ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INGEGNERIA DELL'ENERGIA ELETTRICA E DELL'INFORMAZIONE

INGEGNERIA DELL'AUTOMAZIONE

TESI DI LAUREA
IN
ADVANCED CONTROL SYSTEMS M

5C COMPLIANT LOW-LEVEL COPTER CONTROLLER

Candidato:
Federico Maria Rossi

Relatore:
Prof. Lorenzo Marconi

Anno Accademico 2016/17
Sessione III

SUPERVISORS:
Prof. Lorenzo Marconi
Dr. Herman Bruyninckx

ABSTRACT

Modern engineering faces the problem of implementing systems in a flexible, scalable and reusable way. This problem becomes exponentially hard as the complexity of systems increase, unless systems and systems components developers foresees and find ways to overcome such scaling problematic.

Many techniques try to address the scaling problem by defining paradigms to standardize or, at least, harmonize the efforts of different developers teams. Nevertheless developers tends to underestimate their importance, either because of lack of education, either because they consider overcomplicated or pointless their adoption in small to medium projects. This is indeed true sometimes: the lack of mature, open, free, supported and widely accepted tools makes the benefit one can archive from the introduction of such techniques void w.r.t. the effort required to learn and adopt them. This is a vicious cycle, however projects like ROS show how also small projects can benefit from a proper and formal approach to the problem, especially once a "critical mass" of adopters is reached.

ROS is a open-source component based software framework, where components are usually written and released by third party developers for being reused, avoiding the necessity of "re-inventing the wheel"

This thesis provides a study case: the Model Driven Engineering (MDE) methodology and the 5C "separation of software behavior" paradigm are applied in the development of a novel low-level copter controller. The following chapters will briefly introduce such techniques and will show how to apply them in a real-world problem.

As already highlighted, such small and stand-alone application does not gain enough benefit from this approach to justify the effort required, however the purpose of this thesis is to support and popularize a certain attitude to problems approach, to release working example code, contribute to the maturity of the underlying software framework and, last but not least, to enrich the knowledge and the skills of author readers.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth

1

ACKNOWLEDGEMENTS

First and foremost I want to thank my family for all their love and encouragement. For my parents who raised me with a love of science and supported me in all my pursuits.
I am grateful to my supervisor Prof. Lorenzo Marconi that helped me in my thesis and to my co-supervisor Dr. Herman Bruyninckx that really opened my mind.
Thanks to the whole ROB group, especially Nico Huebel, Enea Scioni and Lin Zhang for their support and their company during my work.

Part I

BACKGROUND

2 INTRODUCTION

Every day new technologies are available on the market making the realization of advanced robotics applications easier and accessible to large public. This is leading to a spread of a number of different approaches to typical problems causing many redundant implementations of relatively common problems and, usually, waste of energy. This thesis investigates a flexible and scalable approach to the development of software for robotics, trying to maximize the potentiality of the underlying components while abstracting the complexity of the low-level implementation. The software has been implemented on a new component-based software framework, trying to surmount the limits of other (sometimes more mature) frameworks like Open-RTM, ROS, OPRoS and SmartSoft.

Details on this work will be explained in the details in this thesis; the sources of the software and everything necessary to reproduce the results can be found in the following git repositories.

- The LaTeX source of this thesis:
  `https://github.com/fmr42/MasterThesis`

- The UBX project forked from the original repo to support system-wide installation:
  `https://bitbucket.org/fmr42/ubx-core/`
  `https://bitbucket.org/fmr42/ubx-base/`

- The controller here presented and the `bcsk` library it depends on:
  `https://github.com/fmr42/bcsk`
  `https://bitbucket.org/fmr42/ubx-control/`

This software has been developed and tested on a Fedora 21 GNU/Linux system running on AMD64 processor and on an Ubuntu Trusty 14.04 running on an ARM processor, however everything as been studied to be highly portable and, once satisfied the dependencies, should compile an all POSIX-compliant platform (Linux, OSX, BSD).

3 TERMINOLOGY

3.1 MODEL DRIVEN ENGINEERING

Model driven engineering (MDE from now on) is a promising approach to problem solving, especially for complex applications. MDE alleviate the complexity of the problem by using a domain specific language to express domain specific concepts. This aims to maximize compatibility between systems, especially those implemented by different individuals or different teams: such goal is archived by making terminology and design patterns uniform. While MDE is quite popular in software engineering (especially in large software projects), it is still considered underestimated in the robotics field.

3.2 METAMODELLING

A model is an abstraction of a real system and it exactly describes its behavior; multiple systems can conform to the same model, meaning that they behave the same (e.g. multiple implementations of a control system). The same way, a meta-model is a model of model, meaning that it is yet another abstraction; it is used to describe at a more conceptual level rules and constraints of a given class of models. To make an example, in the study case presented in this thesis the extended Kalman filter conforms to the mathematical model of an extended Kalman filter which in turns conforms to the meta-model of a *Bayesian estimator*, like the models of other types of filters, e.g. particle filters.

3.3 DOMAIN SPECIFIC LANGUAGES

Domain specific languages (DSL) are languages specifically designed to easily express schemes or patterns commons within a certain field, without the need of details of some particular implementation. DSL are commonly used to describe models and meta-models. A DSL is not necessarily a new language: it will be show for example that in UBX a particular Lua table is used to describe the structure of a component-based software.

4 APPROACH TO THE PROBLEM

In the following two different control strategy are compared: the *sense-plan-act* and the *task-skill-motion* strategy. Pro and cons of the two approaches are compared to justify the choices made within this thesis.

4.1 SENSE-PLAN-ACT

Although this may seems the fastest and simpler approach, on a long-term period causes many redundant and bug-prone re-implementation of the same patterns.

A common approach to control is to solve the overall complex problem by combining the low-level functionalities of the underlying platform into a monolithic control loop, resulting in a controller strictly tied to that specific problem and therefore poorly re-usable.
From a long-term prospective, it is wider to spend more time during the development to come out with a scalable solution for the future.

4.2 TASK-SKILL-MOTION

Sometimes it is more convenient to re-implement from scratch then re-use old (maybe well-tested!) solutions...

Although sense-plan-act is the most widely used control strategy, other alternatives have been proposed. As in MDE we use different DSL's to represent the behavior of the system to be controlled at different levels of abstraction, we can in the same way define multiple control loops, one for every level of behavior abstraction. In many complex application it is usually present a mission composed by more task. Each task can be realized employing the skills of the platform and each skill if archived by low-level motion control.

We can see that the mission is a far more abstract control problem than the low-level actuation: a mission can be coordinated by a software or human supervisor, usually with more relaxed time constraints, and may distribute the necessary tasks to more agents. Each agent may than decide how to complete the task autonomously, on the basis of its own skills (which obviously depend on the actual hardware platform on which the agent is running). In turn, the skills are composed by one or more low-level motions.

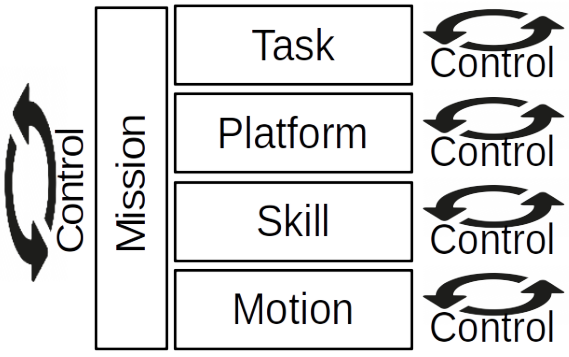


Figure 1: The task-skill-motion control strategy.

The control problem proposed in this thesis implements a skill (i.e. the ability of the vehicle to fly around) by combining motions (that are the actual control inputs to the actuators). This skill is actually controlled by a human interface but everything has been studied to integrate the solution to such context.

