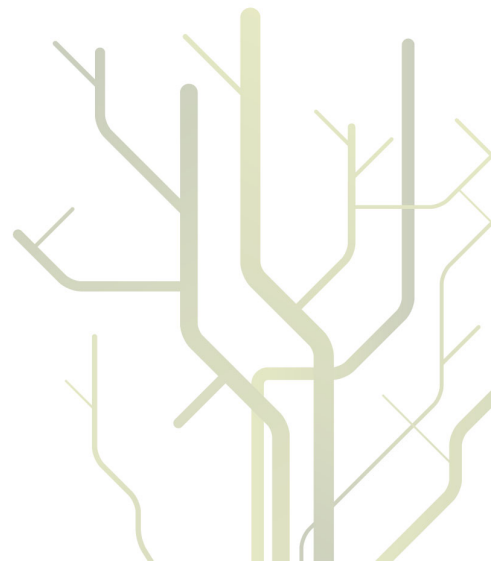


A configuration tool for process oriented UAV programming



Ørjan Pettersen

INF-3990
Master's Thesis in Computer Science
June, 2010



Abstract

This thesis covers the design, implementation and evaluation of a configuration tool for process oriented Unmanned Aerial Vehicle (UAV) programming. In addition it will examine if and how a process network can be used to control sensors and communication channels on an UAV in flight.

NORUT-IT is currently developing a sensor platform based on UAVs. The mission computer software they have at the moment have room for improvement when it comes down to issues regarding routing and prioritizing between available network connections. One issue that has been identified is not being able to route between multiple available networks. They have to predefine which network connection the UAV should use to connect to the ground station when it is in flight. It will use this connection the entire flight even if a faster and/or cheaper network connection is available in some areas of the mission.

Together with this issue, having a platform with a number of sensors working together, will be challenging to configure and might require programming skills to some degree to set up correctly.

Communicating sequential processing(CSP) have properties that can help in building concurrent, reliable and scalable programs. By using CSP and a process configuration tool, the complexity of configuring the mission computer of an UAV can be reduced.

The implementation will demonstrate a tool that are believed to be intuitive and will lower the challenge of configuring a process network intended to control the sensors and communication channels on an UAV.

The process network creator tool have a graphical interface and a collection of premade CSP processes. It will also have the ability to convert the graphical representation of the process network into a running CSP process.

Acknowledgements

I would like to thank my supervisor John Markus Bjørndalen for his inputs and help in general. I would also like to thank my fiance Janne Arntsen for her support during this time. In the end I would like to thank to my kitten, Daisy, for her numerous attempts to create beautiful code when I was asleep. I guess the intention was good, but the result not so much.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Problem definition	2
1.3	Outline	2
2	Related work	3
2.1	CSPBuilder	3
2.2	LabVIEW	4
2.3	Robolab and NXT-G	5
2.4	CSP	5
2.4.1	PyCSP	5
3	Design	9
3.1	The overall system design.	9
3.2	UAVBuilder design	10
3.3	Module design	10
3.3.1	The header part	12
3.3.2	The process part	13
3.4	Process network design	14
3.5	Connection prioritizing	15
3.6	Data filtering	15
4	The visual tool	17
4.1	Graphical user interface	17
4.1.1	wxPython	18
4.1.2	Advanced User Interface	19
4.1.2.1	Issue	19
4.2	UAVBuilder	20
4.2.1	Drag'n drop support	21
4.2.2	Load/Save	21
4.2.3	Process builder	22
4.2.4	Gui preferences	22
4.3	Module library	22
4.4	Building process network	25

5	Test phase	27
5.1	Testing the UAVBuilder	27
5.1.1	Issues discovered	27
5.2	Testing the CSP network	27
5.2.1	First test environment	28
5.2.2	Second test environment	28
5.2.3	Connection prioritizing	28
5.2.4	Data filtering	29
5.3	The Paparazzi project	29
6	Discussion	31
6.1	The configuration tool	31
6.1.1	Future work	31
6.2	Connecting prioritizing	32
A	DVD-ROM	35
B	Userguide	37
B.1	Introduction	39
B.2	Installation	39
B.2.1	Dependencies	39
B.3	UAVBuilder	40
B.3.1	Usage	40
B.3.1.1	Delete module	41
B.3.1.2	New connection	41
B.3.1.3	Delete channel	41
B.3.1.4	Build module	42
B.3.2	Menu	42
B.3.3	Create custom module	43
B.3.3.1	Configuring modules	44
B.3.4	Preferences	46
B.3.5	Building process network	46
B.3.5.1	Running the process network	47
B.3.5.2	Stop the process network	48
B.4	Example source	48

List of Figures

2.1	PyCSP process and channel.	6
3.1	Basic work flow.	9
3.2	UAVBuilder	10
3.3	Module design	11
3.4	Fig. (a) shows a process with one channel connected to three receivers. Fig. (b) shows a process using three channels connected to three receivers.	14
3.5	Process network design	15
3.6	Data filtering	16
4.1	Hello World window in Debian running Gnome and Windows XP.	18
4.2	Simple three panel layout using the AUI library compared with using a window splitting approach.	19
4.3	The configuration tool layout.	21
4.4	Module panel tree	24
4.5	Configuration panel demo	24
B.1	UAVBuilder layout.	40
B.2	Right click menu.	41
B.3	Delete channel.	42
B.4	Module editor.	43
B.5	Configuration panel demo.	45
B.6	Preferences window.	46
B.7	Process network example.	47

Chapter 1

Introduction

The use of Unmanned Aerial Vehicles (UAVs) has grown substantially last decade. Not only in military operations, but also in civilian use. Especially within the field of research, they are used for surveillance, monitoring and remote sensing.

These UAVs often use either publicly available open source or commercial available auto pilots to fly autonomously. In addition to the auto pilot, a mission computer is often also present in the UAV. This computer can control the various sensors on board the plane and store data gathered by the sensors. It can also be connected to the auto pilot making it possible to alter the mission plan if necessary.

In order to communicate with the UAV, it usually have one or more types of modem on board. Communication can take place over GSM/3G, Iridium(satellite) and/or radio each with a different set of speed and connection cost.

NORUT-IT is currently developing a sensor platform based on UAVs. During the development and testing, they have come across a problem regarding how to control, route and prioritize communication based on the currently available communication channels. While the plane is in the air it will move between areas where there are different communication channels available. A satellite connection will usually be operative most of the mission, but it is slow and have a high communication cost. In other areas there might be a GSM connection available, and close to the launch site there might be a wireless network available, giving a fast and cost free connection.

The current system running on the mission computer does not support changing communication channel when the UAVs flying. It is limited to using the communication channel that is almost certain to give them a connection to the degree the mission requires. Even if there are an alternative connection in parts of the mission area that is faster and have lower communication cost than the default one, the UAV is limited to using the chosen channel.

Another area that will pose a challenge is to find techniques that will try to reduce communication cost and keeping real time data up to date at the ground. To be able to prioritize certain types of important flight data and other time

critical data on behalf of lower priority sensor data. This by applying filters to the sensor data stream, throttling down or redirect the data stream from for instance a network connection to storage.

1.1 Motivation

Given the complexity of the mission sensor control software, creating a reliable concurrent system will pose a challenge for the scientist with little or no programming experience as well as the experienced programmer.

After seeing how easy children adapt to the graphical way of programming Lego Mindstorm[1] robots, a similar tool for creating software to control mission sensors in the UAV, together with the use of Communication Sequential Processing (CSP)[3], would hopefully reduce the complexity of making a UAV ready for flight.

1.2 Problem definition

This thesis will focus on developing a graphical tool for creating CSP networks, and to provide an easy way to configure the network without the need for programming experience. Support for transforming this graphical representation of the network into a running process network will also be implemented.

Alongside this tool, the thesis will examine how process oriented programs can be used to deal with the issue of dynamically routing between available network connections. It will also be examined how the process network can be used to reconfigure sensors and communication channels, together with how the sensor data stream could be throttled down or redirected, in order to reduce the amount of data that needs to be sent to the ground.

1.3 Outline

The rest of the thesis is organized as follows. In chapter two it will be given an overview of related applications that in some way have had an influence on the development of the application and the CSP network. Chapter three will cover the design choices made during the development process. Chapter four will look closer at the implementation of the configuration tool and issues that came up. Chapter five will describe the testing that was done. Chapter six will discuss the findings. Chapter seven give an outline of further work.

Chapter 2

Related work

This thesis consist of two parts. First the making of the graphical tool that is used to create the CSP network, then the creation of the CSP processes that is used in the graphical tool to build the network itself. So, this chapter will have two parts. The first one will present some related real world applications and research that is concentrated around graphical flow based tools, the second part will present some CSP related research.

