Modeling an Autonomous Helicopter and its aintenance Environment using Modelica

Johan Parmar

July 9, 1999

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

It does not matter what field you are working in, to develop new products is costly business. To decrease the large costs and the time it takes for a company to develop new products, computer simulations are many times helpful. The military industry has been using computer simulations for a number of years. Commercial industries have in recent years opened their eyes for this time- and cost-effective way to develop new products. The idea is to construct a model and simulate how for example it behaves, looks and so forth before you actually produce it.

In the WITAS project there is a need to simulate dynamic behavior of an autonomous aircraft, in this case a helicopter, within a virtual environment. Previously a simplified model developed by Pontus Lidman[1] was used. Because of its simplicity there was a need for a more complex, realistic helicopter model. There is also need to simulate a service environment station, the so called maintenance operating environment, where the helicopter can land and interact with a robot that can fuel the aircraft, load/unload cargo etc. A new modeling language called Modelica has been used in this thesis to develop these simulation models.

This thesis was done at the Programming Environments Laboratory (PELAB) [1] at the Department of Computer and Information Science, at Linköping University. Thesis supervisor has been Prof. Peter Fritzson.

1.2 Objective and problem statements

The main objective of this thesis is to design an object-oriented helicopter and its service environment simulation model in Modelica. Besides this, it also to build an interface between the helicopter model and an existing control system

[1] Former Master thesis student at Pelab, Linköping University

(developed in Ada). A secondary objective is to learn how to use the new modeling language Modelica and evaluate it.

These are the general problem statements:

- Modeling problems — What equations to use and how to express them in Modelica.
- Integration with control — What changes have to be made to integrate the model with the existing control system?
- Integration with service environment — Who are the actors?
- Integration with WISS — How are the models incooperated in to WISS?
- Performance requirements — Should the simulations run in real time?

Some of these statements have been dealt with to some degree by the author or a member in the Pelab group. The restrictions made in this thesis work can be read in the next section.

1.3 Constraints

Since this is a Master's degree thesis with limited time available some restrictions where made on the work.

- Some limitations were made on how realistic the models should be. The main focus has been on using Modelica as the modeling language.
- The simulation time of the models has not been required to be in real time.
- The maintenance environment has been made very basic due to shortage of time.

1.4 Report overview

This report consists of 9 chapters and 2 appendices. The first Chapter gives an introduction to the work and the problems to be dealt with. Chapter 2 describes the WITAS project. First an introduction is given followed by a description of the system. In Chapter 3 an overview is given of the modeling language Modelica which is used in this thesis, focusing on aspects relevant for this thesis. Chapter 4 presents different helicopter models. First a comparison between the models is made, followed by a description of each model. A description of helicopter control system is given in Chapter 5. Chapter 6 presents a robot model and its control system and in Chapter 7 the maintenance operating environment is introduced where the helicopter and the robot is integrated. Finally Chapter 8 gives a short introduction to different tools used in the development of the different models while Chapter 9 discusses the work done in this thesis and future work.

In appendix A a connection diagram of the different parts of the object-oriented helicopter is presented. Appendix B includes some of the code that represents the SolidWorks generated helicopter model.

Chapter 2

The WITAS project

2.1 Introduction

The WITAS project [2] concerns research in the area of autonomous aircrafts and other autonomous systems. This project is sponsored by the Knuth & Alice Wallenberg foundation. WITAS stands for *Wallenberg laboratory for research on Information Technology and Autonomous Systems.*

The WITAS laboratory is a group of researchers at Linköping University. This group is divided into four sub-groups: autonomy, software, vision and hardware. The autonomy group is lead by Dr. Patrick Doherty and Dr. Dimiter Driankov, the software group by Prof. Peter Fritzson, the vision group by Prof. Gösta Granlund and the hardware group by Prof. Krzysztof Kuchcinski. The coordinator for the whole project is Prof. Erik Sandewall.