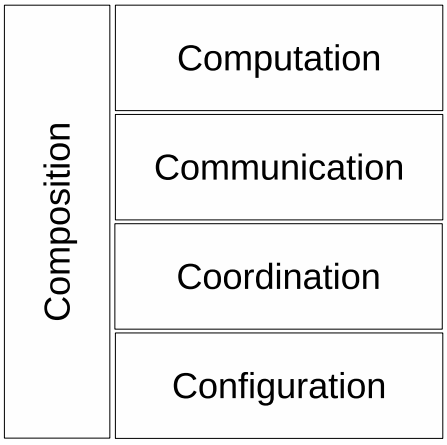
5 THE 5C'S OF ROBOTICS

5.1 50 YEARS OF

Most engineering problems are still solved as particular and standalone problems, so if a solution is not already available for that problem, a new one is developed with no concerns of making that solution re-usable for similar problems. Industry often force the researchers to opt for a quick, simple and cost-effective solution, ignoring the fact that tens or hundreds (maybe thousands!) of other engineers already addressed that problem, reimplementing the solution almost each time. Certainly it would be useful to reuse the same solution if possible, eventually improving it if it does not exactly fit the problem.

5.2 SEPARATION OF CONCERNS

The separation of concerns if an approach to the code re-usability problem promoted by BRICS. Researchers had identified 4 possible concerns, or behavior, that should be kept distinct with the goal of decoupling them and allowing an easy and standardized practice to reuse them across different applications, but also to make it easy to reimplement or substitute them within the *same* application. These behaviors are computation, communication, coordination and configuration[5].
An additional concern was added to give this practice a good trade-off between scalability and complexity: more components can be composed to obtain a new component also having only one of the listed possible behaviors.



5.2.1 *Computation*

Computation is the core of a system functionality: it usually require read and/or write access to data, configuration and synchronization to produce the expected result.

5.2.2 *Communication*

Communication is responsible for data transmission between different computational components with the proper quality of service. A communi-

cation component usually store and retrieve data for computational components. In case a communication block do actually communicate with *other* agents, it is named a *bridge*.

5.2.3 *Coordination*

Coordination is responsible for the actual discrete behavior of the system and can be represented by a finite state machine, in other words it ensures the correct computational behavior at the correct time.

5.2.4 *Configuration*

Configuration is responsible for the correct setup of all the components; it should be kept distinct and, indeed, configurable. A bad practice in software engineering is to hard-code configuration values; yet there are obvious exceptions like mathematical constants, e.g. π.

5.2.5 *Composition*

While separation of the previous 4 concerns gives a way to de-couple a system components, it is still necessary to introduce a trade-off between modularity and simplicity of the implementation. For this reason the last introduced behavior is *composition*, which gives the capability of combining components (with different concerns) into a macro component which in turn has one of the described behavior.

Part II

THE STUDY CASE

6 THE CONTROL ALGORITHM

Unmanned aerial vehicles (UAVs) are a class of aircraft under the control of either a remote human pilot or a on board computer. UAVs were originally developed for military purposes, although recent advances in technologies, like Li-Ion batteries and brush-less motors, have brought this type of aircraft to others scopes of usage, like commercial, scientific and also recreational. Although most UAVs are fixed-wing aircraft, rotorcraft are becoming popular too due to the low price and ease of maintenance and run. The demand for this type of drone is increasing and the global market for commercial use is expected to grow in the next few years, thats why the study case is a low-level controller for helicopters, ducted-fan tail-sitters and multi-propeller helicopters.

Quad-copter, more commonly referred to as drones*, are commercially available to a large amateur public.*

6.1 A NOVEL APPROACH TO THE COPTER CONTROL PROBLE

A common control strategy is a cascade feedback: an outer loop for the position and an inner loop for the attitude. The inner control loop is usually composed by 3 control loop for the roll, pitch and yaw attitude angles. While this is a easy and intuitive approach to the problem, a more advanced one can be developed: the rest of this chapter describes a novel approach as presented in [6].

6.2 QUATERNIONS FOR ROTATIONS REPRESENTATION

Although the attitude has 3 degrees of freedom, any minimal parametrization suffers of singularity problems (or gimbals-look).

Rotations of 3D objects in a 3D world can be expressed by 3 scalar quantities, usually Euler angles or roll-pitch-yaw angles (RPY).

A very common alternative representation is rotational matrices: those do not suffer of singularities problems but are quite redundant since 9 scalar values are used to express just 3 degrees of freedom.

A rotation matrix is a 3x3 orto-normal matrix, so although it is composed by 9 scalar values, the matrix itself has just 3 degrees of freedom

A good trade-off between compactness and ease of representation is archived with unit quaternions, a notation that allows to represent rotations in a compact form and with no gimbals-lock problems. Another advantage to consider is numerical stability: numerical drifts make both rotation matrices and unit quaternions loose their own properties: respectively being orthonormal and having module one; it is however way easier to scale a vector than force the orthonormal structure of a matrix. More details can be found in [10] and [11].
No insight in Hamiltonian math is reported there, it just follows a

note about the notation used in the following chapters: in general a quaternion q is given by:

$$q := \begin{bmatrix} \eta \\ \epsilon \end{bmatrix} \tag{1}$$

where η is the *real* component and ϵ is the *pure imaginary* vector component of q and the Hamilton product between 2 quaternion is indicated by the symbol $\oplus$.

6.3 THE COPTER MODEL

The dynamic model of the copter is given by:

$$\begin{aligned} M\ddot{p} &= -u_f R e_3 + M g e_3 \\ \dot{R} &= R S(w) \\ J\dot{w} &= S(Jw)w + u_\tau \end{aligned} \tag{2}$$

Where we call p the position vector of the center of gravity of the copter in the inertial reference frame $\mathcal{F}_i$ and w the angular velocity of the vehicle in the body reference frame $\mathcal{F}_b$. R is the rotation matrix representing the orientation of $\mathcal{F}_b$ wrt $\mathcal{F}_i$, J is the inertia matrix of the system and u_f and u_τ are respectively the force and the torque vector generated by the propellers. The $S(m)$ function generates a 3×3 matrix from vector m such that the matrix product between $S(m)$ and another 3×1 vector n correspond to the cross (vector) product between m and n. S can be constructed as:

Note that the while the u_f is a scalar force which direction always aligned with the z axis of $\mathcal{F}_b$, u_τ is a 3-dimensional vector.

$$S(m) := \begin{bmatrix} 0 & -m_3 & m_2 \\ m_3 & 0 & -m_1 \\ -m_2 & m_1 & 0 \end{bmatrix} \tag{3}$$

The copter model can be rewritten using quaternions to represent rotations by mean of the Rodriguez formula[7]:

$$\mathcal{R}(q) = I + 2\mu\mathcal{S}(\epsilon) + 2\mathcal{S}(\epsilon)^2$$

as

$$\begin{aligned} M\ddot{p} &= -u_f \mathcal{R}(q) e_3 + M g e_3 \\ \dot{q} &= \frac{1}{2} q \oplus \begin{bmatrix} 0 \\ w \end{bmatrix} \\ J\dot{w} &= S(Jw)w + u_\tau \end{aligned} \tag{4}$$

6.4 POSITION CONTROL LOOP

We can write the position error dynamic as:

$$M\ddot{\overline{p}} = -u_f \mathcal{R} e_3 + Mge_3 - M\ddot{p}_R \tag{5}$$

And define the control force as:

$$\begin{aligned} v_R^c(\ddot{p}_R) &:= Mge_3 - M\ddot{p}_R \\ v^c(\overline{p}, \dot{\overline{p}}, \ddot{p}_R) &:= v_R^c(\ddot{p}_R) + \kappa(\overline{p}, \dot{\overline{p}}) \end{aligned} \tag{6}$$

where $\kappa(\overline{p}, \dot{\overline{p}})$ is a feedback action that can be computed as follow:

$$\begin{aligned} \zeta_1 &:= \overline{p} \\ \zeta_2 &= \dot{\overline{p}} + \lambda_1 \sigma(\frac{k_1}{\lambda_1}\lambda_2) \\ \kappa(\overline{p}, \dot{\overline{p}}) &:= \lambda_2 \sigma(\frac{k_2}{\lambda 2}\zeta_2) \end{aligned} \tag{7}$$

where k_1, k_2, λ_1 and λ_2 are parameters to be tuned.
In Equation 6 is mandatory to respect the constraint

$$R_R e_3 = \frac{v_R^c(\ddot{p}_R)}{\|v_R^c(\ddot{p}_R)\|} \tag{8}$$

The control scalar control force is than computed as:

$$u_f := \|v^c(\overline{p}, \dot{\overline{p}}, \ddot{p}_R)\| \tag{9}$$

Equation 9 gives the total thrust the propeller should generate, however as already explained it is also necessary to compute the torque necessary to correct the vehicle attitude.
Section 6.5 will describe the attitude control low.