2.1 CSPBuilder

CSPBuilder[4][5] is created by Rune Møllegård Friberg and Brian Vinter at the University of Copenhagen. It is a visual tool created in Python that is used to model a CSP network. The user add processes to a canvas and connect channels between the processes to create a process network. The processes appear on the canvas as named boxes, and the connections as lines between the processes.

The process network is built up by components. A component can be a single process or an already made process network. that is connected to the new network. The components does not contain any code, but a link to the file with the code. The components are arranged in different libraries in order to make it easier for the user to find the desired component.

They have included a component building wizard to make it easy for the user to create additional components. This to enable the user to take advantage of the huge amount of scientific code already available. The wizard have support for importing code written in Python, of course, but also C and C++ trough SWIG, and FORTRAN 77 and 90 through F2PY.

Configuration of the components is also possible. Each component that wants to be configurable must have a `setup()` function, where it is defined what is configurable and how this is being done, i.e trough a text dialog or by opening a file.

From the representation of the process network in the tool and the configuration data, a data structure is saved in a `.csp` file. This file is then converted

to a structure resembling a CSP network and executed by the program.

This application is the one resembling UAVBuilder the most. Not just in name, but also in what it does and to some extent how it looks, and could probably be used to solve some of the issues of this thesis. But for three reasons it was chosen not to.

The CSPBuilder have a two panel layout with the configuration in a separate dialogue. By observing non programmers programming Lego robots[1] using a three panel layout with a configuration panel easy accessible at the bottom, and a list of modules in the left panel, that solution looked more intuitive and easier to use. By creating a new GUI with a split window solution for creating both an air and a ground network simultaneously, could be implemented.

Second, the CSPBuilder uses a multiple file arrangement for the modules/-components. A single file solution where the python code and the module/-component information are in the same file would hopefully make importing, exporting and moving the modules/components between installations/UAVs less complicated.

Third, the system for configuring a module/component is different. CSPBuilder uses a separate setup function where the configuration entries and the configuration interface is defined. This means that the creator of the module/-component need to some extent know how to use the wxPython[18] library.

In UAVBuilder a separate process handles the configuration data. This is a solution to chosen to make it possible to change the configuration of the process network from the ground when the UAV are in flight. This process is also used to handle the configuration parameters entered before flight in the UAVBuilder.

2.2 LabVIEW

LabVIEW[6] is a graphical programming environment created by National Instruments. It was originally released for the Apple Macintosh in 1986, but is now available on a variety of platforms like Microsoft Windows, Unix, GNU/Linux and Mac OS X.

Developed to give scientists and engineers an intuitive tool for creating applications for data acquisition, instrument control, and industrial automation. It is using a flowchart resembling environment to program in. The user places graphical function-nodes and connect wires between the nodes. The graphical programming language is called G. The latest version at the time being is LabVIEW 2009, released in August 2009.

Since this is not using CSP to control the notes, it could not be used in this thesis. But being one of the most widely used tools for graphical programming, there was a few visual tips to pick up.

2.3 Robolab and NXT-G

They are both tools for programming Lego robots, and both based upon LabVIEW and therefor resembles it to some degree in look and feel.

Robolab[7] is the oldest one. It is at version 2.5 and provides a graphical programming environment to the old RCX brick from Lego. It has a progressive programming phase in three stages where the programmers skills can be matched.

NXT-G[1] is the next generation Lego programming tool. It is used to program the NXT brick, but also have support for the older RCX brick. It has a GUI with mainly three parts. A collection of programming elements to the left. The programming area in the middle and a configuration area in the bottom.

They are both made specifically to program the Lego robots, so using these tools to solve the challenges in hand was not possible. But the user interface is built up very intuitive, making it easy for non programmers to adapt to the graphical way of programming. With this in mind, the graphical user interface of UAVBuilder was inspired especially by NXT-G, being the most modern of the two.

2.4 CSP

Communicating Sequential Processes(CSP) is a formal language that was first presented in a paper by C.A.R Hoare[2] in 1978. It uses mathematical algebra to describe patterns of interaction in concurrent systems.

2.4.1 PyCSP

PyCSP[8] is a python[19] implementation of the basic abstraction of CSP. It is under development by John Marcus Bjørndalen at the University of Tromsø, and Brian Vinter and Rune Møllegård Friberg at the University of Copenhagen.

It is currently at version 0.6.2. During the time of the thesis this library has evolved from version 0.3.0 to 0.6.2 with a number of significant changes, making version 0.3 not compatible with 0.5 and later. The biggest visible change for the developer, is the number of channel types available.

A small example showing this is found inn Listing 2.1 and 2.2. The example is pretty basic. Two processes, one consumer and one producer. The producer sends a counting value to the consumer, which prints it to the screen.

Listing 2.1: PyCSP code pre v0.5

```

import time
from pycsp import *

@process
def consumer(cin):
    while True:
        print cin()

@process
def producer(cout):
    i = 0
    while True:
        cout(i)
        i += 1

chan1 = One2OneChannel()
Parallel(consumer(chan1.read), producer(chan1.write))

```

Listing 2.2: PyCSP code post v0.5

```

import time
from pycsp import *

@process
def consumer(cin):
    while True:
        print cin()

@process
def producer(cout):
    i = 0
    while True:
        cout(i)
        i += 1

chan1 = Channel()
Parallel(consumer(IN(chan1)), producer(OUT(chan1)))

```

PyCSP v0.3 had four different channel types, depending on how many producers and consumers that was connected together. Now there is just one channel type. This change makes it much easier to auto generate the channel and Parallel statement, since finding out what type of channel is needed, is no longer necessary.

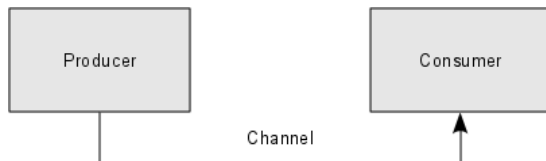


Figure 2.1: PyCSP process and channel.

The example above also show the two most central abstractions of CSP. The process and the channel. A CSP network consists of a number of processes communicating over channels. The processes can be run sequentially with a Sequence construct or, like in this example, in parallel with a Parallel construct.

Chapter 3

Design

This chapter will present the design choices that were made during the development.

3.1 The overall system design.

Figure 3.1 shows the general idea of how the work flow of the overall system is intended to be. A process network is graphically represented and configured in the UAVBuilder. Then it is built to a runnable process network and written to disk.

Since the UAV is highly mobile, it will probably not be available every time a process network is created. So having an automated routine transferring the process network to the mission computer in the UAV, was not an important feature. So all the needed modules and the build file are manually copied in to the mission computer in the UAV.

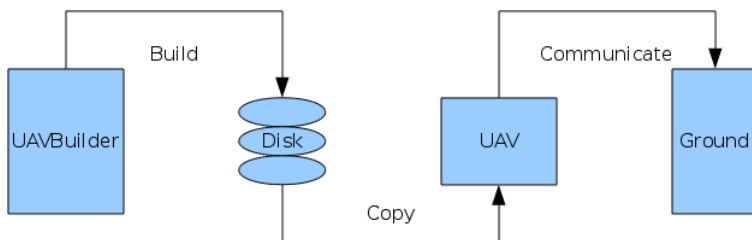


Figure 3.1: Basic work flow.

The build file is then executed when the mission computer is booted in the UAV or at a given point later depending of the initial setup of the UAV. The UAV will then start communication with the ground using normal sockets.

3.2 UAVBuilder design

The design of the UAVBuilder GUI is shown in figure 3.2. A panel with a list of available modules is on the left side. The configuration panel is on the bottom and the canvas where the process network is created is in the middle.

This arrangement was inspired by the NXT-G[1] Lego programming software. By using it and seeing how fast non experienced programmers get familiar with this setup, a similar look where the user have the most used parts of the software visually available, was created.

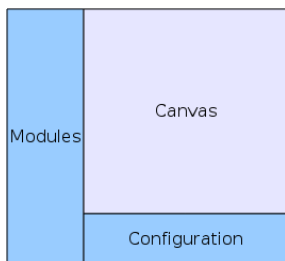


Figure 3.2: UAVBuilder

3.3 Module design

The module library was organized in the following manner. The module configuration and the code for the CSP code was collapsed into one file. Figure 3.3 show the module file, and how the configuration data is on the top and the process part is on the bottom of the file.

The reason for this approach is to make it simple to move between installations. Either as a part of the save file, copied directly between installations or copied from the mission computer in the UAV. Either way all the information needed by UAVBuilder and the process network will be available.