Below follows a short overview of the project. The parts that concern this thesis are emphasized. For a more thorough discussion see Doherty [3].

2.2 Design approach

During the first 3 years of the WITAS project, the major objectives of the project have been to develop a virtual computer-simulated environment including a command and control architecture combined with an active vision system that supports an UAV (*Unmanned Autonomous Vehicle*) in full or semi autonomous mode. The virtual operational environment is over an area that includes different geographical types of terrain. It also contains traffic networks including tunnels and bridges. The tasks of the UAV can be to monitor traffic situations, identify and follow individual cars and to provide assistance in emergency situations.

Because of the size of the project and the number of people involved, there was a need for a design and evaluation framework (see figure 2.1) together with a corresponding software system (WISS, see section below). The evaluation framework consists of the following iterative processes :

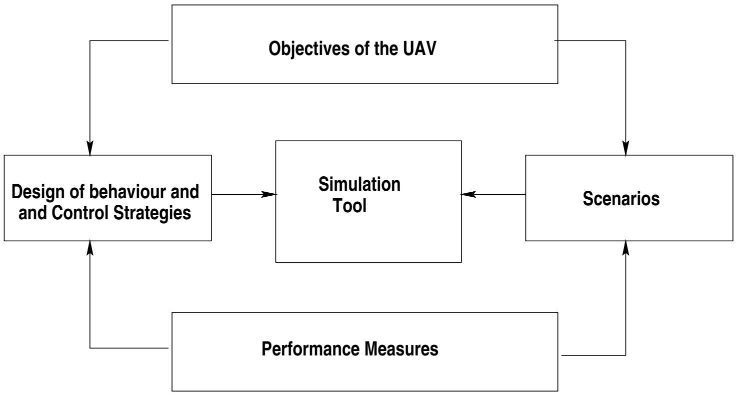


Figure 2.1: Iterative design and evaluation framework.

1. Determine the major objectives for the UAV to achieve.
2. Choose an operational environment that is sufficiently complex to provide a challenging testbed to test if the objectives are met.
3. Choose or design a hybrid deliberative/reactive architecture for the command and control aspects of the UAV in addition to a sensory platform suitable for the chosen operational environment and for achieving the objectives.
4. Construct one or more scenarios in the operational environment to test one or more functionalities pertaining to the UAV architecture and sensory platform. The scenarios should be setup not only to test individual UAV functionality, but with the major objectives in mind.
5. Evaluate the UAV's success in achieving mission goals in the scenarios through simulation.
6. Iterate on steps 2-5, occasionally adjusting step 1.

(Doherty [3])

The simulation environment is constructed in a modular way, to allow software modules to be replaced by more advanced modules either in software or hardware.

2.2.1 WMES

WMES stands for *WITAS Maintenance Environment System.* The idea is to provide autonomous service environments for UAV's. Possible tasks can be:

- Refueling of the UAV;
- loading and unloading of cargo;
- exchange of components in the UAV;
- minor repairs.

A possible scenario can be that the UAV lands on a landing platform that might be situated on a truck. A mobile robot with a refueling hose is presumed to be mounted on the truck. When the UAV has landed, the robot will grasp the hose and move it to the fueling tank opening. The hose is connected to the tank opening and refueling is started. When the refueling is done the robot and hose return to their original positions and the UAV takes off.

The WMES system for the Maintenance Operational Environment is primarily a system for modeling and controlling three-dimensional mechanical, dynamic systems. Interaction between different UAVs, between UAVs and service robot, collision detection and guidance are some important issues.

2.2.2 WISS

The WISS (see figure 2.2) is the backbone of the system. It provides a software infrastructure used in the design and evaluation of *Command and Control Software* for UAVs. The WISS contains a *Modular Simulation Platform* (MSP) and an *Intelligent Vehicle Control Architecture* (IVCA). MSP is a distributed, model-based simulation platform that allows several different agents, each with its own individual command and control architecture to be connected to the system. The communication between different agents and the simulation server is handled by TCP/IP which is practical if different parts of the system are run on different machines.