6.5 ATTITUDE CONTROL LOOP

The torque control vector is computed in a control loop nested into the position control loop, therefore, as usual in cascade control loops, it has a faster dynamic. We start by defining the error attitude quaternion and the error angular velocity vector:

$$\begin{aligned} \overline{q} &= q_c^{-1} \oplus q \\ \overline{w} &:= w - \overline{w}_c \end{aligned} \tag{10}$$

with

$$\overline{w}_c := \mathcal{R}(\overline{q})^T w_c \tag{11}$$

and q_c computed as explained in Section 6.6.

Then the control torque is given by:

$$\begin{aligned} u_\tau &= u_\tau^{FF}(\overline{q}, w_c, \dot{w}_c) + u_\tau^{FB}(\overline{q}, \overline{w}, \overline{h}) \\ u_\tau^{FF}(\overline{q}, w_c, \dot{w}_c) &= J\mathcal{R}(\overline{q})^T \dot{w}_c - S(J\overline{w}_c)\overline{w}_c \\ u_\tau^{FB}(\overline{q}, \overline{w}, \overline{h}) &= -k_p \overline{h}\overline{e} - k_d \overline{w} \end{aligned} \tag{12}$$

In the previous formula k_p and k_d are positive gains and $\overline{h} = \{-1, 1\}$ is obtained by the hybrid function $\mathcal{H}_c$

$$\begin{aligned} \mathcal{H}_c &= \begin{cases} \dot{\overline{h}} = 0 & \overline{h}\overline{\eta} \geqslant -\delta \\ \overline{h}^+ \in \overline{sgn}(\overline{\eta}) & \overline{h}\overline{\eta} \leqslant -\delta \end{cases} \\ \overline{sgn}(s) &= \begin{cases} sgn(s) & |s| > 0 \\ \{-1, 1\} & s = 0 \end{cases} \end{aligned} \tag{13}$$

$\eta \in (0, 1)$ is the hysteresis threshold.

The control problem is more extensively explained in [6], however the article does not specify how to compute the reference attitude R_R; this problem is addressed in the following section.

6.6 ATTITUDE SET-POINT GENERATION

The attitude reference orientation should be computed satisfying constraint 8. The problem has just 1 degree of freedom which, for standard uses cases (i.e. for non acrobatic maneuver), is the vehicle yaw angle.
This thesis proposes an algorithm to compute the reference rotation matrix R_R based on geometric projection. Considering the yaw angle given, then the heading vector is defined as:

$$d(yaw) = [\cos(yaw), \sin(yaw), 0]^T \tag{14}$$

The matrix R_R is computed column-by-column as follows:

$$\begin{aligned} R_R e_3 &:= \frac{g e_3 - \ddot{p}}{\|g e_3 - \ddot{p}\|} \\ R_R e_2 &:= \frac{R_R e_3 \times d(yaw)}{\|R_R e_3 \times d(yaw)\|} \\ R_R e_1 &:= \frac{R_R e_2 \times R_R e_3}{\|R_R e_2 \times R_R e_3\|} \end{aligned} \tag{15}$$

This technique allows to easily compute the reference rotation matrix that can be then easily converted to the quaternion rotation thanks to

the Rodriguez formula. The control attitude required in Equation 10 is then computed as follows:

$$\begin{aligned} R_c e_3 &:= \frac{v_c}{\|v_c\|} \\ R_c e_1 &:= \frac{R_R e_1 \times R_c e_3}{\|R_R e_1 \times R_c e_3\|} \\ R_c e_2 &:= \frac{R_c e_3 \times R_c e_1}{\|R_c e_3 \times R_c e_1\|} \end{aligned} \tag{16}$$

end finally

$$q_c = \mathcal{R}^{-1}(R_c) \tag{17}$$

7 UBX FRAMEWORK

UBX is a novel framework created to support the development of 5C compliant programs. The coder does not have to write and compile an executable (indeed, the *behaviors* of the implemented algorithm should be kept isolated) but have to write and compile special *function blocks* which in turns implements those behaviors and are then composed *at runtime* by UBX to produce the desired result. Blocks are actually nothing more than functions contained in libraries, the path of those libraries and how the ports of the blocks should be connected is specified in a configuration file.

7.0.1 *Environment*

A UBX program instance is named *node*. A node is a set of blocks properly connected. There are 3 types of blocks:

- i-blocks: those blocks are capable of storing, transmitting and/or receiving data; they implement the communication and configuration behavior of the node (therefore i-blocks should not manipulate data); they only have an input port (to which data can be written) and an output port (from wich data can be read); i-blocks capable of communications out of the scope of the node are named *bridges*;
- c-blocks: are responsible for the computational behavior of the node; when triggered, they can read and/or write from/to i-blocks and perform calculus; c-blocks can have an arbitrary number of input/output ports and must have a trigger port;
- s-blocks: implement the coordination behavior of the node, they can *trigger* (i.e. activate) c-blocks or other s-blocks to produce the overall node expected behavior; s-blocks can have an input trigger port and one or more output trigger port;

Blocks have a life cycle: when instantiated a block is in a pre-init state; by calling an init function an instance reaches a pre-start state from which it can jump to a started state by calling the start function. Once started, c-blocks can be triggered (which cause the c-blocks execute the step function) while i-blocks call the read or write function on respectively read and write operations.
The UBX framework is responsible for calling the state-transition function of the blocks and always pass as argument a memory pointer

to the so called *context* of the instances (which is unique for each instance); among other things, the context contains a *private data* pointer the instance can use to allocate, indeed, private data. Blocks instances are not aware of blocks instances they are connected to, they do only see pointers (contained in the instance's context) passed by the UBX framework.

Thanks to the private data, c-blocks are actually capable of storing data even if formally they should not; this feature was introduced for storing configuration but can be exploited for other purposes.

7.0.2 *Ports*

c-blocks and i-blocks are connected by linking their data ports. A data port has a data type which is predefined by the coder as a `struct`, typically the data type contains both meta-data and payload data: the meta-data contains an `uri` to the model to which the payload conforms to.

7.0.3 *c-blocks*

c-blocks are pure computation: they should not store any information or data apart from their own configuration: once triggered, they should just read data, perform computation and write out the output data. They implements the *computation behavior*.
The life cycle is composed by an *initialization* phase, a *start* phase, one or more *step* phase, a *stop* phase and a *clean-up* phase. Each phase is implemented in a function, the UBX framework is resposable for calling the functions; UBX passes one single arguments to the function: a pointer to a `ubx_block_t` C struct.
During the initialization phase a c-block should perform all the preliminary operations it needs and it should reach a state where it is ready to start with no error. Typical operations during the initializations phase is:

- Check for the availability of hardware resources;
- Allocate the memory necessary during normal operations;

During the start phase a c-block should finish its configuration and be ready for being triggered; a typical operation is set-up all the parameters. The step phase is usually triggered by an i-block and make the c-block perform its functionalities.

7.0.4 *i-blocks*

i-blocks implement the *configuration and computation behavior*: they are used to store and communicate data (in the main memory, in files, on the network, ecc.) but should not perform non-transparent manipulation of data. If an i-block is able to communicate out its own UBX context, than it is named as *bridge*. The most common type of i-blocks, at

least in this application, is a pool of allocated system memory used to exchange data between c-blocks: a read request from a c-block make the i-block write out the stored data while a write operation from a c-block make the i-block store the incoming data.

7.0.5 *s-blocks*

s-blocks take cares of *coordinantion*. Their role is to trigger c-blocks or other s-blocks. In this application a single s-block is necessary: as it will be showed this single s-block iterate the control loop at a given timing.

7.0.6 *Installation*

The UBX team uses a stand-alone package[8] containing the UBX framework and few demo blocks; this package is not meant for installation. `git` can be used to download the project:

```
1 git clone https://github.com/UbxTeam/microblx
```

However some works has been done in the context of this thesis, some commits have been accepted in the official UBX project, others were not. One important accepted work is a deploy system for the `ubx_launcher` tool[2]. With this feature the main node launcher is able to also read a `.udc` file containing the configuration to start the blocks in the proper order, without having to access the web interface to manually initialize and start the blocks nor having to create custom deploy scripts. The deploy file can be (optionally) passed via an argument after the `-d` switch.