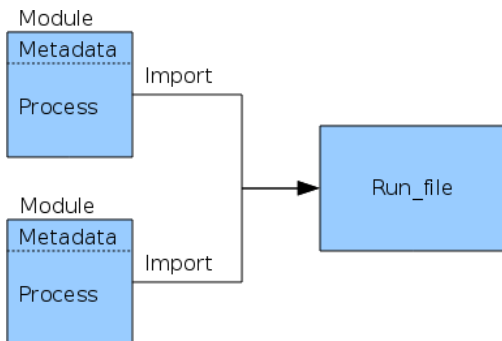


Figure 3.3: Module design

To be able to edit the module file, plain text files were used. Then any preferred text editor in addition to the built-in editor in UAVBuilder can be used to edit them. And by using python files, they can be used directly by importing them into the run file of the process network like figure 3.3 shows.

In order to use the module in UAVBuilder, just copy the file into the module directory, and it will be picked up by the program when it starts and made available in the module tree view.

The module files are organized with two parts, a header containing metadata and the process code. Listing 3.1 shows an example of a module text file. The header contains information needed by UAVBuilder. The rest is the code for the process used when the process network is running. When UAVBuilder reads the file, it only extracts information from the header. It does not use any information from the code part of the file. And the opposite when the process network is running. It only uses the code part of the file, not the header.

Listing 3.1: Module text file

```

""" Simple count module. """

#####
#
# NAME: count
# PACK: Test/Producer
# ICON: images/count.png
# FUNC: count
# INCO: 0
# OUTG: 1
# CONF:
# DESC: Simple count module.
#
#####

import time
from pycsp import *

@process
def count(count):
    name
    data = 0

    while True:
        cout[0]('SENSOR_DATA', (_name, data))
        time.sleep(1)
        data += 1

```

This is done to reduce the complexity of the function extracting information needed by the builder since it does not need to search through the process part of the file to find information. Another reason is that the process code will not be cluttered up with unnecessary information not related directly to the execution of the process.

The module file will never be altered in any way when creating, configuring and running the process network. Configuration made in the builder will be saved to a separate configuration file which is accessed by the configuration process and sent to the process. This to make sure that the file will be just like the developer of the module intended.

3.3.1 The header part

The header shown in listing 3.1 have some fields that was found necessary in order to represent the module in the builder.

- Name: Describes the name of the module. This is used in the builder as an identifier in the tree structure in the module panel. It will also be displayed in the configuration window and as an identifier when deleting connections between modules.
- Pack: This decides how it will be shown in the module tree list. This was included so the creator of the module could decide where the module is found in the module tree list.

- **Icon:** The path to the icon image. The icon is used to easily identify a module on the canvas.
- **Func:** The name of the process function. To tell the process network builder which process to import and use in the build file.
- **Inco/outg:** Defines the number of incoming/outgoing channels the module can handle. 0 for none, 1 for one and 2 for many. By being able to define the numbers of connections the process supports, the complexity of the process can be reduced. This is since the process does not need to be able to handle every possible situation. For instance, it can be created to handle only one channel, and it will be limited by the builder to receive only one connection from another process.
- **Conf:** Defines the configuration parameters to the process. Has four configuration types. The four choices it supports are static text, a string box, a multiple choices list and a list of mutually exclusive choices. These four types were implemented because I believe they will cover most needs. This field might be blank if the process does not have any configuration.
- **Desc:** A description that describes the module. Is shown in the status line when highlighting the module in the module tree list and in configuration window when it is focused on the canvas.

3.3.2 The process part

The process part uses standard python/pycsp syntax. See listing 2.2 for a closer look at what the syntax looks like. I however have made a small change to how the parameter to the process are handled.

Using a single channel as a parameter to the process like in figure 3.4a, with possible multiple receivers/senders in the other end, have their applications in some situations. For instance, when working with receiver processes where the order or the content of the data received by the process, does not matter.

But if the data needs to be targeted to a specific process, like in our case with many specific processes controlling sensors, this one-to-many approach is not very efficient. When using a single channel to send data to multiple receivers, you get a first-come, first-served situation, where you cannot know who will be served first. So the information needs to be sent at least as many times as there are receivers, and even then you cannot be absolutely sure the correct process have got the data since one process could have got the data twice.

So instead of using a single channel as a parameter to the process, a list of channels was used instead. Each channel has a single receiver/sender connected like figure 3.4b shows. By doing it like this it is possible to create routing processes that target the data to a specific process instead of a number of processes. This is useful when sending configuration data to sensor processes since it is easy to keep track of where to send the data and when the process have received it.

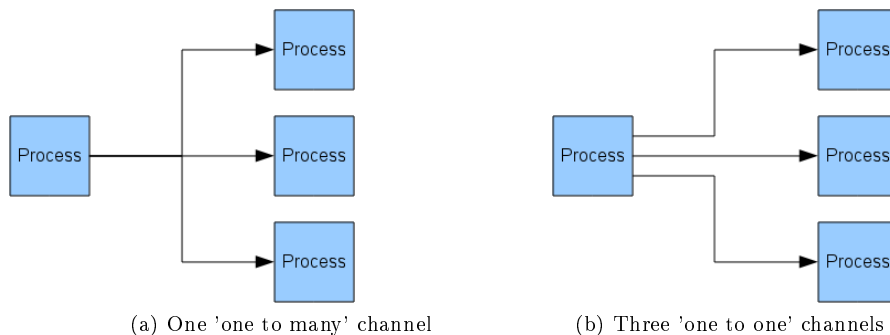


Figure 3.4: Fig. (a) shows a process with one channel connected to three receivers. Fig. (b) shows a process using three channels connected to three receivers.

This is basically only an issue when sending data from one process to many receivers. Not so much when there are many processes sending data to one receiver, since the data easily can be marked with the senders identifier. But for simplicity and consistency I have used the same approach in every situation. I think it is easier to relate to when creating modules that you always have to handle a list of channels.

3.4 Process network design

An example process network is found in figure 3.5. The sensors processes are connected to a demultiplexer. The demultiplexer will send the data either to the communication process or to a storage process depending of the available connection. The communication process is connected to a process network on the ground. The communication between the UAV and the ground uses network sockets.

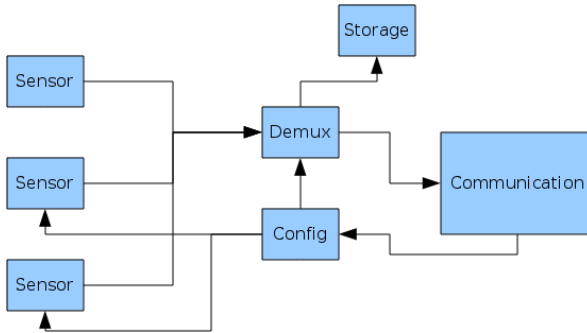


Figure 3.5: Process network design

One advantage with the modular design is that the end user easily can add more processes between the sensors and the communication process. But data, for instance configuration data from the ground to a sensor, would have to travel the in the opposite direction compared to the sensor data flow. This means that each process also had to support sending data the other way, adding unwanted complexity and higher workload to each process.

This is solved by having a separate configuration process connected to the different processes in need of configuration data. This will hopefully reduce the complexity of the other processes.

3.5 Connection prioritizing

Based on the mission, one or more connection channels between the UAV and the ground will be prepared. In flight, the communication process in the UAV will check if any of these channels are available, and send data over the fastest and least expensive connection.

This is solved by using a prioritized list over the channels prepared. To check if any of the channels are available, the communication process simply tries to connect to them periodically. If it is available the channel will be flagged as available, if not it will be flagged unavailable. When sending data, the channel with the highest priority is used.

Channels will not be checked by the communication process to see if they are unavailable, only to see if they are available. If a channel has been flagged available, it will stay like this until sending data over the channel fails. Then it will be flagged unavailable and the data sent over the next available channel.

3.6 Data filtering

Depending of the speed and cost of the available connection, different amount of data will be sent to the ground. If the only connection available is slow and

expensive, one might want to limit the data sent to the ground to the most important data. On the other hand, if a fast and inexpensive connection is available, data with a lower priority could also be transmitted.

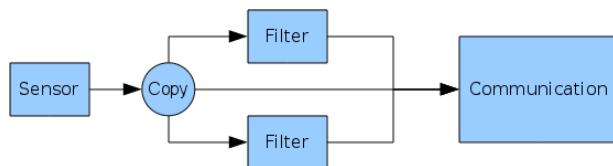


Figure 3.6: Data filtering

By adding filters to the data stream between the sensor process and the ground communication process, like figure 3.6 shows, it will be possible to reduce the amount of data sent. A filter can for instance discard part of the data or do some processing on the data and by this reduce the amount. The filter could be made specific to a given process, like reducing picture quality from a camera, or as a more generic type, like discarding half the sensor readings.