2.2.3 MSP

As mentioned above, MSP stands for *Modular Simulation Platform.* It consists of the following parts:

1. The *Simulation Server* which basically is a communication interface to the actual simulation and rendering software. It registers and initializes agents that want to connect to the MSP and controls the clock progression of these and their synchronization with the main (i.e simulation) clock.

2. The *Object State Simulator* receives actuation and sensory control commands from all agents that take part in the simulation. There is a mathematical model for each agent that characterizes its physical behavior. Such behavior can be described in modeling languages like Modelica which will be discussed in the next chapter. In this part the actual simulation takes place. Furthermore it also sends information/commands to the rendering machinery.

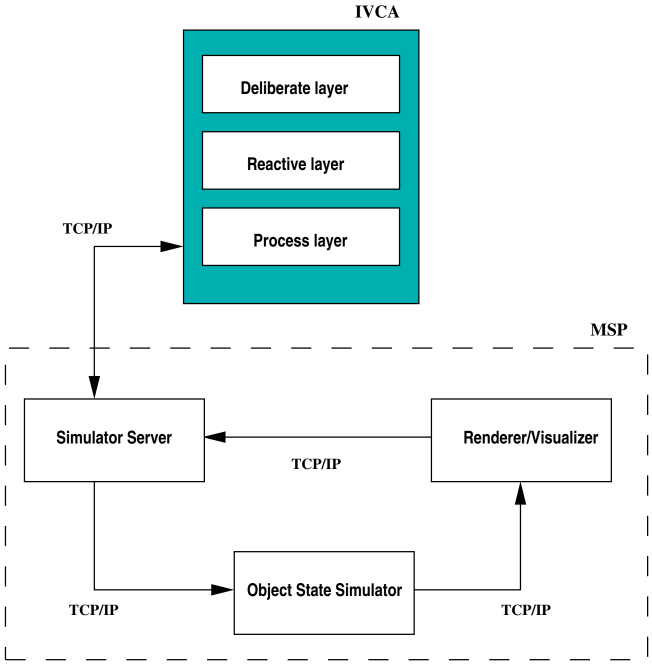


Figure 2.2: WISS overview.

3. The *Renderer/Visualizer* contains a SGI platform for rendering and visualizing the objects in the object state simulator. It also contains an image server for transferring or/and transforming image streams to other or participating agents.

2.2.4 IVCA

IVCA is built up using a three layered architecture, where each layer is a separate computational process.

- *The deliberative layer* produces plans for the reactive layer, in order to plan the movements of the UAV. It also checks whether or not a plan was successful and makes decisions about the continued operation of the UAV.
- *The reactive layer* is responsible for producing predetermined responses for specific events, e.g. if the chased vehicle goes into a tunnel and out of sight for the UAV.
- *The process layer* is responsible for calculating the feedback control loops connecting the sensing with actuation. It responds quickly to changes in the environment.

If a lower layer in the architecture can't handle an arised situation, the layer above is activated.

2.3 Goals

The present aim of the project is to demonstrate, before the end of year 2003, an airborne computer system which can make rational decisions concerning the continued flight of an aircraft based on vision sensors, radio communication and geographical information system. The development of different parts of the system is combined with research in the areas of information technology that are represented in the project.

Chapter 3

Modelica

3.1 Introduction

Modelica [4] is an effort to combine the concepts of object-oriented and non-causal modeling with proven technology from existing languages into a new, more general modeling and simulation language. The general equation-based physical modeling capabilities of Modelica makes modeling easier between different areas such as electronics, hydraulics, mechanics, etc.