Chapter 10 contains examples on how to lauch the nodes.

In the context os this thesis however the official project has been forked and splitted into multiple git repositories to organize the work and bound the scope of each of them. In the details:

- ubx-core: contains the core library and tools from the original UBX project, no blocks there. The link to the repository is: `https://bitbucket.org/fmr42/ubx-core`
- ubx-base: contains base blocks from the original UBX project: `https://bitbucket.org/fmr42/ubx-base`
- ubx-control: contains, among other things, all the blocks necessary to run the copter controller and the configuration and deploy configuration file for the controller simulation nodes; the link to the repository is: `https://bitbucket.org/fmr42/ubx-control`

To download the projects move to the preferred workspace and run:

```
git clone https://bitbucket.org/fmr42/ubx-core
git clone https://bitbucket.org/fmr42/ubx-base
git clone https://bitbucket.org/fmr42/ubx-control
```

`ubx-control` depends on the following library too:

```
git clone https://github.com/fmr42/bcsk
```

Prior to compile the projects, be sure to satisfy the required dependencies:
`luajit`, `libluajit-5.1-dev` (>=2.0.0-beta11, for scripting (optional, but recommended)
`clang++` (only necessary for compiling C++ blocks)
`gcc` (v4.6 or newer) or `clang`
only for development: `cproto` to generate C prototype header file

Then, install the projects by running:

```
make
sudo make install
```

into each cloned projects folders. The `install` make goal is not implemented in the original project but is very useful because it copies the shared libraries and the scripts to the proper folders and allow a more natural use of the tool-chain.
Chapter 10 will explain how to launch the nodes while Chapter 9 will explain the implemented blocks in the details.

Part III

REALIZATION

8 MODELLING

Most of the modeling work has been performed on the data since there is no component meta-model to conform to. However, as a proof-of-concept exercise, it will be reported a conceptual example of how a model for this controller may look like.

8.1 DATA MODEL

Data modeling is partially supported in UBX in the sense that only a M0 and M1 model can be expressed. In the details, each data port defined in i-blocks and c-blocks have a predefined data-type; UBX only defines primitives data-types like integers and floating-points numbers and characters. Additional and composed types can be defined however; MicroBLX is not a mature project and therefore data-types are hand-written and stored in the `types` directory of the `ubx-control` project, but the idea behind data-types models is that an abstract meta-model should be instead be defined by the user, for example it is possible to do so using the JSon language, and leave to a toolchain the dirty work of generating the language-specific headers files defining those data-types. Vice-versa it should be possible from the data instance to go back to its model, thats why all the data types defined have a meta data field that should be used to store the URI of the relative model.

8.1.1 *State data model*

The data meta-model of the state is a probability function representing an estimate of the real state of the system. This formulation is too general for being used with a Kalman filter, but may be useful if a more advanced estimator would be introduced, for example a particle filter. In this particular application we simply suppose to have a probability function representable with a Gaussian function, therefore a vector and a covariance matrix have been used to represent the state.
The state vector is a composition of the position vector and its first and second derivative and the orientation quaternion and its first and second derivatives (with a total size of 21 elements). The covariance matrix is accordingly a 21×21 elements matrix.

8.1.2 *Sensor data model*

The sensor data model is similar to the state data model in the sense that is composed by a vector of arbitrary length (specified in an integer variable) and its covariance matrix. Please note that while the covariance matrix is a property of the sensor data only, the sensor model needed by the Kalman filter require the knowledge of both the state and the sensor data types, therefore it is specified as an input of the filter, not a property of the sensor data.

8.1.3 *Set-point and control action model*

The set-point and the control action can be represented using just a vectors: in this study case the set-point vector is a vector with the same structure used to represent the state, but no covariance matrix is necessary; the control action is a composition of the scalar thrust and the vector torque to be generated by the propellers.

8.2 FILTER MODEL

The meta-model of the extended Kalman filter is a Baysian estimator, multiple filters conforms to this type of filter like, for example, the particle filter. It is important to conform to the meta-model since conceptually it could be possible to have an automated model transformation tool able to select alternative filters to replace the original one and a supervisor able to switch the filter node on-line whenever necessary.
As example, think to have a controller that usually flies outdoor relaying on a GPS signal filtered by a Kalman filter for position estimation, but also embedded with a particle filter for indoor map-based navigation; it could be possible to have a software supervisor able to detect the feasibility of switching the GPS Kalman filter with the map-based particle filter and able to hot-swap those filters, thus allowing the vehicle to adapt the control system to the environment. For this to be possible, the supervisor should be able to perform model-transformation of both the data model and the filter model, thats why whenever this will be possible it will be necessary to have models of the components. A scheme in the following page shows conceptually the structure of the filter model.

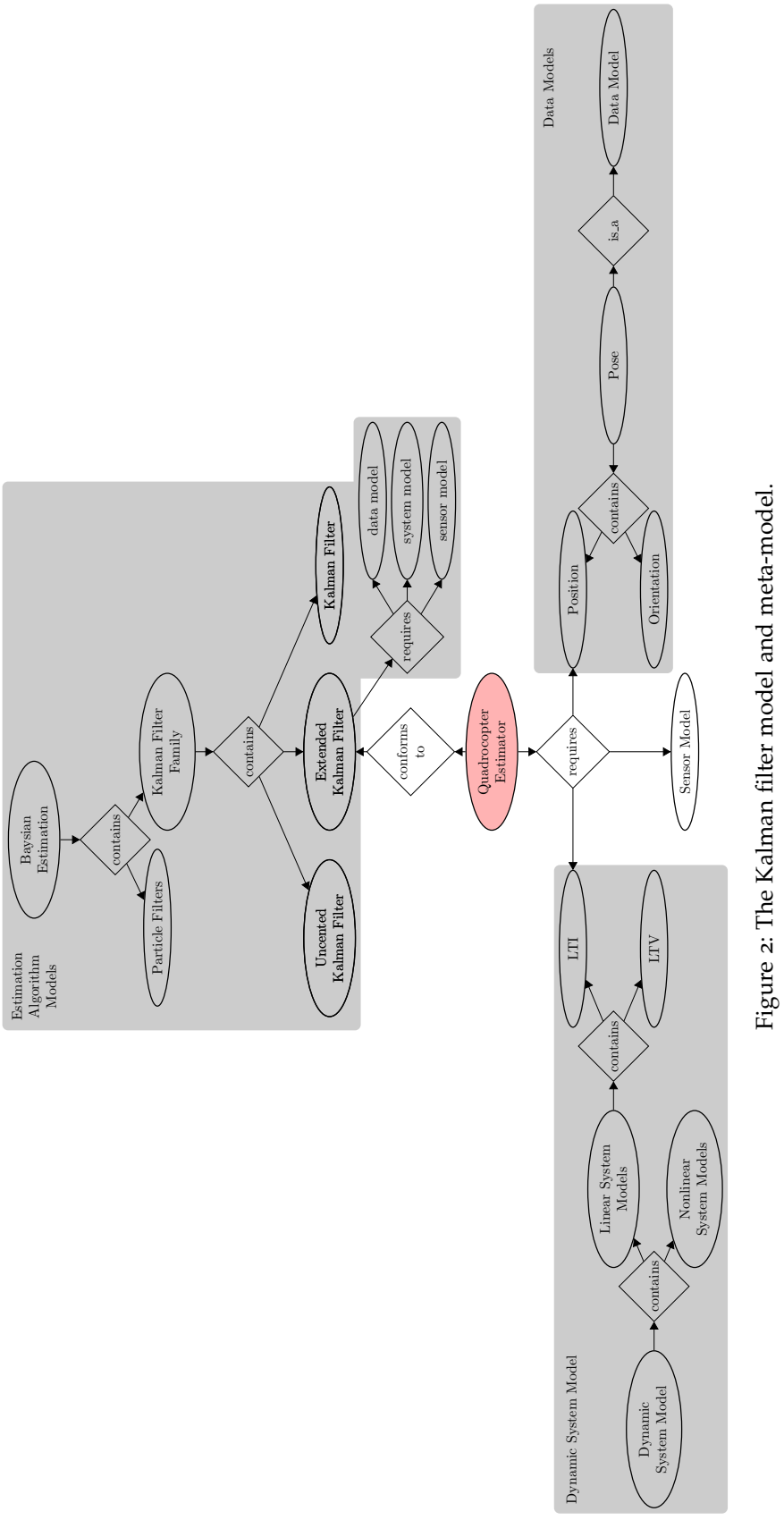


Figure 2: The Kalman filter model and meta-model.