The communication process will receive both a filtered and an unfiltered data stream. The data stream would be tagged so the communication process easily can decide which data to transmit to the ground. The data stream not transmitted would just be discarded.

Chapter 4

The visual tool

4.1 Graphical user interface

The first issue that came up in the beginning of the GUI construction, was which GUI toolkit to use. The thesis had no requirements to the application other than it should be able to create a graphical representation of a process network. This opened up for defining some preconditions of my own.

First requirement, ease of use, with a good amount of documentation and examples showing the different aspects of the toolkit. This to easily get help when different issues arise. Second, cross platform availability. Even if a Windows and a Mac OS is not being developed at the moment, using a GUI toolkit which have support for more than GNU/Linux will make it simpler to port the tool to a different OS at a later time. So, with this software possibly running on different platforms, native look in the supported platforms, so it won't feel alien using it, would follow naturally.

With these in hand, Pythons[19] built in GUI toolkit, Tkinter[14] failed short. It does not provide native look and feel. Other possible tool kits what was considered was pyGTK[15], pyQT4[16] and wxPython[18]. In the end wxPython was chosen over pyGTK and pyQT4 due to its extensive demo application, where many of the different aspects of the library is presented. Because its close resemblance to wxWidgets, a lot of information regarding wxWidgets also apply for wxPython. wxPython just seemed like the best cross platform and most documented Python GUI toolkit out there.

In the beginning of the development, wxGlade[20], which is a GUI designer for creating wxWidgets[3]/wxPython user interfaces, was used to create the basics of the graphical user interface for UAVBuilder. But it was a bit limiting, and did not give the control over the development as wanted, maybe because the time this tool was used was not long enough. But as the developers on the wxGlade web page states.

It is not (and will never be) a full featured IDE, but simply a "designer": the generated code does nothing apart from displaying the created widgets.[20]

So a regular text editor was chosen instead, and the coding was done manually. This would also give a better in depth understanding of how wxPython works.

4.1.1 wxPython

wxPython is a GUI toolkit for the Python programming language. It allows Python programmers to easy create programs with a graphical user interface. wxPython is a Python implementation of wxWidgets, which is a very popular cross platform GUI library, written in C++.

wxPython is, like wxWidgets, a cross platform toolkit. Currently supported platforms are 32-bit Microsoft Windows, most UNIX or UNIX-like systems, and Macintosh OS X.

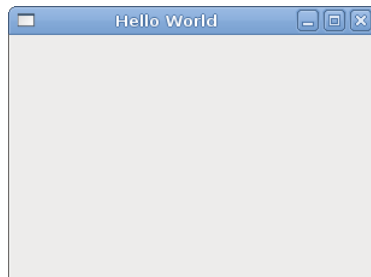
Creating a window is quite simple and does not require a lot of code. Listing 4.1 shows a small wxPython example of creating a window. Figure 4.1 shows the result in GNU/Linux and Windows XP.

Listing 4.1: hello_world.py

```
#!/usr/bin/env python
import wx

# Simple Hello World frame.
class hello_world(wx.Frame):
    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)
        self.Show(True)

app = wx.App(False)
frame = hello_world(None, title = 'Hello World', size = (300, 200))
app.MainLoop()
```



(a) Debian, Gnome



(b) Microsoft Windows XP

Figure 4.1: Hello World window in Debian running Gnome and Windows XP.

In the development of UAVBuilder wxPython v2.8.10.1 to v2.8.11.0 was used.

4.1.2 Advanced User Interface

The Advanced User Interface(AUI) library is a part of the wxPython package. This library enables the developer to create a better looking and a more advanced user interface than the standard wxPython package does. In order to use Floatcanvas, NumPy[21] must be installed. NumPy is available for *NIX, Windows and Mac OSX. So using Floatcanvas will not break any future cross platform support.

I decided to use the AUI library after playing around with window splitting. Using the AUI library gave the best look and together with some extra features like removing, rearranging and enlarging the different parts of the user interface, I felt it was the best choice. Figure 4.2 shows two windows, one created with the AUI library, and one using splitted windows.

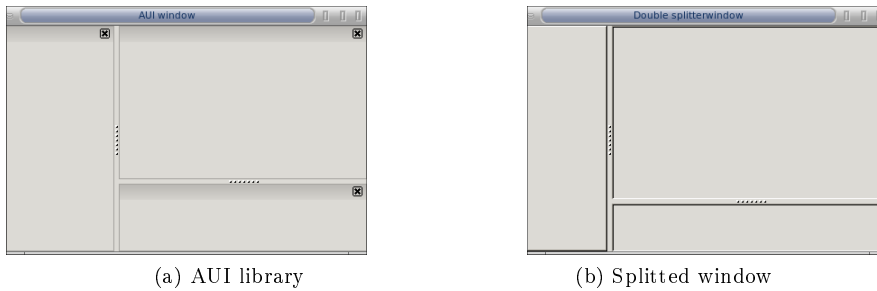


Figure 4.2: Simple three panel layout using the AUI library compared with using a window splitting approach.

4.1.2.1 Issue

One issue I came across when using Floatcanvas, was that there are no way to specify the order which each object on the canvas are drawn. The first object put on the canvas are also the bottom one and the last are on top. This is more of a cosmetic issue, by having the focused object drawn above all the other ones, it would not hide behind other objects when it was dragged around.

The way to solve this was to remove the objects from the canvas by clearing it with a built in function. A function was created that would clear the canvas and put the objects back on the canvas in a predefined order, leaving the focused object to be added last. The order which the objects was added back together,

was all the lines first, then came the triangles, and last the bitmap representing the module itself.

By doing this a new problem came up. The objects was removed from the canvas, without creating new objects. The old objects was just reused and put back in the predefined order. The problem was that when an object is added to the canvas, it is assigned an unique identifier in order to identify the object when an event occur. But when the canvas is cleared by the built in `Floatcanvas.ClearAll()` method, the id generator was also reset. Any new objects will then be assigned identifiers already used by the reused old objects, making the unique identifiers less unique. The result was that multiple objects was hit in one single event. After discussing this with the `Floatcanvas` developer, he had no immediate solution to this, but a solution to support reusing objects will be added, and that will probably also take care of this issue.

In order to fix this, a redesign of the 'rearrange objects' function could be done to not reuse the old objects, this to avoid the issue completely. But this meant that it had to be created new objects each time an object was moved on the screen. This seemed like an unnecessary extra step. So the best way was to deal with the issue and make a fix for it.

The first possible solution was to add parameter to the `Floatcanvas.ClearAll()` method, making it possible to decide if the colour assigner was to be reset. But this meant that the `Floatcanvas` source had to be changed, which is not a good idea. Another solution was to preserve the current state of the id generator, and recreate it after the canvas was cleared. The latter solution was chosen since this was the most doable solution.

4.2 UAVBuilder

As mentioned earlier the visual tool have a three panel layout for easy access to the most used parts.

The module panel have a tree structure containing all the modules available in the module directory. In the bottom of the window the configuration panel is found. It has all the different configuration options easily at hand. On the top of the configuration panel, the canvas is placed. This is where the process network is created.

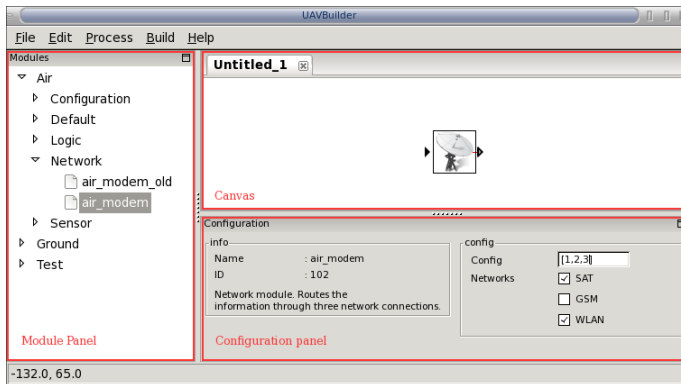


Figure 4.3: The configuration tool layout.

4.2.1 Drag'n drop support

When this feature was first tested, the OGL graphics library, which is a part of the wxPython package, was used. But it soon became clear that this library had some issues. First of all, the OGL development seems to be dead. There is no maintenance being done on the code base. Second, there is not much documentation around and some of the documentation is outdated. In order to get help, the best chance was to ask at the wxPython mailing list. But often, a part of the response would be, 'use a different library'. However, the library is still a part of the wxPython package. So, it might be dead, but it is not gone.

