MODELLING AND SIMULATION OF A UAV AUTO-PILOT EMBEDDED SOFTWARE USING UML AND A CASE TOOL

Véras, Paulo Claudino, pcv@ita.br
Góes, Luiz Carlos Sandoval, goes@ita.br
Villani, Emília, evillani@ita.br
Div. Eng. Mecânica-Aeronáutica, Instituto Tecnológico de Aeronáutica – ITA
Pça. Mal. Eduardo Gomes, 50, 12228-900. São José dos Campos – SP.

Abstract. *The main difficulty in developing quality software is specifying and designing the conceptual construct that underlies the software. The system modelling makes possible the validation of the designed system and the evaluation of its features before implementation. This work aims the auto-pilot embedded software modelling of an Unmanned Aerial Vehicle – UAV and the verification of the accomplishment of the system requirement through its simulation. The language used to model the software is the Unified Modelling Language – UML and it is implemented in the computer using the Rational® Rose® RealTime – RRRT as the Computer-Aided Software Engineering – CASE tool. In order to allow the simulation of the feedback control system of the software, a flight simulator developed in MatLab® environment, into Simulink®, is integrated with RRRT through a serial communication. Simulation responses can be seen in both CASE tool, which prints calculated values and shows statechart diagrams, and flight simulator, which has an artificial horizon. Results show that the software is successfully modelled and implement in the CASE tool, as well as the integration of RRRT and MatLab is very useful for assess the software behaviour during its simulation.*

Keywords*: Software modelling, UAV, Unified Modelling Language, Rational Rose RealTime.*

1. INTRODUCTION

One of the main difficulties in the development of quality software is the conceptual specification and design (Brooks, 1995). In order to overcome this difficulty, the software development can include and be supported by a modelling process, which covers, among others, the activity of capturing and analyzing the system requirement through use cases (Iwu et. al., 2007). The purpose of the modelling process is to make easier the communication of ideas between designers and assess the system features before implementation. It is, therefore, an important tool for supporting the validation of the designed system (Damaševičius, 2004).

Modelling and analysis techniques have been increasingly used to enhance traditional system engineering techniques and improve the system quality (Massink, 2006). Among the methods and languages proposed for software modelling, the UML (Unified Modeling Language) has emerged as a pattern in both academic and industrial field. Its application is supported by CASE (Computer-Aided Software Engineering) tools that facilitate the development process from analysis through code (Kimour, 2005).

The use of UML and CASE tools is particularly important in the aerospace context, where one of the main concerns is safety. The design of embedded software for aerospace application is critical. As a consequence, the certification of the software by an independent authority is an essential requirement. According to Palma (2005), software development which follows a standard and documented process has more ease to obtain a certification. In order to obtain quality and reliability, a number of techniques can be employed, such as standardization, structuralized design, formal specification languages and other tools considered capable of reducing the error introduction probability (Palma, 2005). Many works are also dedicated to developing analysis procedures based on modelling techniques and CASE tools (Paludetto et al, 1999), (Giese et al, 1999).

In this context, the purpose of this paper is to present a new approach for embedded system validation based on the integration of a CASE tool with a dynamic simulation environment. The idea is to analyse not only the embedded control software but also the behaviour of the controlled system. On the contrary of traditional approaches of Software Engineering, the purpose is to validating the control software + controlled plant (Fig. 1).

This approach is applied to the auto-pilot embedded software of an UAV (Unmanned Aerial Vehicle). The software is modelled using UML, and the models are implemented in the RRRT (Rational® Rose® RealTime) CASE tool. The software model in the RRRT is integrated with a flight simulator that models the aircraft dynamics. The flight simulator has been developed in the MatLab® environment, using Simulink®.

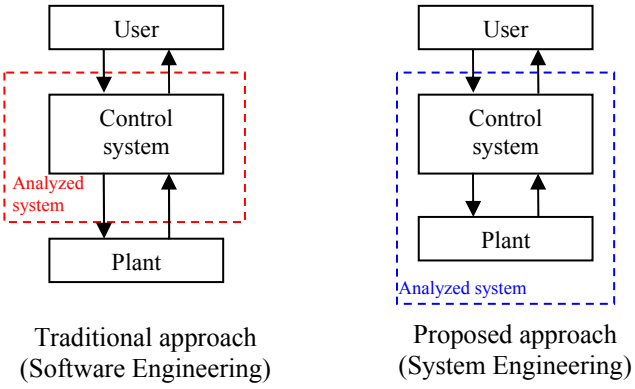

Figure 1. Traditional approach versus proposed approach.

This paper is organized as follows. The next section describes the UAV that is used as a case study and the autopilot software. Section 3 is dedicated to the embedded software modelling in the RRRT. Section 4 describes the flight simulator and its integration with the RRRT. Finally, Section 5 discusses some results and presents the main conclusions of this work, highlighting the advantages and drawbacks of the proposed approach.

2. THE UAV AND ITS AUTO-PILOT EMBEDDED SOFTWARE

The UAV used as case study is 1.8 meters long, has 3.6 meters of wingspan and can carry 12 kilograms of payload. Its onboard computer uses the ADSP-BF533 microprocessor, also known as *Blackfin*. This microprocessor works with a clock of 300 MHz, has 128MB of RAM and 10 levels of pipeline. The on board computer is also composed of some devices that make the communication with actuators and sensors using a serial cable and the RS-232 protocol. The signals sent by the microcontroller to the actuators are converted by a driver using the PWM (Pulse Width Modulation) technique. These signals control servomotors, which are connected to the control surfaces (ailerons, elevator and rudder) and to the engine. The microcontroller receives data from the GPS (Global Positioning System) receptor, inertial sensor, pressure sensor and control-radio receptor.

