ( CURRENT_LIB_TEXT "Set to ON to enable the example library." )
```

The string <CURRENT_LIB_NAME> will be used both to toggle installation via -DWITHLIB_<CURRENT_LIB_NAME> and to check its existence through CHECK_INTERNAL(<CURRENT_LIB_NAME>).

Attention :
For libraries intended for release that are not core libraries, the parameter CURRENT_LIB_DEFAULT should be off. The same applies to any hardware libraries.

6 Notes

6.1 Known problems

- The frameview tool by Arthur Menke occasionally malfunctions on modern, faster computers. If communication is too fast, commands may be dropped. This affects both the initialization phase as well as the drawing routines during episodes. Every sending of commands should be followed by a sleep of 1-10ms (see CatAndMouse graphic for instance).

References

[1] CMake. http://www.cmake.org.

[2] Eclipse Foundation. http://www.eclipse.org.

[3] KDevelop. http://www.kdevelop.org.

[4] Qt Creator IDE. http://qt.nokia.com/developer/qt-creator.