Due to these issues, the Floatcanvas library was often recommended as an alternative to OGL. Pretty well documented, lots of users and an email list. So quite easy to get help if needed. So the OGL library was changed in favor of Floatcanvas.

The Floatcanvas library is another part of the wxPython package. Its main goal is to give the developer an easy way of drawing objects to the screen, which suited the project fine. It does also have support for zooming that might come in handy at a later time. Being able to zoom in and out on the process network would be a nice feature in the future.

4.2.2 Load/Save

Support for loading and saving projects have been added. This to be able to resume or share a project, or make changes in an existing one. In order to make this possible, two solutions was assessed.

The first one was just to dump all the data in to a binary file. The advantage with this solution is above all simplicity. Easy to read and write to the file if you know the format. But this is also the main disadvantage. You have to know

the format, and it makes it hard for a third party developer to use or convert the save file. The file size also tends to be bigger with this approach.

The second one was to save it in an human readable manner to a text file. The advantage of this approach is the human readability of the save files, making it is easy to develop third party applications that can read them. The disadvantage is the complexity of the load/save part, since there is a lot of information that needs to be preserved, and recreated at a later stage.

The second solution was chosen due to its third party friendliness. The XML[22] format was chosen due to its standardized and open format. An XML style file is generated preserving all the necessary data for each process and the relation between them.

In the save file the entire module file is also saved. This enables save files to be exported between installations without the need to attach the needed module files. However, an import function needed to extract the module part from the save file is not yet implemented. Another possibility this solution enables, is to create the process network directly from the save file without the need of UAVBuilder. This opens up some interesting possibilities. Like having the UAV itself create the process network based on the save file, or transfer the save file directly to the UAV in flight, and have the current process network shut down and replaced with a new one.

4.2.3 Process builder

UAVBuilder have a built in text editor to create modules. The editor is opened with a skeleton of a module, where the user have to add the necessary information concerning the module configuration together with the python code for the process in order to create a complete module. This editor does also have a syntax highlighting feature for Python.

4.2.4 Gui preferences

In order to be able to configure UAVBuilder itself, there is added support for saving and restoring preferences to a file. This file is read when the application is started, and written when the preferences window is closed.

At the moment this feature is more like a proof of concept kind of thing. It works fine, but it only have a few configuration entries. Most of the configuration stuff is still only possible to change through editing the source code itself. But the foundation has been laid by having a separate class with a lot of default values.

4.3 Module library

The module system uses regular python files. They have a large header where data needed by UAVBuilder is located. Further down, a @Process statement is

found. This is where the PyCSP code starts. So a module consists of a header read by the UAVBuilder and a CSP process, used in the process network.

In Listing 4.2 an example module file is shown. There one can see the two parts the module consists of. The header on the top and the PyCSP process code at the bottom. The process in this module serves only as an example.

Listing 4.2: Configuration panel demo

```

"""
Test module for the configuration panel.
${<name> statictext <string>}${<name> textctrl <type>(<val>)}
${<name> checkbox <*1,2,*3>}${<name> radiobox <1,2,*3,4>}
"""

#####
#
# NAME: config_panel_test
# PACK: Test/Configuration
# ICON: images/conf_panel.png
# FUNC: config_panel_test
# INCO: 0
# OUTG: 2
# CONF: ${Application statictext UAVBuilder}${Update textctrl 10}
# CONF: ${Networks checkbox *SAT,*GSM}${Number radiobox One,*Two}
# DESC: Test module for the configuration panel.
#
#####

from pycsp import *

@process
def config_panel_test(cout):
    _name = 'config_panel_test'

```

Some of the header is self explanatory, but there is a few lines worth a closer look. First there is the PACK line. This line is used as a path to create a module tree list grouped together accordingly to the content on the line. This solution is a somewhat dynamic where the creator of the module self can decide how he want to group the modules. He can put whatever he like here, just divided by a forward slash and it will turn up in the module tree list. The limitation at the moment is it handles only two levels. So it needs to be on the form <level1/level2>. Figure 4.4 show how Listing 4.2 will look like in the module panel in UAVBuilder.

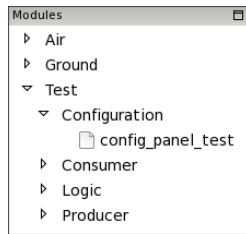


Figure 4.4: Module panel tree

The INCO and OUTG line specifies how many connections in and out a module handles. This is specified by three values. 0 for none, 1 for one and 2 for many. This is put in to let the creator of the module have some control over number of connections to and from a process and not needing to take into account every possibility.

The last one is the CONF line. This is used to specify what kind of configuration possibilities the module have. There are available four types of configuration entries.

- **StaticText.** This is for displaying text in the configuration panel. Not possible to change. Has the form <Name statictext value>.
- **TextCtrl.** Makes it possible to sent a string to the module. This is changeable. Has the form <Name, textctrl, value>.
- **CheckBox.** This is for selecting one or more values. Has the form <Name, CheckBox, *value,value,*value>. The asterisk defines which value that should be on. Multiple choices can be selected at once, or non at all.
- **RadioBox.** This is for selecting one of a number of mutually exclusive choices. Has the form <Name, radiobox, value,*value,value>. The asterisk defines which value that should be on. It can be only one at the time.

There are some limitations for what letterers that can be used in the value field. Words and such cannot be separated by spaces, '\$', '{', '}' or '*' since all of these are used in some way in the configuration statement.

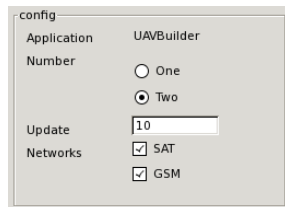


Figure 4.5: Configuration panel demo

Figure 4.5 shows the configuration window created based on the code in Listing 4.2. A small issue here. The order of which the elements in the configuration window appear, does not correspond with the order the elements are defined in the module file. The reason for this is that it has been used a dictionary to store these values in the application. Given that the dictionary is an unordered set of key: value pairs, the order of the output will not be the same as the input. Since this was not a critical issue, it has not been dealt with.

4.4 Building process network

The process network builder examines the connections between each module put on the UAVBuilder canvas. Based on this examination it creates the appropriate number of channels and connect them in a parallel statement in a build file. The build file is a regular python file with the used modules imported. A typical build file would look like the one in listing 4.3

Listing 4.3: Process network build file.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from pycsp import *

from modules.conf_new import conf
from modules.count_new import count

chan0 = Channel()
chan1 = Channel()
chan2 = Channel()