The UAV considered in this work takes off and lands controlled by a pilot through the FR (Frequency Radio) control. Only the cruise flight may be carried out automatically. It depends on the flight operation mode, which has three possible values: 0, 1 or 2. Mode 0 means that the flight is totally controlled by the pilot and there is no interference of the control system - the control signal sent by the pilot is transmitted to the actuators with no change. In this mode, the surface tilt angle is directly proportional to the pilot command. In the second operation mode (mode 1), the pilot also sends the surface command, as in mode zero. However, a feedback control system interferes in the pilot command. This intervention is the feedback of the angular speed in question (for example, the pitch angular speed (q) for the elevator – which controls the airplane altitude). The result is the input of a PID controller, which provides the output to the elevator. The last operation mode (mode 2) is totally automatic and is used in cruising flight. The UAV is controlled by the onboard computer, whose embedded software carries out the control in real time. In this case, other than the angular speed, the feedback system uses also the angle (in the altitude control, this is the pitch angle – θ).

Figure 2 shows the general feedback control system of the UAV. For each operation mode, corresponding links must be made to arrange the current control system. In operation mode 0, only the connection number 4 is closed. In mode 1, connections number 2 and 5 are closed, while in mode 2, connections number 1, 3 and 5 must be closed.

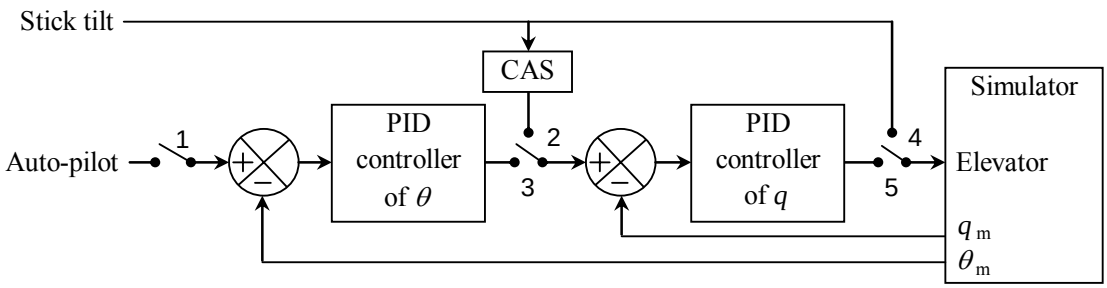

Figure 2. Feedback control system.

The UAV embedded software is divided basically in two modules, the first one is responsible for the guidance of the airplane, while the other is the auto-pilot. The guidance must calculate the UAV trajectory based on the GPS receptor, sensors signals and desired destination. This trajectory is sent to the auto-pilot module, which acts in the servomotors in

order to maintain the aircraft in the direction defined by the guidance. In this paper the proposed approach is applied and restricted to the elevator controller of the auto-pilot module.

Independently of the current operation mode, the auto-pilot module receives the operation mode, the stick tilt, the pitch angular speed, the pitch angle and the reference pitch angle. Depending on the operation mode, it uses just the data that it needs to calculate the command to the actuators, the remaining data is ignored. The operation mode 0 uses only the stick tilt. The operation mode 1 uses the stick tilt and pitch angular speed. Finally, mode 2 uses all the data above mentioned, except the stick tilt. During the execution of the control software, all these data are received with a frequency of 50 Hz. In each interaction, the auto-pilot software calculates the control signals and sends it to the actuators.

3. THE UAV AUTOPILOT EMBEDDED SOFTWARE MODELLING

The first step of the software modelling process is to identify the system use cases and actors and draws the Use Case Diagram. It shows in a high level the needs of the system. Firstly, the actors are identified by verifying the devices that the system communicates with. In one hand, there are the devices whose role is to send data to the software: GPS receptor, Inertial Sensor, Boom (pressure sensors) and Pilot. On the other hand, there are the actuators that receive commands from the system: Elevator, Ailerons, Rudder, Flap and Motor. Actually, the commands are sent to servomotors that, in turn, are connected to the mentioned actuators. The most general use case is called "Adjust Attitude", whose role is to calculate the attitude of the UAV. It uses five other use cases to perform this use case (each one corresponds to one actuator). Furthermore, there is one use case to carry out the communication between each one of the actors and the "Adjust Attitude" use case. The role of these use cases is to send the data proceeded from the actors to that use case, as well as to adjust the UAV attitude with each one of the actuators (send the calculated command to them). The diagram of Fig. 3 shows these elements in the Use Case Diagram.