9 IMPLEMENTATION

This chapter gives an introspection of the UBX project and the derived work. There are 3 section: one fore the core libraries, necessary to run nodes, one for the blocks from the official project and one for the blocks designed and implemented in the context of this thesis.

9.1 CORE UBX LIBRARY

This library is composed by the core shared library `libubx.so` and the Lua scripts and macros necessary to link the component functions together. Please note that since all the UBX functionalities are implemented in C or C++ function, the native Lua interpreter is not capable of linking them; it fact, the LuaJIT interpreter has been used. LuaJIT implements the FFI library which allows calling external C functions and using C data structures from pure Lua code[3].
The `Makefile` has been expanded beyond the compilation directives already present in the official UBX project: it also contains directives to install the core libraries, the scripts and the macro into the system, plus generates a proper configuration file automatically loaded by the `ubx_launch` command.

On ?? Orocos has been used to deploy the node since it supports Lua components, the Orocos Lua interpreter however if currently the native one, for this reason the project itself has been forked and re-linked to the LuaJIT library to allow execution of a MicroBLX node on top of Orocos.

9.2 OFFICIAL UBX TEAM BLOCKS

The `ubx-base` projects contains blocks from the official UBX team; some of those blocks are not used or are re-implemented while others are fundamental for the simulation to work. Moreover it contains the native (basic) data types like `int`, `char`, `long`, `long long`, `short`, `double` and `float`. It worths to highlight the presence of two blocks:

- `ptrig`: an s-block that, once properly configured, trigs a list of c-blocks in a timed loop;
- `webif`: this c-block does not need to be triggered nor to be connected: it is able to open a html interface useful to configure, control and monitor the node it is added to.

9.3 STUDY-CASE BLOCKS

This section describes the blocks designed for the purpose of controlling a quadcopter.

9.3.1 *Copter controller (`copter/ctrl_copter`)*

This block implements the control algorithm as explained in Chapter 6 and [6]. It reads the state and the set-point of the vehicle and writes a control action in the form of vectored-thrust control action (i.e. the scalar thrust and the vectored torque the propellers should generate).
It is implemented in C++ in order to exploit the capability of the mathematical library Armadillo that greatly simplify operations on vectors and matrices.

9.3.2 *Extended predict-only Kalman filter (`copter/kf_predict_copter`)*

Models the behavior of the vehicle: reads the estimate of the state and the control action and generate a new estimate of the state (which is usually written back to the same i-block the state was read from).
This block is written in C++ and use Armadillo too.

9.3.3 *Update-only Kalman filter (`control/kf_update`)*

Fuses the sensor data with the state. Using the notation from Wikipedia[9], the sensor model matrix H is read from a file specified in the UBX node configuration file while the sensor data covariance matrix R is computed on-line from the covariance embedded in the sensor data.

9.3.4 *Fast shm memory access (`control/shm`)*

This general-purpose i-block is able to dynamically allocate vectors of arbitrary length of arbitrary data-types in shared memory. Is is used for fast data passing between c-blocks. Data can be accessed by blocks in the same node by properly linking the port connection.
Note that, if necessary, multiple shm blocks running on different nodes on the same machine can share the same memory pool, however no synchronization mechanism has been implementer to avoid race condition and therefore it has been chosen to grant write permissions only to the block that allocated the pool while other blocks do only have read access.

9.3.5 *AHRS+ sensor data acquisition (`control/sensor_myAHRS`)*

This i-block acts as bridge between the node and the physical Hardkernel AHRS+ sensor. It is worth to point out that the official library from the sensor manufacturer can not be used with the `extern "C"` directive that is indeed necessary to run C++ blocks in UBX, so a wrapper has been created.

9.3.6 *GUI (`copter/copter_graphic_dump`)*

Even if there is a dump of the state of the controller to the terminal, a graphical output is present too. This is a c-block that when triggered reads the state and the set-point of the vehicle to generate and display a 2-D output. The block use the OpenCV library for the rendering. The block obviously refuse to start out of a graphic environment.
The output produced is a view of the vehicle from the top, the size and the center of the environment in wich the copter is moving can be adjusted, the current copter state and set-point is rendered respectively as a red and a green circle with a small tooth to visualize the heading. More detailed information are showed in text format on the top of the screen.

9.3.7 *Set-point generation (`copter/prim_trj_gen`)*

The set point of the vehicle is generated by this block from 10 primitives trajectories. The trajectory code is read from a TCP socked, then the primitive trajectory is scaled on the basis of configurable parameters both in time and space and generated as a function of time. The trajectory primitive are, by now, to translate along the 4 cardinal directions, move up and down and rotate by 90 degrees clockwise or anti-clockwise.
The code of the primitive is passed via a TCP connection by a GUI application described in Section 9.4, the connection is handled using the ZMQ[12] library.

9.4 VIRTUAL REMOTE CONTROLLER

This GUI application act as remote controller for the copter, however it does not directly generates the set-point for the vehicle, which by the way may be unfeasible because of timing constraints, but instead instructs the trajectory generation block about which pre-defined trajectory it should generate. This virtual remote controller is named "Fake Planner" because it can (should!) be replaced by a software planner (or supervisor) capable of decision taking.

10 SIMULATION

Since a real copter was not available for testing, 2 proof-of-concept simulation were performed. In the first one the controller has been used with an extended Kalman filter but no sensor has been used: the state of the copter is estimated by mean of the control signal and the model of the copter; the second simulation demonstrates the feasibility of a full Kalman filter update and predict behavior: an real AHRS sensor is used to update the state of the copter accordingly to the sensors data and a constant control signal.

10.1 KALMAN FILTER IN PREDICT-ONLY MODE

The node core functionality is implemented in the extended Kalman filter and in the controller, a layout of the node is depicted in Figure 3. The set-point of the copter is computed by the node on the basis of 10 primitives that can be called via a tool that acts as remote controller for the copter, the communication between the node and the controller is realized via a TCP connection.
The controller block `ctrl` reads the copter state from the block `state` and the set-point from the block `setpoint` , than computes and store the control action in `ctrl_action`. The Kalman filter block `kf_pred_cptr` in turns reads the state and the control action applied and write the estimate of the state back.

Other blocks are however necessary:

- `ptrig1` is the i-block in charge to call all the c-block of the node (except the graphic dump) in an infinite loop with a period of 1ms;
- `tgj_gen` is a c-block responsible for generating the current set-point: it receives the trajectory primitive from a tcp connection and computes the actual copter set-point as a function of the time;
- `state` is a i-block that store the state of the copter in virtual memory;
- `setpoint` is a i-block that store the set-point of the copter in virtual memory;
- `ctrl_action` is a i-block that store the current applied control action in virtual memory;

- `timing` generates a virtual clock allowing to simulate the system beyond the actual computation capabilities of the computer platform.
- `graphic_dump` opens a gtk 2-D graphic interface showing the position of the copter; a screenshot is reported in Figure 4;
- `ptrig_graphic` trigs `graph_dump` with a period of 10 ms;