Parallel(conf([OUT(chan0),OUT(chan1)]),count(IN(chan0)),count(IN(chan1)))
```

In earlier versions of PyCSP library the developer had to specify the type of channel between processes, i.e. one to one, one to many and many to one. The new version automatically detects which type of channel is needed. But, by connecting one producer to many consumers, you would not know which process actually received the data. This works fine if there are a number of homogeneous consumer processes connected to the producer. But if there is a number of heterogeneous consumers and the producer wants to send data to a specific consumer, using a one to many connection will not work very well.

So, to resolve this issue, the process network builder always uses an exclusive channel between two processes. This makes it possible to target the data to a specific process. This is not necessary good for load balancing, since there can be processes idling while others are overloaded, but the process network will most of the time consist of heterogeneous processes.

Chapter 5

Test phase

The first stage was testing the GUI during the development. The second stage was testing the different CSP modules and the process network created by the UAVBuilder.

5.1 Testing the UAVBuilder

The testing of the GUI was an ongoing process throughout the whole development time of UAVBuilder . This was done on the local laptop running a continuously updated version of ArchLinux and Fluxbox.

5.1.1 Issues discovered

When loading a saved process network, there is a small deviation on where the objects are placed on the canvas compared to where they was before the process network was saved. The deviation is not big, but it is there. Since this is a minor bug and not critical to the use of the tool, the reason why the placement of the objects are not consistent has not been examined further.

In the configuration window does not the elements appear in the same order as in the module file. An example of this behavior can be seen in listing 4.2 and figure 4.5. The reason for this is the use of a dictionary when converting the configuration statement. Since this does not affect the usage tool either, creating a fix for this issue has been put aside for the time being. A possible solution would be to use a list instead of the dictionary. Or add an index as a part of the value in the dictionary.

5.2 Testing the CSP network

In order to create a more realistic test environment, a GNU/Linux distribution was installed in VirtualBox locally on the laptop. This would represent an UAV running an arbitrary Linux distribution. Since the whole environment runs on a

single laptop, with few or no constraints on computing power, storage or battery power compared to an UAV, there is no need for some kind of embedded OS running cheaply.

By using a full Linux distribution one also get the advantage of having available the needed software in the distributions package system. No need for tedious configuration to make things work.

5.2.1 First test environment

The first testing environment was a non GUI installation of ArchLinux installed in VirtualBox. ArchLinux was chosen since this is the same distribution that is used on the development platform, and is therefor already well tested under the development of the configuration tool and the CSP modules.

When the distribution is booted, the UAV system automatically starts running, sending information to the ground station. Sensor modules for this environment utilized some of the built in sensors on the laptop. They accessed the temperature of the CPU and the system and the web camera.

5.2.2 Second test environment

The second test environment was created using Debian installed in VirtualBox. Together with Debian, the Paparazzi project package was installed. Debian already has packages for the Paparazzi project. Paparazzi was used to gather more realistic flight data like speed, height and GPS position. For a closer description of the Paparazzi project, see section 5.3.

This was a more manual approach. A demo flight in Paparazzi had to be configured and started before the UAV system was started. The UAV system would then access the flight data over the Ivy[13] bus in the running Paparazzi demo. The Ivy bus is a text based communication protocol used to broadcast information from an application as text messages. It has a regex based subscription mechanism which enables listeners to easily gather wanted information. Paparazzi is using the Ivy bus to broadcast flight data.

Sensor modules for this environment hooked on to the Ivy bus and gathered air speed, heading and position.

5.2.3 Connection prioritizing

The connection prioritizing process was tested by having a server on the ground accepting connections on three different channels. The different channels used regular sockets, and were differentiated by simply using different ports. This because the IP address was the same on all three channels.

In order to simulate that the different channels are available at different areas in the flight, data received from the UAV was used to enable or disable the different connections on the ground. In the first test environment a simple counter, counting up and down, was used as a positioning device. When using Paparazzi, the GPS position was used.

Three channels was initially configured, WLAN, GSM and satellite. The flying UAV will at certain intervals try to connect to the predefined connections to check if there is a connection available.

Testing of the communication process have shown that

By running the test with the counting process, it was easy to see if any of the data was lost during the switching from one channel to another. Testing showed that when switching from an available channel, to a higher prioritized channel, for instance from the satellite channel to the GSM channel, no data were lost. This because the second channel would have been tested to be available prior to the data transmission.

When sending data on a channel that suddenly are not available any longer, data were lost. The reason for this is that prior to the data transmission, the communication process do not know if the channel is available or not.

5.2.4 Data filtering

Since no real world sensors was available, any further investigation of this issue have not been done.

5.3 The Paparazzi project

The Paparazzi[9] project is a free and open source UAV project. It consists of a hardware part and a software part, both developed under the influence of the Paparazzi community. The projects main goal is to create a versatile and powerful autopilot system.

The project includes both an airborne system and a ground system. The airborne system uses a custom made hardware board running the autopilot software. The hardware part of the ground system is a modem communicating with the UAV, together with a software ground station running on a laptop. The ground station can show live telemetry from the plane, depending on the current configuration of the UAV.

There is a Debian repository available in order to install the necessary software required to run the Paparazzi Ground station on a computer using a Debian[10] based distribution. It's also available an Ubuntu[11] based live CD with the Paparazzi software ready to use. The Paparazzi hardware and software is freely available under the GNU[12] General Public Licence for anybody to use, study, modify and/or share.

Chapter 6

Discussion

In this thesis the focus has been on creating a graphical tool to enable easy experimentation with CSP networks. A tool for transforming the graphical representation into a running CSP network has also been developed.

Using a CSP network in order to route, and reconfigure sensors and communication channels on an UAV booth before and in flight have been examined.

6.1 The configuration tool

The tool created is up and running. It is possible to create, configure and build a process network. Comparing the tool to a real world UAV configuration scenario is difficult since one has not been available.

But, If the needed modules to control the sensors were available, creating a simple process network that would transmit data to the ground, would not pose a big challenge.

6.1.1 Future work

- Canvas scroll. At the moment the size of the process network is limited by the size of the canvas. It is not possible to scroll the canvas in any direction. This is an issue in need of being dealt with.
- Preferences. Saving preferences needs to be extended further. At the moment there are already some values in the default class which can be added to the preferences window. The rest of the code, especially the text editor, have stuff that could be suitable in the preferences.
- Use a process network as a module. One feature that should be examined further is the possibility to use an earlier made process network as a module in a new network. One way this could be solved by is by having a special module providing a connection between the old and the new network.

- Import and export function. At the moment, there is no import of modules function. The easiest way to move modules between installations, is to copy them from the module directory. This is not a very user friendly way if a save file should be exported. Then the creator also have to include all the used modules together with the save file. The save file already contains all the required data to be able to recreate the module at a new installation. The only thing that is missing is a tool that extracts the module from the save file. Such a tool would come in handy.
- Windows version. Even Python, wxPython and NumPY all have cross platform support, UAVBuilder does not run properly in Windows. The initial window is loaded, but adding processes to the canvas give an error.
- Ground network. If the user would like to create a ground network in addition to the aerial one, he must open a new canvas and create it there. It works, but it may not be the most intuitive solution. Another way could be to divide the current canvas into an air and a ground part. Worth looking into.
- Mainstreaming the GUI. Still a few things that could be addressed. Like icons in the menu and adding a toolbar. But this is more of a cosmetic matter.

6.2 Connecting prioritizing

Still some work to be done here. It works, but have some issues regarding loss of data when a connection suddenly becomes unavailable.

Still a lot of work that needs to be done regarding the CSP network. The current state of the modules have shown that it can be used to control and configure the sensors, and be able to dynamically route the data through the fastest available connection. But they are overloaded, especially the `air_modem` module and needs to be divided into smaller units. This will also suit the CSP model better. Having small specialized processes will probably make the whole CSP network faster.

Bibliography

- [1] Lego mindstorms, <http://mindstorms.lego.com/en-us/default.aspx>
- [2] C.A.R. Hoare. Communicating sequential processes. Communications of the ACM, 21(8):666-677, pages 666–677, August 1978.
- [3] C.A.R. Hoare. Communicating sequential processes. Prentice-Hall, 1985.
- [4] CSPBuilder, <http://code.google.com/p/cspbuilder/>
- [5] Rune Møllegård Friberg and Brian Vinter. CSPBuilder - CSP baset Scientific Workflow Modelling, 2008.
- [6] Labview, <http://www.ni.com/labview>
- [7] Robolab, <http://www.lego.com/eng/education/mindstorms/home.asp?pagename=robolab>
- [8] <http://code.google.com/p/pycsp/>
- [9] Paparazzi, <http://paparazzi.enac.fr>
- [10] Debian, <http://www.debian.org>
- [11] Ubuntu, <http://www.ubuntu.com>
- [12] GNU, <http://www.gnu.org>
- [13] IVY software bus, <http://www2.tls.cena.fr/products/ivy/>
- [14] Tkinter, <http://wiki.python.org/moin/TkInter>
- [15] pyGKT, <http://www.pygtk.org/>
- [16] pyQT4, <http://wiki.python.org/moin/PyQt4>
- [17] wxWidgets, <http://www.wxwidgets.org/>
- [18] wxPython GUI toolkit, <http://www.wxPython.org>
- [19] Python programming language, <http://www.python.org>
- [20] wxGlade, <http://wxglade.sourceforge.net/>

- [21] <http://numpy.scipy.org/>
- [22] <http://www.w3.org/XML/>

Appendix A

DVD-ROM

A DVD is applied with this thesis. It have the following structure.

- `\src`, where the source code is located.
- `\doc`, where the documentation is located together with a user guide.

Appendix B

Userguide

Unmanned Aerial Vehicle Builder

UAVBuilder User Manual v1.1



Ørjan Pettersen

June, 2010

Faculty of Science and Technology

Department of Computer Science

University of Tromsø, 9037 Tromsø

B.1 Introduction

UAVBuilder is a graphical tool for building and configuring CSP sensor networks running in an Unmanned Aerial Vehicle(UAV). It is created using Python and wxPython. Its purpose is to create and configure an aerial process network controlling the various sensors on board an UAV, and a ground process network processing the data received from the UAV.

Section 2 will show how to install UAVBuilder and what is needed to make it run. Part 3 will present the different aspects of it, together with a simple example of a SCP network, and how to build and run it.

B.2 Installation

At the moment there is no installation routine for UAVBuilder. Just decompress the tar.gz file either by the distributions way of doing it or by manually decompressing it in a terminal. UAVBuilder is run by double clicking on 'builder.py' file or again, manually running it from a terminal. Listing B.1 shows the complete terminal session in GNU/Linux.

Listing B.1: Terminal way of running UAVBuilder.