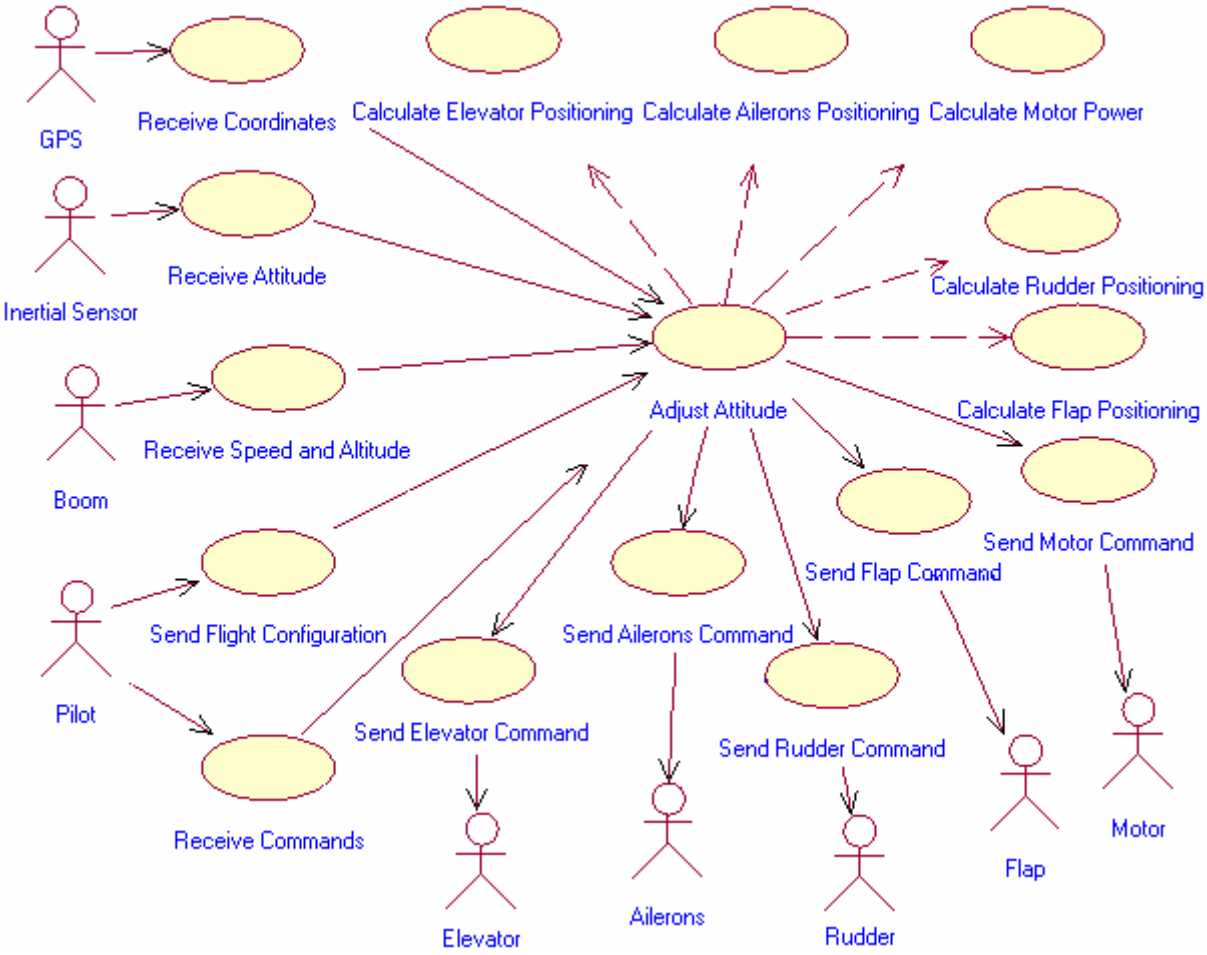

Figure 3. Use Case Diagram.

The next step is to make an Activity Diagram for each use case, detailing its functionality. As an example, Fig. 4 shows the Activity Diagram of the "Send Attitude" use case. Firstly, this use case initializes the execution, starts the timer, waits to receive data from the Inertial Sensor and sends it to "Adjust Attitude" use case. After that, it verifies

whether the execution was finished or not and in the positive case, it verifies if the time interval of data reception has been reached or not (20 ms). In the positive case, this cycle is repeated. In the negative case, it waits for the time interval.

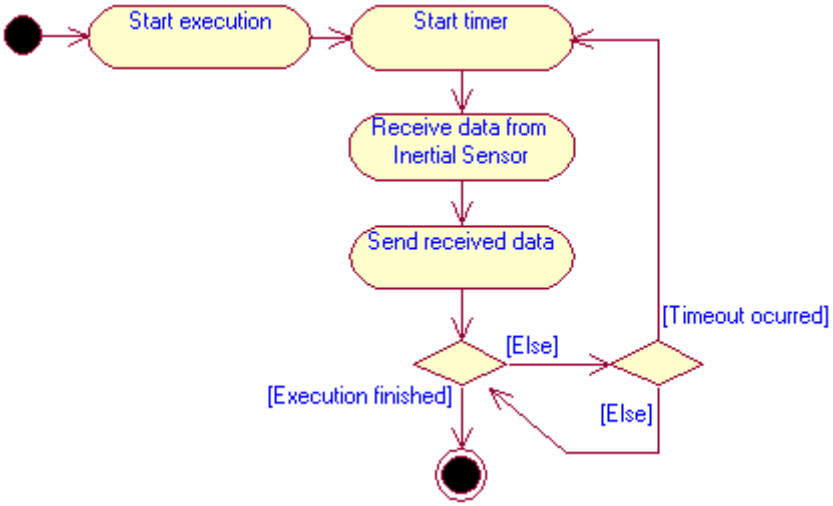

Figure 4. Activity Diagram of "Send Attitude" use case.

The third set of diagrams is the sequence ones. A Sequence Diagram is built for each use case.

Figure 5 presents as an example the Sequence Diagram corresponding to the scenario of receiving data from Inertial Sensor. The first action of the system is to open and configure the serial communication port. After that, "DataReceiver" capsule class sends a pointer to the corresponding port of the "ElevatorController" capsule class. The sequence of the actions is then to send the data from "Inertial Sensor" actor to the "DataReceiver" capsule class, which sends the data to the "ElevatorController" capsule class. These latter two actions are repeated until the system execution is halted. The Sequence Diagram makes visible how the solution is implemented, since capsules classes are explicated in the diagrams, as well as their interaction with each other.