So what makes Modelica so powerful compared to other existing modeling languages? The language consists of several strong features which are summarized and explained below:

- *Object-oriented modeling* — Using object oriented concepts like classes, objects, inheritance, hierarchical structuring, provides reuse of modeling knowledge and structure to handle complexity of large systems.
- *Non-causal modeling* — This means that equations (i.e. differential and algebraic equations) are used, not assignments [1] when building up the models. Then the Modelica compiler automatically solves out the correct variable/variables regarding the problem. This enables physical modeling; being able to directly reflect physical components and their interconnections in the model.
- *Multi-domain modeling* — Modelica makes it possible to connect models of systems from different physical domains like electronics, hydraulics, mechanics, etc.
- *General type system* — Modelica unifies object-orientation, multiple inheritance and templates within a single class construct. This makes the Modelica language easier to use while keeping expressibility.

[1] You can still use assignments while using Modelica if necessary. Assignment statements can be placed in the algorithms section of function declarations (see section 3.4)

3.2 A Modelica example

To get a better picture of the Modelica language and some of its features, we will go through a complete Modelica example. As an example we will take the electrical circuit in figure 3.1.

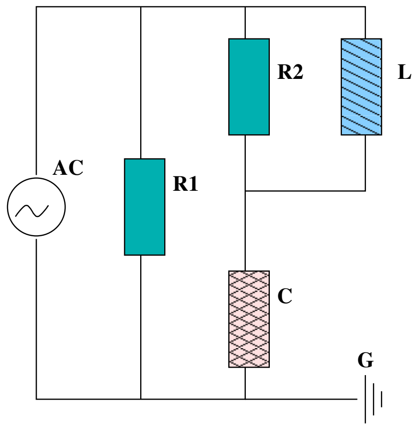


Figure 3.1: Electrical circuit.

A complete Modelica description of the electrical circuit in figure 3.1 looks like:

```
model Eletric_circuit
  Resistor R1(R = 10);
  Resistor R2(R = 200);
  Capacitor C(C = 0.05);
  Inductor L(L = 0.05);
  Vsource AC;
  Ground G;
equation
  connect(AC.p,R1.p);
  connect(R1.n,AC.n);
  connect(R1.p,R2.p);
  connect(R2.n,C.p);
  connect(C.n,AC.n );
  connect(R2.p,L.p);
  connect(L.n,C.p);
  connect(AC.n,G.p);
end Eletric_circuit;
```

The Electric_circuit is declared as a model. It includes some object declarations and an equation section. In this model all the statements in the equation section are connect statements. Connect statements are used to connect instances of connection classes. The model specifies components and connections between the components. As you can see, the physical structure of the circuit is preserved in the model. The statement,

```
Capacitor C(C = 0.05);
```

declares a component C of a class Capacitor and sets the default value of the capacitance C to 0.05.

To start off with the simplest component Ground,

```
model Ground ''Ground''
  Pin p;
equation
  p.v = 0;
end Ground;
```

it consists of a an object called p of class Pin and there is a single equation that states that the value of v for this pin is set to zero. v here in our example stands for voltage. The object Pin is defined as follows

```
connector Pin
  Voltage v;
  flow Current i;
end Pin;
```

Notice that this object is declared as an connector instead of a model as we used earlier. Both are examples of restricted classes that will be discussed in section 3.3.

The Pin class has two attributes, v and i that are of type *Voltage* resp. *Current*. If for example, we have a connection `connect(pin1, pin2);` where $pin1$ and $pin2$ are of connector class Pin, this produces two equations :

```
pin1.v = pin2.v
pin1.i + pin2.i = 0
```

The first equation states that voltage on two branches that are connected should be equal. The second corresponds to Kirchoff's law that states that currents sum to zero at the node. The prefix **flow** in front of the type *Current* determines i to be a flow variable that generates these sum-to-zero equations. When flow is not specified equality equations are generated.

Kirchoff's Voltage law and Kirchoff's Current law seem to be enough to describe any physical system in any domain.

A connector must contain all quantities needed for the interaction. For this example we need the quantities *Voltage* and *Current*.

```
type Voltage = Real(unit = ''V'');
type Current = Real(unit = ''A'');
```

Real is a predefined variable type. Notice that the unit specifications can be used by a tool for automatic unit checking.