```
# cd <path_to_file>
# tar -xvzf uavbuilder.tar.gz
# cd uavbuilder
# ./builder.py
or
# python builder.py
```

B.2.1 Dependencies

In order to run the program, the following software needs to be installed. Python[1] v2.5, wxPython[2] v2.8 and NumPy[3] v1.3. It is quite possible older versions will work, but it has not been tested.

The execution of the CSP network created, requires PyCSP[4] v0.5 and later. Versions before v0.5 will not work, because of some changes to how channels are named.

To install the extra software, it is preferable to use the built in package manager in your distribution. If you choose not to use the package manager or if they do not exist as a ready package, follow the installation routine on the developers web page.

B.3 UAVBuilder

UAVBuilder have three panels which serves different purposes. Figure B.1 shows the UAVBuilder layout.

- The module panel has a tree structure with all the available processes listed. It searches through the process catalogue and creates a tree representation of all the processes.
- The canvas is the area where the processes are dropped. The canvas has drag and drop capabilities making it possible to rearrange the processes after they have been dropped on to the canvas. Multiple canvases are possible, and each is shown in its own tab. The order of the tabs can also be rearranged.
- The configuration panel shows information concerning the process and custom configuration options.

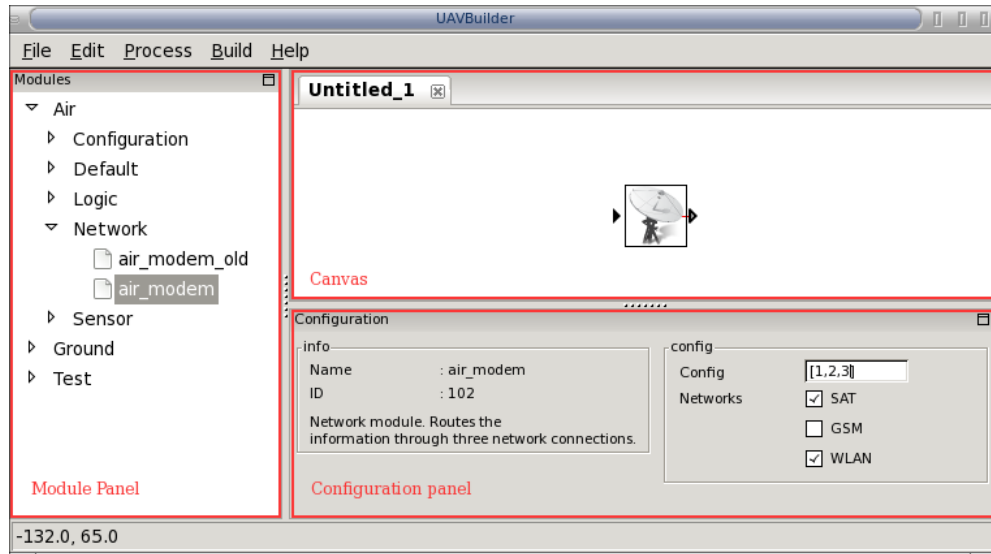


Figure B.1: UAVBuilder layout.

B.3.1 Usage

To select a module from the module panel, double click with left mouse button on the module. To drop it on the canvas, click in the canvas where you want the module to be. A module placed on the canvas, can be moved by dragging it around with the left mouse button.

The image on the canvas has some small triangles that represent connections in and out. A triangle indicates one or more connection capabilities. No triangle indicate no connection. Incoming connections are indicated on the left with a triangle pointing towards the module. An outgoing connection is indicated with a triangle on the right side pointing away from the module. The outgoing triangle is movable to be able to connect to another module.

The configuration of the module is done in the configuration window. Each module has it own set of configuration elements specified from a set of available types. For a closer description see section B.3.3.1

B.3.1.1 Delete module

A module can be deleted from the canvas by clicking with the right mouse button on it. Figure B.2 shows right click menu.

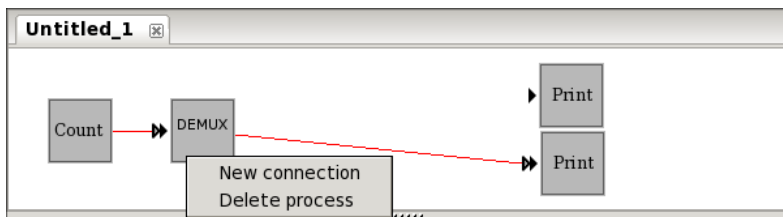


Figure B.2: Right click menu.

B.3.1.2 New connection

Default each module that has at least one available connection out, will have a triangle showing this connection. But if the module supports more than one outgoing channel, it has to be manually created by right clicking on the module. Figure B.2 shows this menu.

B.3.1.3 Delete channel

To delete a channel connected to another module, simply right click on the triangle connected to the target module. A list will show all the channels connected to the module. The channel is identified by its parent id and name. Choose the appropriate channel to delete. Figure B.3 shows the delete menu.

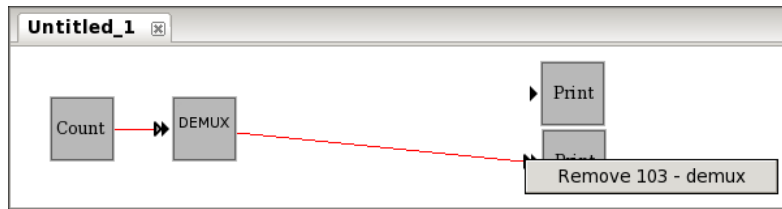


Figure B.3: Delete channel.

B.3.1.4 Build module

When the appropriate modules are added and connected to each other, the process network is created by hitting <Ctrl+b> or <Build -> Build Process Network> in the menu.

B.3.2 Menu

Overview of the different menu choices and what they do.

File -> New

Creates a new canvas. Loads in a new tab.

File -> Load

Opens a file dialog window. Load project from file.

File -> Save

Opens a file dialog window if the project is new. If the project has been saved before, it is saved directly to the old file.

File -> Save as...

Opens a file dialog window to save a project to file.

File -> Save all

Opens a file dialog to save all canvases to file.

File -> Quit

Quit application.

Edit -> Preferences

Open preferences dialog.

Process -> Custom Process

Opens a text editor for creating custom processes.

Build -> Build

Converts the graphical representation to a CSP process network.

Help -> Help contents

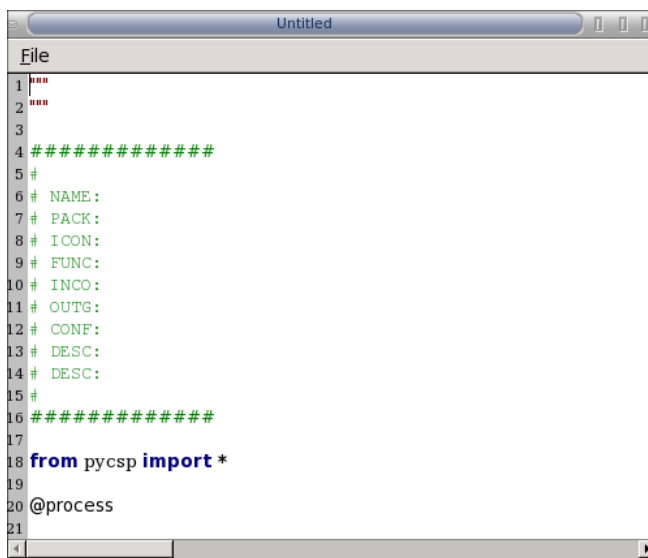
Open this file with the distributions default PDF reader.

Help -> About UAVBuilder

Shows an about box.

B.3.3 Create custom module

The tool features a text editor to create custom modules. The editor supports syntax highlighting for python, and is is opened with a skeleton of a module.