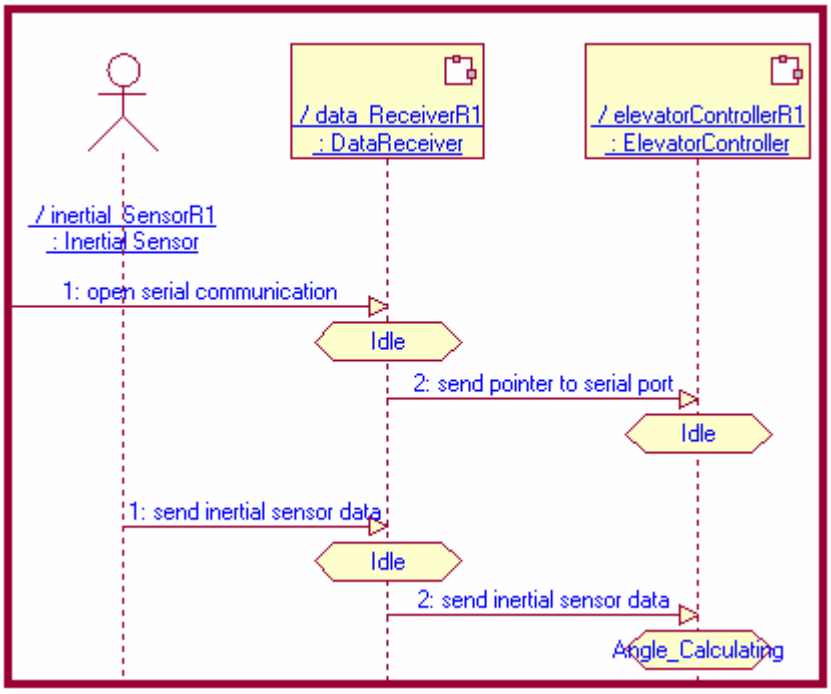

Figure 5. Sequence Diagram of Inertial Sensor data receiving.

The next diagram in this modelling process is the collaboration one. The Collaboration Diagram shows the same information of the Sequence Diagram, but it focuses on the relationship between components, instead of making

explicit the time sequence of interactions. It shows, for example, that “Inertial Sensor” actor interacts with “DataReceiver” capsule, which interacts with “ElevatorController” capsule. Due to the limited space it will be omitted in this paper.

The Structure Diagram is the next step. It shows the communication between the capsule classes. “DataReceiver” communicates with “ElevatorController”, which also communicates with “Elevator”. This communication proceeds in the following way: first, “DataReceiver” capsule class receives data coming from all of the sensor devices (in this study, these data come from the flight simulator), such as Inertial Sensor and Boom (as told before, this is done through the PC serial port). After that, these data are sent to “ElevatorController”, whose task is to calculate the actuation of the elevator in order to maintain or carry the UAV to the desirable altitude. After calculating, the actuation command is sent to “Elevator” capsule class, which finally plays the role of actuator. This is done by sending back to the simulator the calculated command. The simulator thus calculates new values to the sensor devices according to the actuator command and the airplane dynamic. After that, this cycle is repeated. The Structure Diagram is shown in Fig. 6.

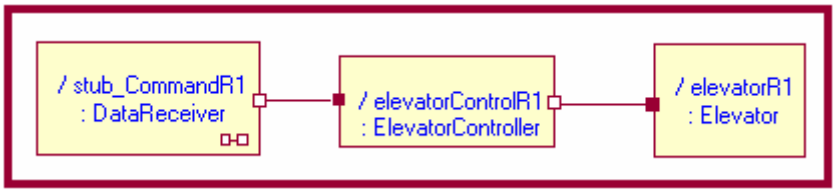

Figure 6. Structure Diagram.

In order to model the remaining of the software, corresponding to the control of the rudder and ailerons, the necessary changes in previous diagrams are to insert a capsule class to each new actuator and its respective controller (for example, “Rudder” and “RudderController”, where “Rudder” is linked to “RudderController”, which is linked to “DataReceiver”). Therefore, “DataReceiver” would be a ‘central’ capsule class whose role would be receive data from sensor devices and send them to the concerned controllers. Each controller would calculate the actuation command of the concerned actuator and would send this command to it.

State Diagrams exhibit the behaviour of each capsule class shown in Fig. 6. Through them it is possible to see how their tasks are implemented. Modelling software in RRRT allows one to insert the source code in the State Diagrams and, during the execution of the model, these codes are executed.

The State Diagrams of “DataReceiver” and “Elevator” capsule classes are very simple, since their purposes are just receiving and sending data through serial communication port.

Figure 7 shows the State Diagram for the “DataReceiver”. The “open_serial_port” transition is the first to be executed and this is done just once. This opens and configures the serial communication port, as well as sends a pointer that points to the serial port to the “ElevatorController” capsule class. After that, “send_vector” transition is executed. This reads data from serial port and sends this data to “ElevatorController”. It receives five numbers of float type: operation mode, stick tilt, pitch angular speed, pitch angle and reference pitch angle.

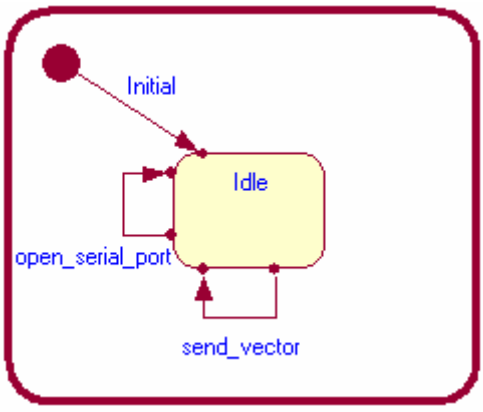

Figure 7. State Diagram of “DataReceiver” capsule class.