The configuration file of the node is here reported:

```
local bd = require("blockdiagram")

return bd.system {
  imports = {
    "types/stdtypes/stdtypes.so",
    "types/control/control_types.so",
    "blocks/ptrig/ptrig.so",
    "blocks/control/shm.so",
    "blocks/control/dump_ctrl.so",
    "blocks/copter/ctrl_copter.so",
    "blocks/copter/prim_trj_gen.so",
    "blocks/copter/kf_predict_copter.so",
    "blocks/control/dump_state.so",
    "blocks/copter/copter_graphic_dump.so",
    "blocks/control/dump_time_stat.so",
    "blocks/webif/webif.so",
  },

  blocks = {
    { name = "ptrig1" , type = "std_triggers/ptrig" },
    { name = "ptrig_graphic", type = "std_triggers/ptrig" },
    { name = "ctrl_action"  , type = "control/shm" },
    { name = "state"  , type = "control/shm" },
    { name = "setpoint"  , type = "control/shm" },
    { name = "ctrl" , type = "copter/ctrl_copter" },
    { name = "ctrl_dump" , type = "control/dump_ctrl" },
    { name = "state_dump" , type = "control/dump_state" },
    { name = "kf_pred_cptr" , type = "copter/kf_predict_copter" },
    { name = "tgj_gen" , type = "copter/prim_trj_gen" },
    { name = "graph_dump"  , type = "copter/copter_graphic_dump"    },
    { name = "timing"  , type = "control/dump_time_stat"    },
    { name = "webif1"  , type = "webif/webif"    },
  },

  connections = {
    { src="ctrl.ctrl_action" , tgt="ctrl_action" },
    { src="ctrl.ctrl_action" , tgt="ctrl_dump" },
    { src="state" , tgt="ctrl.state" },
    { src="setpoint" , tgt="ctrl.setpoint" },
    { src="kf_pred_cptr.stt_out" , tgt="state" },
    { src="kf_pred_cptr.stt_out" , tgt="state_dump" },
    { src="state" , tgt="kf_pred_cptr.stt_in"  },
    { src="ctrl_action" , tgt="kf_pred_cptr.ctrl_in" },
    { src="tgj_gen.setpoint_out"  , tgt="setpoint" },
    { src="state" , tgt="tgj_gen.state_in" },
    { src="state"      , tgt="graph_dump.state"  },
    { src="setpoint"    , tgt="graph_dump.setpoint"  },
  },

  configurations = {
    {
      name="webif1",
      config = { port="8888" },
    },
```

```
        {
                name="kf_pred_cptr" ,
                config = {
                        mass            = 1     ,
                        J_xx            = 0.01,
                        J_yy            = 0.01,
                        J_zz            = 0.01,
                        dt              = 0.001 ,
                        process_noise   = 0     ,
                },
        },
        {
                name="ctrl_action" ,
                config = {
                        shm_name="/ctrl_action" ,
                        type_name="struct ControlAction",
                },
        },
        {
                name="setpoint" ,
                config = {
                        shm_name="/setpoint" ,
                        type_name="struct SetPoint",
                },
        },
        {
                name="state" ,
                config = {
                        shm_name="/state" ,
                        type_name="struct State",
                },
        },
        {
                name="tgj_gen" ,
                config = {
                        dt      = 0.001 ;
                        zmq_address = "tcp://127.0.0.1:8890"
                },
        },
        {
                name="ctrl" ,
                config = {
                        mass            = 1     ,
                        inertia_xx      = 0.01,
                        inertia_yy      = 0.01,
                        inertia_zz      = 0.01,
                        k1              = 1     ,--0.9  ,
                        l1              = 5     ,--5    ,       -- i.e.
                             max speed for static set-point
                        k2              = 10    ,--8    ,
                        l2              = 10    ,--8    ,
                        kp              = 100   ,
                        kd              = 10    ,
                        hysteresis_thr  = 0.01  ,
                        max_thrust      = 20    ,
                        max_torque_xx   = 3     ,
                        max_torque_yy   = 3     ,
                        max_torque_zz   = 3     ,
                },
        },
        {
                name="timing" ,
                config = {
                        period          = 60*1000       ;
                        update_rate     = 0.001         ;
                },
```

```
                },
                {
                        name="ptrig1" ,
                        config = {
                                period = {sec=0, usec=1000 },
                                trig_blocks={
                                        { b="#ctrl"             , num_steps=1,
                                             measure=0 },
                                        { b="#kf_pred_cptr"     , num_steps=1,
                                             measure=0 },
                                        { b="#tgj_gen"          , num_steps=1,
                                             measure=0 },
                                        { b="#timing"           , num_steps=1,
                                             measure=0 },
                                },
                        },
                },
                {
                        name="graph_dump" ,
                        config = {
                                zoom = 5 ;
                                max_q = 2 ;
                        },
                },
                {
                        name="ptrig_graphic" ,
                        config = {
                                period = {sec=0, usec=10000 },
                                trig_blocks={
                                        { b="#graph_dump"       , num_steps=1,
                                             measure=0 },
        },
      },
    },
  },
}
```

10.1.1 *Launching the node*

The node can be lunched by calling the script `ubx-control/test/04-working_ctrl/launch.sh` This script also executes the gtk app `fake_planner` specifying with an argument to connect to `tcp://127.0.0.1:8890` (obviously the node is configured to listen at the same TCP address and port). Then the script launches and deploys the node. Messages from the node are dumped to the terminal but the state of the node can be also checked via a web interface at `http://127.0.0.1:8888`.

The name of the remote controller executable is `fake_planner` because there may be an actual mission *planner (or supervisor) controlling the drone instead of a human operator.*