```
File
1 """
2 """
3
4 #####
5 #
6 # NAME:
7 # PACK:
8 # ICON:
9 # FUNC:
10 # INCO:
11 # OUTG:
12 # CONF:
13 # DESC:
14 # DESC:
15 #
16 #####
17
18 from pycsp import *
19
20 @process
21
```

Figure B.4: Module editor.

The header of the module file have a number of fields. These fields are required by UAVBuilder in one way or another to be able to use the module.

- **Name:** The name of the module. Will be shown in the module tree list, in the configuration window and as a identifier when deleting a connection between two modules.
- **Pack:** This decides how it will be shown in the module tree list. Requires two levels. Has the form <level1/level2>, i.e <Test/Configuration>.
- **Icon:** The path to the icon image. Will represent the module on the canvas.
- **Func:** The name of the process function. To tell the process builder which process to import.
- **Inco:** Defines the number of incoming channels the module can handle. 0 for none, 1 for one and 2 for many.
- **Outg:** Same as for inco, but for outgoing channels.
- **Conf:** Defines the configuration parameters to the module. Has four configuration types. The four choices it supports are static text, a string box, a multiple choices list and a list of mutually exclusive choices. More on this later.
- **Desc:** A description that describes the module. Is shown in the in the status line when highlighting the module in the module tree list and in configuration window when it is focused on the canvas. It supports multiple lines with a 'Desc' keyword in front of each line.

B.3.3.1 Configuring modules

As mentioned there are four different configuration types.

- **StaticText.** This is for displaying text in the configuration panel. Not possible to change. Has the form <Name statictext value>.
- **TextCtrl.** Makes it possible to send a string to the module. This is changeable. Has the form <Name, textctrl, value>.
- **CheckBox.** This is for selecting one or more values. Has the form <Name, CheckBox, *value,value,*value>. The asterisk defines which values that are enabled by default. Multiple choices can be selected at once, or non at all.
- **RadioBox.** This is for selecting one of a number of mutually exclusive choices. Has the form <Name, radiobox, value,*value,value>. The asterisk defines which value that are enabled by default. It can be only one at the time.

An example of a module file and the configuration part of the UAVBuilder is shown in listing B.2 and figure B.5.

Listing B.2: Configuration panel demo.

```

"""
Test module for the configuration panel.
${<name> statictext <string>}${<name> textctrl <type>(<val>)}
${<name> checkbox <*1,2,*3>}${<name> radiobox <1,2,*3,4>}
"""

#####
#
# NAME: config_panel_test
# PACK: Test/Configuration
# ICON: images/conf_panel.png
# FUNC: config_panel_test
# INCO: 0
# OUTG: 2
# CONF: ${Application statictext UAVBuilder}${Update textctrl 10}
# CONF: ${Networks checkbox *SAT,*GSM}${RadioBox radiobox One,*Two}
# DESC: Test module for the configuration panel.
#
#####

from pycsp import *

@process
def config_panel_test(cout):
    _name = 'config_panel_test'

```

The observant reader will see that the order in which the configuration elements in the module file are not the same as in the configuration window. This is a small bug, but it is not critical for the usage. So at the moment, this issue have not been fixed.

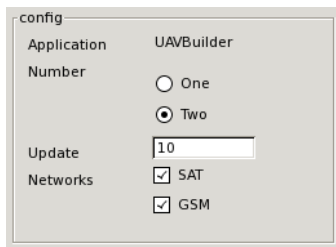


Figure B.5: Configuration panel demo.

In order to be able to utilize the configuration in a process network, a configuration module is needed. It has to be connected to every module that needs to be configured. And the module that wants the configuration needs to handle an incoming connection. Listing B.3 shows the configuration data represented

in the configuration panel in listing B.2 and figure B.5

Listing B.3: Configuration result.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

""" A configuration class containing configuration data for all modules. """

class uav_config:
    config_dict = {'config_panel_test': {'Application': 'UAVBuilder', '
        Networks': ['SAT', 'GSM'], 'Number': ['Two'], 'Update': '10'}}
```

B.3.4 Preferences

The preferences window have one entry at the moment. This is where the modules is located, and can be changed by browsing for a new location or just typed manually. Figure B.6 shows the preferences window.

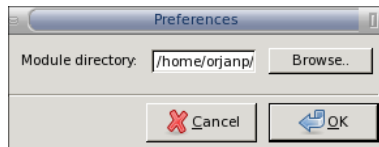


Figure B.6: Preferences window.

The configuration file is a hidden file, named `.uavbrc`. The syntax of the file is shown in listing B.4.

Listing B.4: Preferences file.

```
[modules]
modules_path = modules/
```

B.3.5 Building process network

Figure B.7 shows the graphical representation of a small process network.

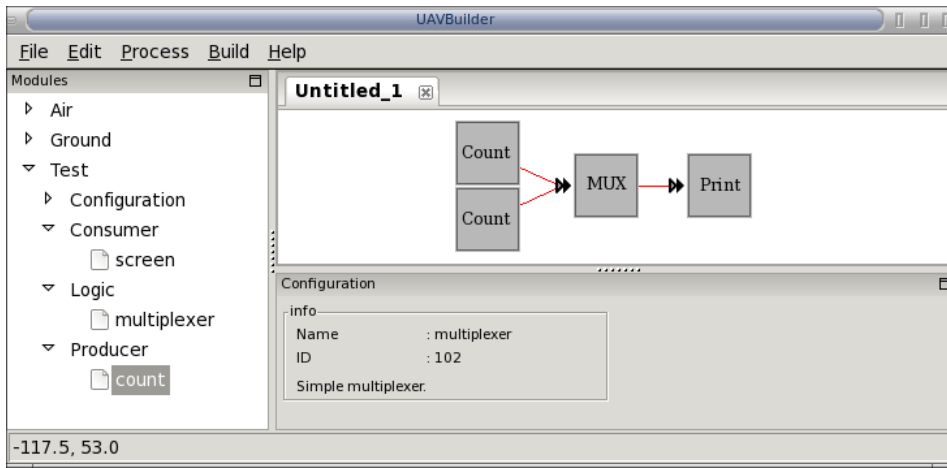


Figure B.7: Process network example.

The two “Count” processes will send a number to the “MUX” process, where the “MUX” sends it to the “Print” process. This process prints the number it receives directly to the screen. Since none of these modules access special built in hardware, they can simply be tested on most computers. To build the process network, just click on the build menu and select build. Then a file called `csp_build.py` will be generated. Listing B.5 show the content of the file generated by the build process. In order to test the build file locally PyCSP must be installed.

Listing B.5: Process builder result.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from modules.screen import screen
from modules.multiplexer import multiplexer
from modules.count import count

#####

chan0 = Channel()
chan1 = Channel()
chan2 = Channel()

Parallel(multiplexer([IN(chan1),IN(chan2)], [OUT(chan0)]),
        screen([IN(chan0)], count([OUT(chan1)]), count([OUT(chan2)]))
```

B.3.5.1 Running the process network

To run the example, issue the following command in a terminal and the result will be in listing B.6.

Listing B.6: Result of running process network.

```
# python csp_build.py
0
0
1
1
2
2
...
```

B.3.5.2 Stop the process network

Since the modules used in this example have no built in kill routine, the easiest way to stop it is by killing the python process from a terminal. Listing B.7 shows how.

Listing B.7: Stop process network.

```
# killall python
```

Appendix B.4 shows the complete source code for each of the three modules in this example.

B.4 Example source

Listing B.8: count.py

```

"""
Simple count module.
"""

#####
#
# NAME: count
# PACK: Test/Producer
# ICON: images/count.png
# FUNC: count
# INCO: 0
# OUTG: 1
# CONF:
# DESC: Simple count module.
#
#####

import time
from pycsp import *

@process def count(cout):
    data = 0
    while True:
        alt = Alternation([(cout[0], data): ''])
        alt.select()
        time.sleep(1)
        data += 1

```

Listing B.9: multiplexer.py

```

"""
Simple multiplexer.
"""

#####
#
# NAME: multiplexer
# PACK: Test/Logic
# ICON: images/mux.png
# FUNC: multiplexer
# INCO: 2
# OUTG: 1
# CONF:
# DESC: Simple multiplexer.
#
#####

from pycsp import *

@process def multiplexer(cin, cout):
    while True:
        lst = []
        for item in cin:
            lst.append({item: ''})
        alt = Alternation(lst)
        data = alt.select()[1]
        alt = Alternation([(cout[0], data): '1'])
        alt.select()

```

Listing B.10: screen.py

```
"""
Simple print module.
"""

#####
#
# NAME: screen
# PACK: Test/Consumer
# ICON: images/print.png
# FUNC: screen
# INCO: 1
# OUTG: 0
# CONF:
# DESC: Simple print module.
#
#####

from pycsp import *

@process def screen(cin):
    alt = Alternation([cin[0]: ''])
    while True:
        data = alt.select()[1]
        print data
```

Bibliography

[1] Python, <http://www.python.org/>

[2] wxPython, <http://www.wxpython.org/>

[3] NumPy, <http://numpy.scipy.org/>

[4] PyCSP, <http://code.google.com/p/pycsp/>

[5] The image used at the front page is from Freeclipartnow,
<http://www.freeclipartnow.com/military/airforce/uav-8.jpg.html>