The State Diagram of “ElevatorController” is more elaborated and is presented in Fig. 8. The first transition to be executed is “get_pointer”, which receives the pointer sent by “DataReceiver”. This transition is executed only once. After that, this diagram decides what way to go, depending on the operation mode. If the mode is not a valid value, it goes back to the first state. If it is a valid value, it goes to the appropriate way, as can be seen in the figure. Doing this, the execution makes a sequence of steps in a loop: receive data from “DataReceiver” in “request_vector” transition;

calculates the elevator actuation command in “calculating” transition, as well as sends it to “Elevator” capsule class. The state diagram only gets out of this loop if the operation mode is changed, which makes it goes to the new operation mode loop.

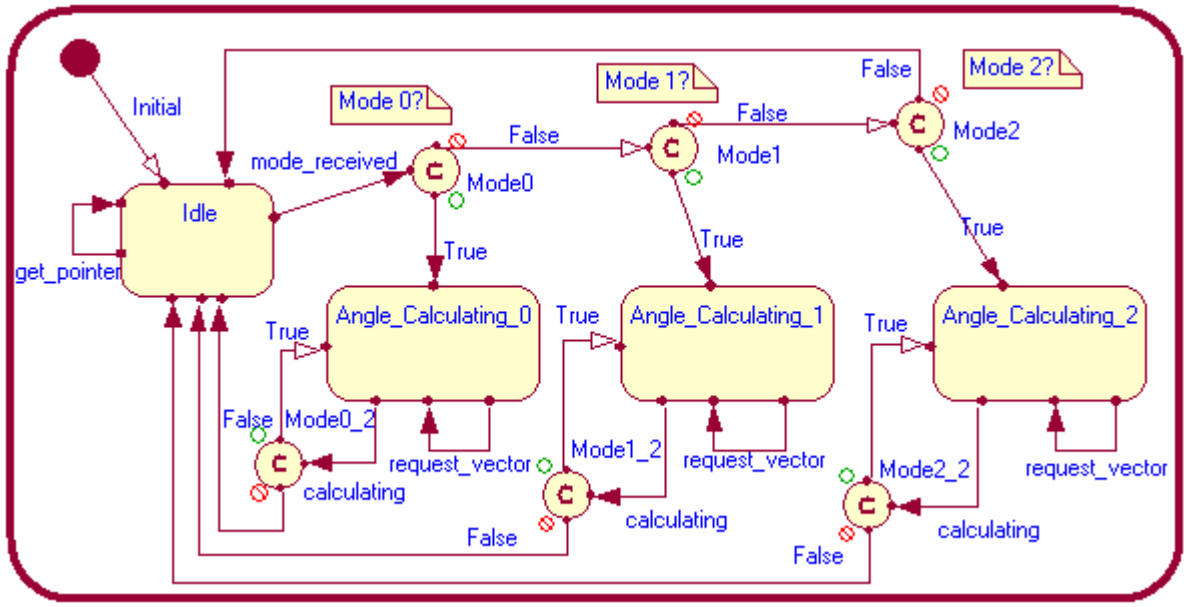

Figure 8. State Diagram of “ElevatorController” capsule class.

Figure 9 shows the State Diagram of “Elevator” capsule class. The “receive_pointer” transition is the first to be executed in this diagram. This receives the pointer to the serial port. It is executed just one time. After that, the “send_command” transition is executed in a loop until the model execution stops. This transition receives the elevator actuation command from “ElevatorController” and sends it to the flight simulator through the serial port.

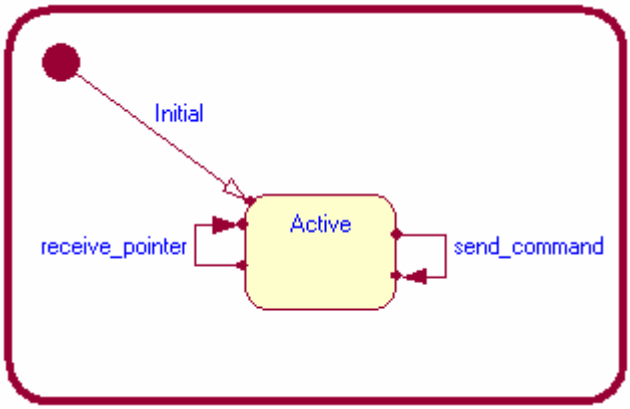

Figure 9. State Diagram of “Elevator” capsule class.

The last diagram approached in this paper is the Class Diagram, which shows the static structure of the system. This diagram displays the classes of the system, their interrelationships and the operations and attributes of the classes. RRRT generates this diagram automatically after Structure and State Diagrams have been done. This diagram has six classes (among capsules and static classes). As it can be seen in Fig. 10, there are two static classes that have not been mentioned before – “Controller” and “Serial” – and one capsule class – “TOP”.