Most electrical components have two pins. Consequently it might be useful to define an abstract class which holds this concept. In Modelica you do this by declaring your model as partial model. Observe that this class can never be instantiated, only inherited by other classes.

```
partial model TwoPin "Superclass of elements
                      with two electrical
                      pins";
  Pin p, n;
  Voltage v;
  Current i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = i.p;
end TwoPin;
```

The model has two pins (p and n), a quantity v that defines the voltage difference between the pins, and a quantity i that defines the current from pin p to n. This can be seen in the equations in the equation section.

Now to complete our example, definitions of resistor, voltage source, capacitor and inductor are presented.

First, the Resistor model which contains Ohm's law:

```
model Resistor "Ideal el. resistor"
  extends TwoPin;
  parameter Real R(unit = "Ohm") "Resistance";
equation
  R * i = v;
end Resistor;
```

$der(v)$ stands for the time derivative of v. The prefix **parameter** indicates that the the value of the variable is constant during simulation runs.

The Inductor model contains an equation that voltage is proportional to the derivation of i:

```
model Inductor "Ideal el. Inductor"
  extends TwoPin;
  parameter Real L(unit = "H") "Inductance";
equation
  L * der(i) = v;
end Inductor;
```

The Vsource model specifies a sine-wave voltage source:

```
model Vsource "Sin-wave voltage source"
  extends TwoPin;
  parameter Voltage VA = 220 "Amplitude";
  parameter Real f(unit = "f") = 50 "frequency";
  constant Real PI = 3.141592653589793;
equation
  v = VA * sin(2 * PI * f * time);
end Vsource;
```

Finally, the Capacitor model specifies that the derivative of voltage v is proportional to the current i:

```
model Capacitor "Ideal el. capacitor "
  extends TwoPin;
  parameter Real C(unit = "F") "Capacitance" ;
equation
  C * der(v) = i;
end Capacitor;
```

3.3 Restricted classes

Classes are the basic structuring elements in object-oriented languages like Modelica. In Modelica there are also seven kinds of restricted classes which in most cases are identical to class, except for some restrictions on their use. This not only makes it easier for the model developer, but also in the construction of Modelica translators due to the fact that only the semantics and syntax of a class has to be considered together with some additional checks on restricted classes.

- The normal way to specify a simulation model is to use the restricted class **model**.
- A **connector** class is a convenient way to gather several attributes of a physical connection together.
- A **record** is structured data type containing a number of fields. Records do not contain any equations or algorithms.
- A **block** is a class with fixed causality. This means that the causal direction of the data is predefined by using the prefixes *input* and *output*.
- The restricted class **type** gives the possibility to derive new classes with certain modified attributes. User-defined types can be declared as `type Matrix33 = Real[3,3] "3 × 3 matrix";` where Matrix33 is declared as a two dimensional array object. Note that Modelica has built-in types such as *Real*, *Integer*, *Boolean* and *String*. Types can be used in connect statements.

- A **package** may only contain declarations of constants and classes. The idea is to encapsulate related classes into a package which can be inherited while dealing with complex models.

- A **function** has fixed causality similar to block. The body of a function contains one algorithm section and no equation section. A function is side-effect free, similar to functions in declarative programming languages. Functions can be called within expressions by positional parameters in the normal way.

Note that type, package and function are not just restricted classes — there are some extra semantics attached to them. It is not possible to change `type Matrix33 = Real[3,3]` to `class Matrix33 = Real[3,3]`. For the other restricted classes it is true that for a valid model for example, record can be replaced by the keyword class and the model behavior will be maintained. This is because the restrictions imposed by such a restricted class are fulfilled by a valid model.