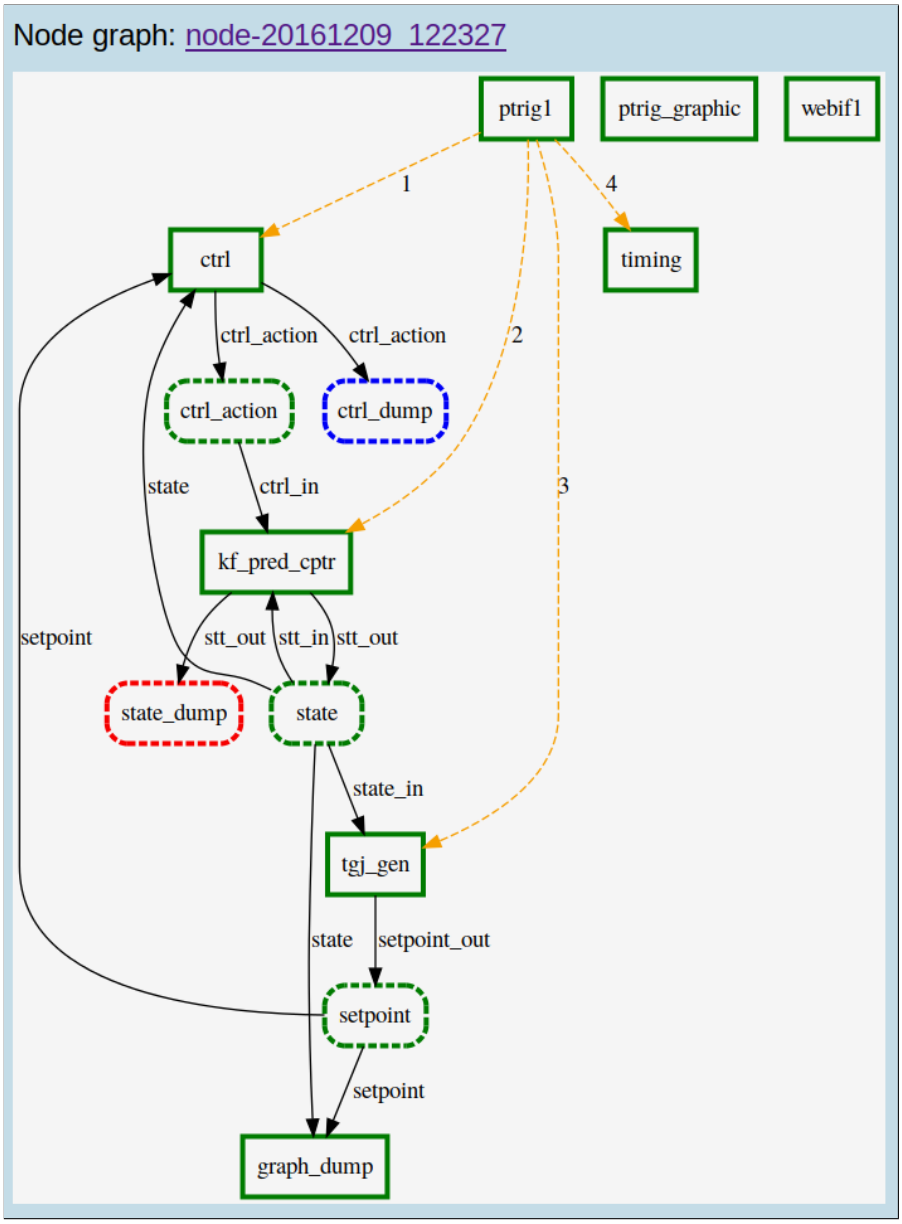


Figure 3: The node used for the simulation in predict-only mode

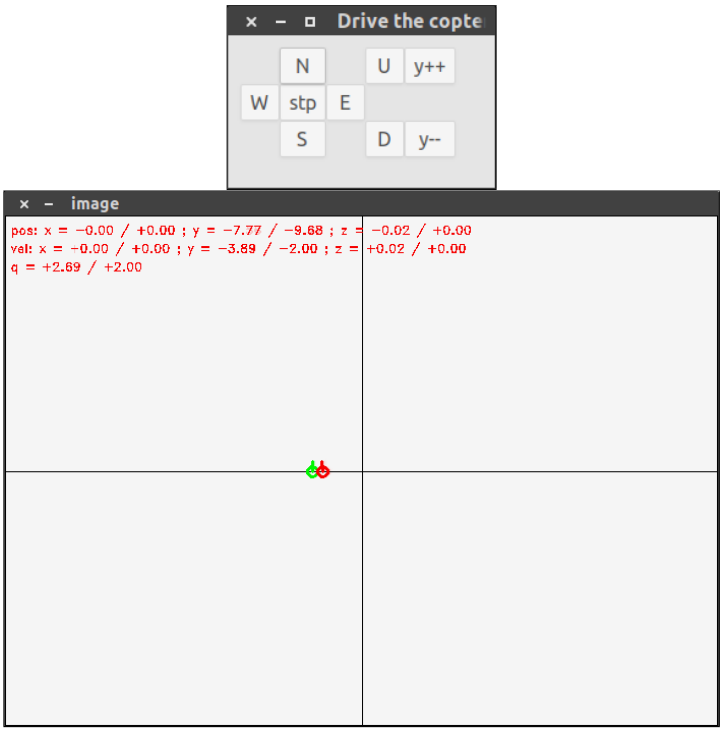


Figure 4: The remote controller HMI interface and the graphic dump of the copter state and set-point

10.2 KALMAN FILTER IN UPDATE MODE

The second simulation was performed in a configuration where the Kalman filter is used to update the estimate of the state of the vehicle by reading from an attitude and heading reference system (AHRS) [1]. This application is a proof-of-concept demonstration for sensor fusion and the controller has no role here, the control action is constant (i.e. no feedback) and manually tuned to make the copter hover. The Kalman filter then update the copter attitude with the data from the sensors and accordingly predict the estimate of the position too.
The AHRS used is a USB device from Hardkernel [4]; it embeds a triple axis gyroscope, a triple axis accelerometer and a triple axis magnetometer, plus it embeds an extended Kalman filter (so data if pre-filtered and pre-fused); data can be read via a I2C or UART or USB serial interface. The i-block used to read is a bridge written in C++ named `sensor_myAHRS` and can be found in the `ubx-control` project.

The node can be launched by executing:

```
cd ubx-control/test/06_test_myAHRS/
ubx_launch -c 06.usc -d deploy.udc
```

As expected, if the AHRS is horizontal the Kalman filter will just correct the heading: since the AHRS embeds a magnetometer, the filter has been told to map the x axis of the fixed reference frame the copter is moving in to the magnetic North.
If the AHRS is oriented, the copter will orient accordingly and will start moving around. Note that however the copter will loose elevation, since the controller generates no torque and just the thrust exactly sufficient to hover horizontally.

As in the previous simulation, it follows the configuration of the node:

```
local bd = require("blockdiagram")

return bd.system {
  imports = {
    "types/stdtypes/stdtypes.so",
    "types/control/control_types.so",
    "blocks/ptrig/ptrig.so",
    "blocks/webif/webif.so",
    "blocks/control/shm.so",
    "blocks/control/dump_ctrl.so",
    --"blocks/copter/ctrl_copter.so",
    "blocks/copter/prim_trj_gen.so",
    "blocks/copter/kf_predict_copter.so",
    "blocks/control/dump_state.so",
    "blocks/copter/copter_graphic_dump.so",
    "blocks/control/dump_time_stat.so",
    "blocks/control/sensor_myAHRS.so",
    "blocks/control/fake_ctrl.so",
    "blocks/control/kf_update.so",
  },
```

```
blocks = {
  { name = "ptrig1"   , type = "std_triggers/ptrig"           },
  { name = "webif1"   , type = "webif/webif"                          },
  { name = "ptrig_graphic", type = "std_triggers/ptrig"       },
      { name = "ctrl_action"  , type = "control/shm"                  },
      { name = "state"        , type = "control/shm"                          },
      { name = "setpoint"     , type = "control/shm"                          },
      { name = "ctrl"         , type = "control/fake_ctrl"            },
      { name = "ctrl_dump"    , type = "control/dump_ctrl"                    },
      { name = "state_dump"   , type = "control/dump_state"                   },
      { name = "kf_pred_cptr" , type = "copter/kf_predict_copter"             },
      { name = "tgj_gen"      , type = "copter/prim_trj_gen"
          },
      { name = "graph_dump"   , type = "copter/copter_graphic_dump"   },
      { name = "timing"       , type = "control/dump_time_stat"               },
      { name = "ahrs"       , type = "control/sensor_myAHRS"          },
      { name = "kf_upd"       , type = "control/kf_update"            },
},

connections = {
      { src="ctrl.control_out"        , tgt="ctrl_action"                   },
      { src="ctrl.control_out"        , tgt="ctrl_dump"                     },
      { src="kf_pred_cptr.stt_out"    , tgt="state"                   },
      { src="kf_pred_cptr.stt_out"    , tgt="state_dump"              },
      { src="state"                   , tgt="kf_pred_cptr.stt_in"             },
      { src="ctrl_action"             , tgt="kf_pred_cptr.ctrl_in"    },
      { src="tgj_gen.setpoint_out"    , tgt="setpoint"                },
      { src="state"                   ,tgt="tgj_gen.state_in"                 },
      { src="state"                   , tgt="graph_dump.state"                },
      { src="setpoint"                , tgt="graph_dump.setpoint"             },
      { src="kf_upd.state_out"        , tgt="state_dump"              },
      { src="kf_upd.state_out"        , tgt="state"                   },
      { src="state"                   , tgt="kf_upd.state_in"         },
      { src="ahrs"                     , tgt="kf_upd.sensor_in"       },
},

configurations = {
      {
              name="webif1",
              config = { port="8888" },
      },
      {
              name="kf_pred_cptr" ,
              config = {
                      mass            = 1     ,
                      J_xx            = 0.01,
                      J_yy            = 0.01,
                      J_zz            = 0.01,
                      dt              = 0.001 ,
                      process_noise   = 0.01  ,
              },
      },
{
      name="ctrl_action" ,
      config = {
              shm_name="/ctrl_action" ,
              type_name="struct ControlAction",
  },
},
{
  name="setpoint" ,
  config = {
    shm_name="/setpoint" ,
    type_name="struct SetPoint",
  },
},
```

```
{
  name="state" ,
  config = {
    shm_name="/state" ,
    type_name="struct State",
  },
},
{
  name="tgj_gen" ,
  config = {
    dt        = 0.001 ;
  },
},
{
  name="ctrl" ,
  config = {
    filepath_U="U",
  },
},
{
  name="ahrs",
  config = {
    device_path = "/dev/ttyACM0"      ,
    quaternion_cov = 0.0001           ,
  }
},
{
  name="kf_upd" ,
  config = {
    filepath_H        = "H"   ,
  },
},
{
  name="timing" ,
  config = {
    period            = 60*1000       ,
    update_rate       = 0.001         ,
  },
},
{
        name="ptrig1" ,
        config = {
          period = {sec=0, usec=1000 },
          trig_blocks={
          { b="#ctrl"         , num_steps=1, measure=0 },
          { b="#kf_upd"               , num_steps=1, measure=0 },
          { b="#kf_pred_cptr" , num_steps=1, measure=0 },
          { b="#tgj_gen"              , num_steps=1, measure=0 },
          { b="#timing"               , num_steps=1, measure=0 },
        },
      },

    },
    {
      name="graph_dump" ,
      config = {
      zoom = 5 ;
      max_q = 2 ;
    },

  },
  {
    name="ptrig_graphic" ,
    config = {
      period = {sec=0, usec=1000 },
      trig_blocks={
```

```
          { b="#graph_dump"     , num_steps=1, measure=0 },
        },
      },
    },
  },
}
```