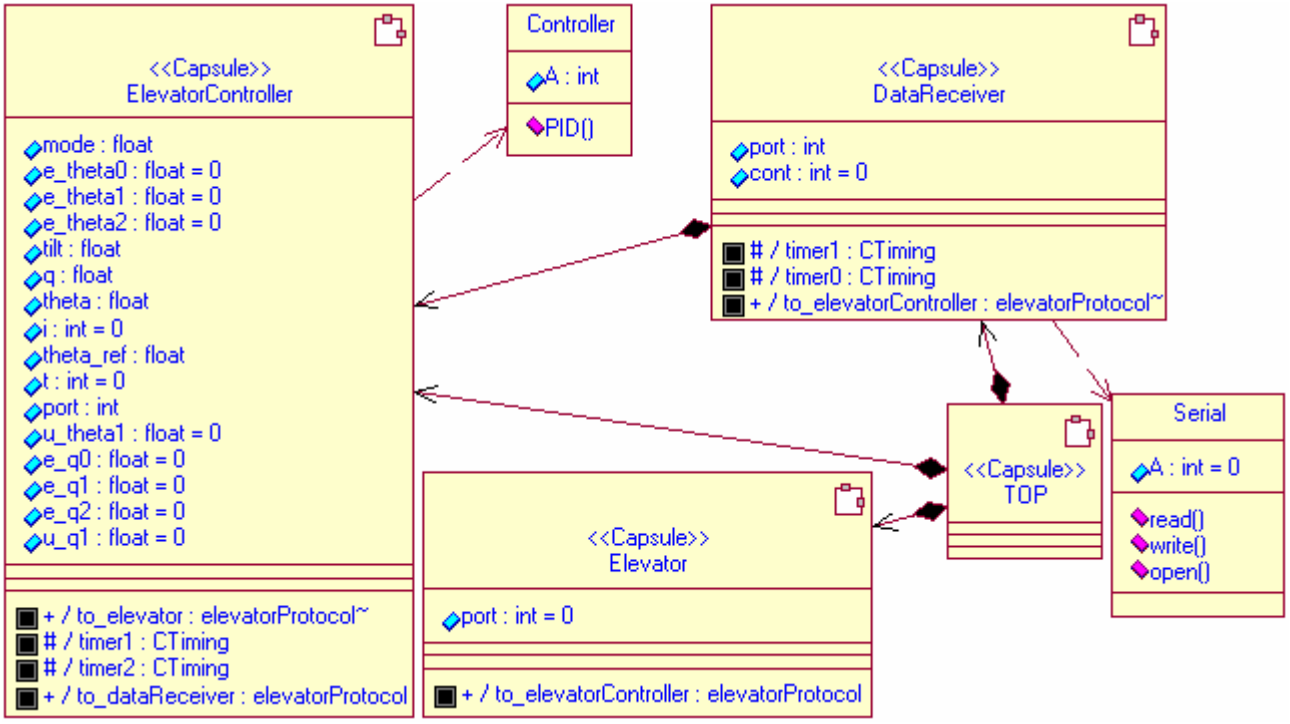

Figure 10. Class Diagram of the system.

The "ElevatorController" capsule class uses the services provided by "Controller" class to calculate the response of the PID controller of the system. This class has an operation called "PID" that calculates the output of the PID controller to θ and q variables. The "Serial" static class provides services of communication between the model and the simulator through the serial port. It has three operations: read, write and open, whose purposes are clear. The only purpose of "TOP" capsule class is to make the links between the ports of the other capsule classes.

Component and Deployment Diagrams will not be treated in this paper. Since a single program is being modelled and it runs in a single processor, these diagrams are not necessary. Furthermore, the software uses no "dll" file or similar, i.e. it is generated only one executable file.

Next section describes the flight simulator used in this work, as well as its integration with the model inside the RRRT.

4. FLIGHT SIMULATOR AND ITS INTEGRATION WITH RRRT

The flight simulator used in this work is developed into Simulink® - MatLab® environment. It models the dynamics of the UAV and considers that the airplane is a six degree of freedom (6DOF) body – of which, three are the rotations around the three axes of the airplane and three the translation along these axes. The simulator structure is composed of five blocks as shown in Fig. 11. However, the configuration used in this work does not include the "Atmosphere Model" block.

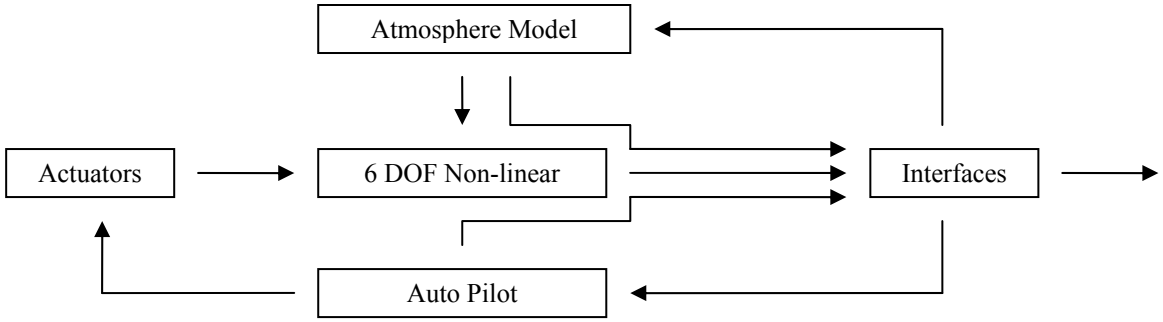

Figure 11. Simulator structure.

The "Actuators" block plays the role of the control surfaces and motor (system actuators). The "6DOF Non-linear" has the dynamics of the airplane modelled with 6 degree of freedom. The "Interfaces" organizes the data coming from

"6DOF Non-linear" and "Auto Pilot" and sends them to blocks out of the simulator. Finally, "Auto Pilot" represents the auto-pilot embedded software modelled in RRRT. It sends the data (described in section 2) from the simulator to the RRRT and receives the actuation command from it. After that, "Actuators" receives this data and the cycle just described is repeated.

The simulator runs in one computer while the embedded software model (into RRRT) runs in another one. The stick tilt is entered through a joystick connected to the simulator computer. This linking can be seen in Fig. 12.

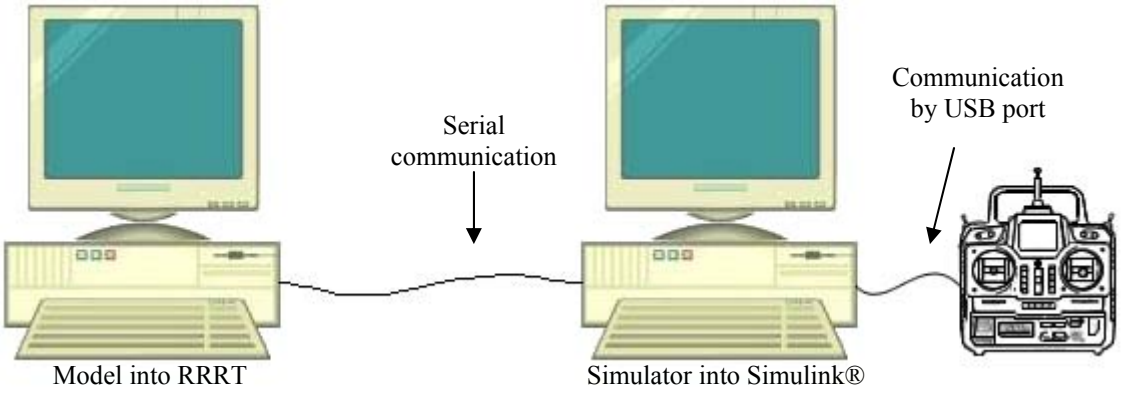

Figure 12. Connection between the control software and the flight simulator.

Joystick data is an input in the simulator, which gathers this data with the operation mode, sensors and GPS receptor data to send them to the RRRT. It functions as a slave in this context, i.e. it just acts when receives a command from Simulator. Thus, the model initialises the software (opens and configures the serial port, sets the initial values to the variables) and holds on its execution until it receives some data from serial port. Whenever it happens, the model calculates the actuator command based on the input data and the logic of the controller and sends the result of this processing back to the simulator as soon as possible. This cycle is repeated throughout the simulation process.

During this combined execution of the control software and flight simulator, it is possible to visualize the behaviour of the model looking at the State Monitors of the State Diagrams. As an example, Fig. 13 shows the State Monitor of the "ElevatorController" capsule class.

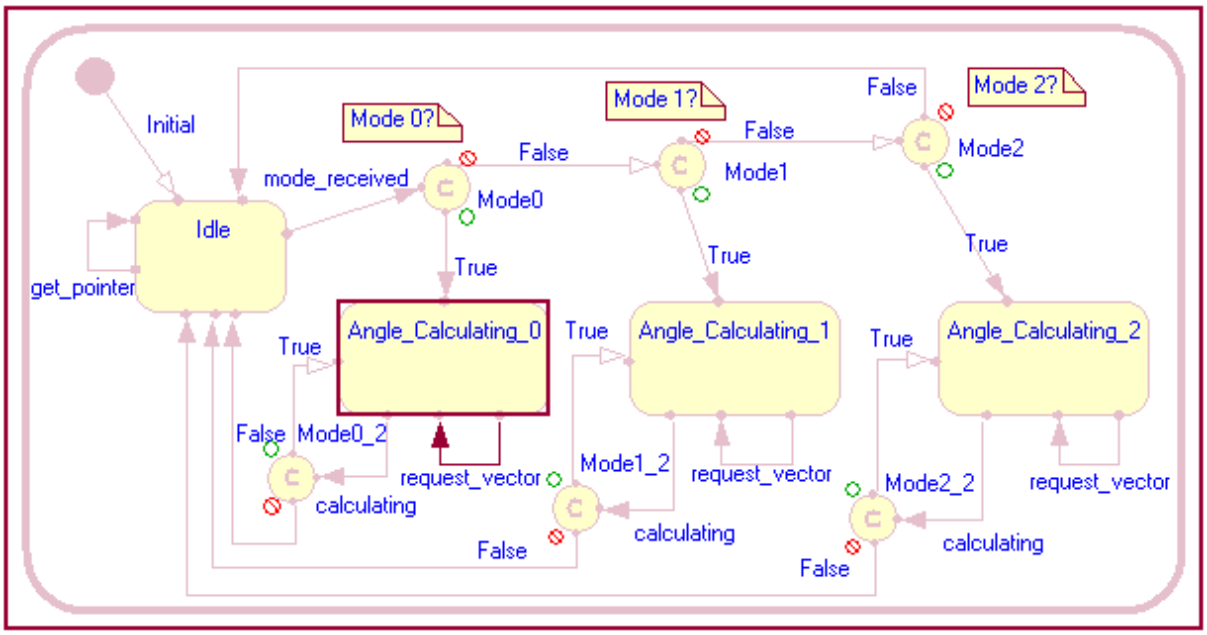

Figure 13. State monitor of the Elevator Controller during the simulation.

At this exactly moment, the model is running on operation mode 0 and is receiving the data from the simulator ("request_vector" transition). The other State Monitors will not be placed here because they have just one state, so it is always known where they are at any moment of the simulation. Beyond the State Monitors, it is possible to see the values received by the RRRT and the calculated command on the prompt window.

On the other hand, the simulator has an artificial horizon that answers to the actuator commands sent by the RRRT. It can be seen in Fig. 14.