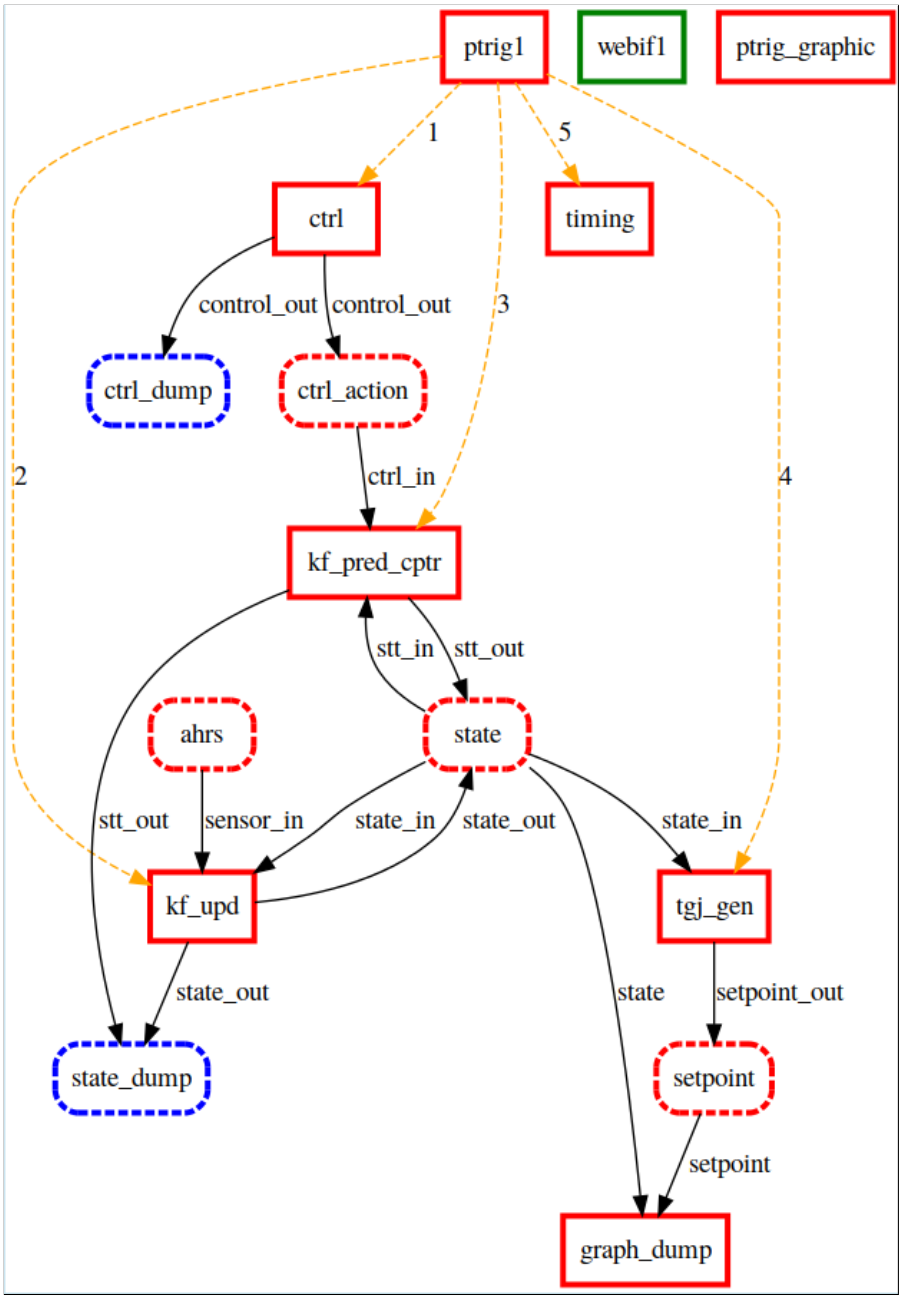


Figure 5: The node used for the simulation in update mode

Part IV

CONCLUSIONS

11 ARCHIVED RESULT

This work demonstrates how a component blocks application can be developed in UBX. Obviously, this framework is not as mature as other projects, but introduces some new concepts not present in other environments.
The whole application works as expected and can easily be ported to other platform: UBX blocks can be compiled for many CPU architectures: AMD64 and ARM has been tested but it should be possible to compile for microcontrollers too.
The time constraints can be easily respected in cheap (but modern) credit-card-sized computer platforms, perhaps a more advanced trigger s-block is necessary.
Moreover within the context of this thesis some work has been done to deploy a UBX node on the Orocos environment: first, the Ororcos OCL source code has been forked and linked against LuaJIT instead of the native LUA library to allow the usage of the FFI library, the forked repo can be found at:
`https://github.com/fmr42/ocl`
Then the trigger block has been modified to run when triggered by Orocos instead than on a time base. Than the two simulations proposed were launched in this new deploy environment and, as expected, they run as before; the only difference was that the timing for the loop come from the more advance real-time Orocos framework. Also, Linux cgroups where tuned to guarantee the Orocos *hard* real-time constrains.
Please note that this last promising result is part of a work still under development, however the Orocos team seamed to appreciate the work done in the fork.

12 FURTHER WORK

12.1 UBX LIMITATIONS

The UBX framework allows to write blocks with well defined separation of concerns, but does not actually *require* it: a coder could, for example, make a c-block store data in its private memory space, or make a i-block process the data with a computational behavior.
Moreover another limitation of UBX is that it still does not support blocks composition, while it should be possible to define sub-nodes to be used as regular blocks.
A new library named AB5C is in a very early developing stage and should replace UBX and solve those problematics.

12.2 MISSING MODELS

There still is a lack of modeling in this work, also because of the lack of tools to do so. More techniques are under development, especially for data; the goal is to be have (someday) developing frameworks ables to auto-generate the code from models; although there already are tool-chains able to declare (not define!) functions from models, the road to tools able to implement components from mathematical models is still long. By now, the more feasible goal is to have a centralized repository of data models (e.g. positions, rotations, ecc.) to conforms to and at least have data-model transformation tool able to generate "adapter" components to trivially connect components not developed for being connected.

12.3 OPEN-SOURCE CONTRIBUTIONS

As a personal opinion, a large users base is essential to reach high goals and the open source software developing model has proved to work once reached a "critical mass" of developers; as an example, take the ROS project: an environment very similar to UBX that is leading to success thank to a wide adoption in the robotic community.
Moreover, universities represent a great source of work force, especially for the complex job of modeling, and should take the initiative in order to tweak a wider (industrial, scientific, recreational, commercial) adoption.

BIBLIOGRAPHY

[1] *Attitude and heading reference system*. Dec. 12, 2016. URL: https://en.wikipedia.org/wiki/Attitude_and_heading_reference_system.

[2] *Author pull request to the UBX team*. Dec. 12, 2016. URL: https://github.com/UbxTeam/microblx/pull/5.

[3] *FFI Library*. Dec. 12, 2016. URL: http://luajit.org/ext_ffi.html.

[4] *Hardkernel myAHRS+ product informations*. Dec. 12, 2016. URL: http://www.hardkernel.com/.

[5] N. Hochgeschwender G. Kraetzschmar L. Gherardi D. Brugali M. Klotzbücher H. Bruyninckx. "The BRICS Component Model: A Model-Based Development Paradigm For Complex Robotics Software Systems." In: ().

[6] R. Naldi M. Furci R. G. Sanfelice L. Marconi. "Global Trajectory Tracking for Underactuated VTOL Aerial Vehicles using a Cascade Control Paradigm." In: ().

[7] M.D. Shuster. "A survey of attitude representation." In: *The Journal of the astronautical sciences* 41.4 (Dec. 1993), 439–517.

[8] *UBX git repository*. Dec. 12, 2016. URL: https://github.com/UbxTeam/microblx.

[9] *Wikipedia Kalman filter page*. Dec. 12, 2016. URL: https://en.wikipedia.org/wiki/Kalman_filter.

[10] *Wikipedia "Quaternion" page*. Dec. 12, 2016. URL: https://en.wikipedia.org/wiki/Quaternion.

[11] *Wikipedia "Quaternions and spatial rotation" page*. Dec. 12, 2016. URL: https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation.

[12] *ZMQ Library*. Dec. 12, 2016. URL: http://zeromq.org/intro:read-the-manual.