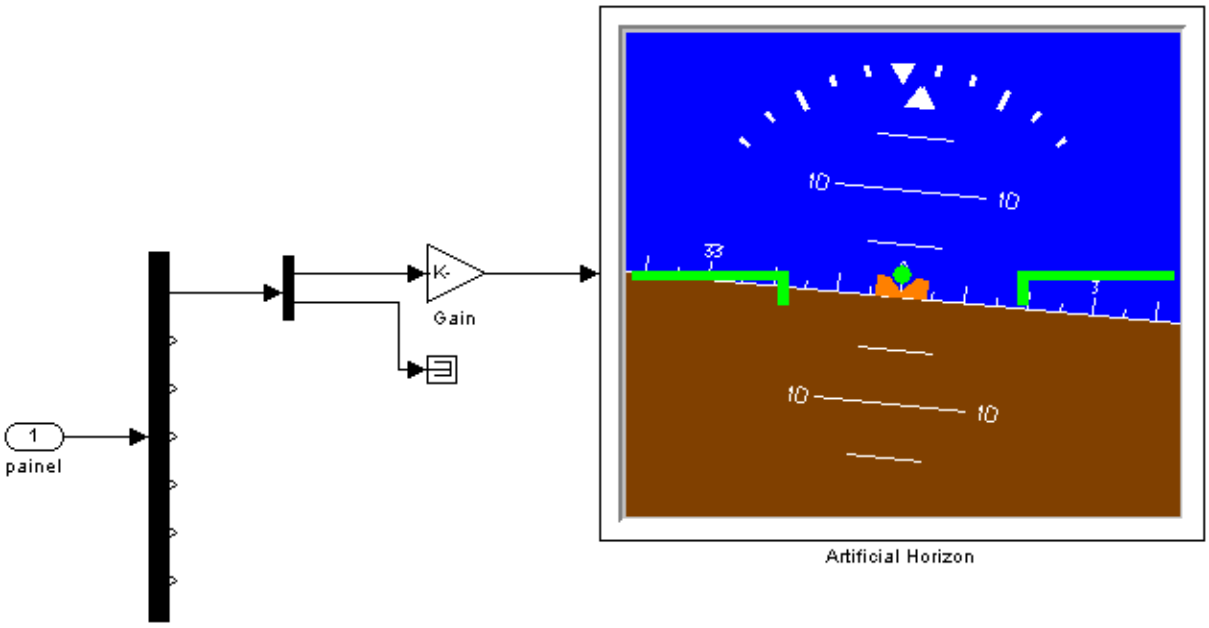

Figure 14. Artificial horizon of the flight simulator

The change of the operation mode modifies the way the artificial horizon answers the stick command because of the difference among the control laws used in each mode. In mode 2, the response is softer than in mode 1, which is softer than in mode 0.

By integrating the RRRT and the MatLab®, it is possible to verify not only the control embedded software, but also the controlled plant under operation. Without this integration, it would not be possible to simulate the feedback control system of the UAV with "real" data of flight. Furthermore, it is possible to test various types of control system by changing just some few lines of code in the RRRT model and making comparison among them.

5. CONCLUSION

In this paper a new approach to validating embedded software is introduced. The purpose of the approach is to modelling the embedded software using a CASE tool and integrating it with a dynamic simulation environment in order to allow the simulation of the feedback control system of the control software. With this idea it is possible to analyse both the embedded control software and the controlled system – the auto-pilot embedded software and the behaviour of the UAV.

The proposed approach is an intermediate step between using a single platform, such as Simulink®-Matlab®, to represent both the control system and controlled plant, and a "hardware on the loop" approach, where the control system is replaced by the real on-board computer which communicate with the flight simulator. In the single platform approach, only the control law is modelled, communication routines and other details of the software implementation are not considered. Therefore the kind of error that may be detected is limited. On the other hand, when comparing to the "hardware on the loop" approach, the proposed one offers as advantages the possibility of using all the debugging tools of the RRRT environment.

At present moment, in order to complete the approach, a study is being performed about the metrics that can be used to evaluate the code generated by the CASE tool. The purpose is to analyse whether the embedded software is well modelled or not. For this purpose, the test tool of RRRT – Rational® Test RealTime – should be used.

6. ACKNOWLEDGEMENTS

The authors would like to thank Flight Technologies, specially Nei Salis Brasil Neto and Benedito Carlos Oliveira Maciel, for gently providing the case-study of this paper and the infra-structure of their company. This research is supported by governmental agencies CAPES, FAPESP, CNPq and FINEP.

7. REFERENCES

Brooks Jr. and Frederick P., 1995, “The mythical man-month: essays on software engineering”, Ed. Addison-Wesley, Reading, USA, 322 p.t

Damaševičius, R. and Štuikys, V., 2004, “Application of the object-oriented principles for hardware and embedded system design”, Integration, the VLSI Journal, Vol. 38, No. 2, pp. 309-339.

Giese, H., Graf, J., Wirtz, G., 1999. "Closing the gap between object-oriented modeling of structure and behavior", Proceedings of the 2nd International Conference on the Unified Modeling Language, Colorado, 1999.

Iwu, F. *et. al.*, 2007, “Integrating safety and formal analyses using UML and PFS”, Reliability Engineering & System Safety, Vol. 92, No. 2, pp. 156-170.

Kimour, M.T. and Meslati, D., 2005, “Deriving objects from use cases in real-time embedded systems”, Information and Software Technology, Vol. 47, No. 8, pp. 533-541.

Lüttgen, G. and Mendler, M., 2001, “Statecharts: From Visual Syntax to Model-Theoretic Semantics”, in Workshop on Integrating Diagrammatic and Formal Specification Techniques (IDFST 2001), Vienna, Suíça, pp. 615-621.

Massink, M., Latella, D. and Gnesi, S., 2006, “On testing UML statecharts”, Journal of Logic and Algebraic Programming, Vol. 69, No. 1-2, pp. 1-74.

Palma, D.P., 2005, “Estudo sobre os aspectos de Desenvolvimento e Certificação de Software para Produtos Aeroespaciais de Emprego Militar”, 55f. Trabalho de Conclusão de Curso de Extensão – Instituto Tecnológico de Aeronáutica, São José dos Campos, Brazil.

Paludetto, M., Delatour, J., 1999. "UML et les réseaux de Petri – vers une sémantique des modèles dynamiques et une méthodologie de développement des systèmes temps réel" L’Object, vol. 5, n.3, pp.443-468.

Rumbaugh, J.; Jacobson, I.; Booch, G. 1999 “The Unified Modeling Language Reference Manual”, Ed. Addison-Wesley, New York, USA, 550 p.

7. RESPONSIBILITY NOTICE

The authors are the only responsible for the printed material included in this paper.