3.4 Equations, algorithms and functions

Equations are the basic building blocks of Modelica and not as in most other modeling and programming languages, assignment statements. This gives more flexibility while modeling and enables physical modeling, making it possible to structure simulation models in the same way as the physical world. A component description can be used with different causalities depending on how the component is connected. Still, sometimes it is useful to use assignment statements. Modelica provides two ways to assign statements. Either you call external functions containing assignments or a Modelica algorithm. The Modelica algorithm is better to use because it is designed to be much more secure than calling external functions. A Modelica algorithm is a function in the mathematical sense. Thus, the same inputs will produce the same outputs.

Functions are restricted classes as specified. Since a function has fixed causality, the input and the output arguments are explicitly marked with the keywords input and output. No equations are allowed in a function declaration and only one algorithm section.

As mentioned earlier it is possible to call functions that are defined outside the Modelica language. Such functions are made accessible by providing an external declaration, e.g below:

```
function getfrotor
 input Real t;
 input Real x;
 output Real v;
external
end getfrotor;
```

External function calls are used in the communication channel between the control system and the model for the WITAS UAV application. The external function above is to get the new desired value of the force (frotor) that is applied in the center of the hub from the control system.

3.5 The MBS library

Modelica makes it possible to model large and complex systems which involve different engineering domains. To make the modeling process easier, several class-libraries from different domains have been developed. In this thesis, a library called *Multi-Body System* library has been used. This library is used in modeling three dimensional mechanical systems. An important feature in the library is the efficient treatment of joint locking and unlocking.

The library consists of the following elements:

- *Inertial systems* — Exactly one inertial system must be present in a model using the MBS library. A joint has to be connected to this system. All vectors, such as rotational vectors of each body in the system are given in their home-position with respect to the inertial frame.

- *Rigid bodies* — There are several classes to define rigid bodies. From the class definition, the mass, center of mass and inertia tensor are defined. The shape can be used in animations. At the moment only basic geometric shapes like boxes, cones and so on are supported. Each body object has at least two connectors where it can be connected to other objects. More attachment points can be introduced by using massless bar elements rigidly connected together.

- *Tree joint classes with state variables* — There are several classes for joints which define state variables. During the simulation of a model, the state variables of the system are obtained from the generalized coordinates of these joints. The one-degree-of-freedom joints can be locked or unlocked during a movement.

- *Tree joint classes without state variables* — There are classes for joints without state variables which are otherwise identical to those with state variables. These joints are used for closed kinematic loops or inverse models [5].

- *Cut joint classes* — These are used for breaking closed kinematic loops.

- *Force elements* — Force elements can be connected between two points of distinct bodies or joints. It is not possible to connect a force element to another. It is easy for the user to introduce new force elements as subclasses of existing ones. The gravitational forces on a multi-body system can easily be taken into account by setting a parameter in the inertial system (i.e. object of the class Inertial)

- *Sensor elements* — A sensor element can be attached between two distinct bodies or joints. It is used for calculating kinematic quantities like distance, speed and so forth between the two points.

The elements from the MBS library have to be connected according the following rules

1. Tree joint objects (with and without state variables), body objects and the inertial system have to be connected together in such a way that a cut A of an object is always connected to a cut B of an object. The connection structure has to form a tree with the inertial system as the root.

2. Cut joint, force and sensor objects have to always be connected between two cuts of a tree joint, body, or inertial system object.

The MBS library can be combined with three sub-libraries called *Advanced*, *Drive* and *Train*. Sub-library *Advanced* contains elements for advanced usage. This library have not been used in thesis, therefore I will not go into more details about it. Sub-libraries *Train* and *Drive* contain one dimensional rotational and translational components from the drive train library which can be used for multi-body systems.

The classes from the MBS library are explained more thoroughly in Larsson [6].

3.6 Other Modelica features

3.6.1 Hybrid modeling

Modelica is quite capable of handling hybrid systems. Basic elements have been implemented to be able to model, e.g., sampled data systems and to synchronize the discrete and continuous parts with each other. A new technique is used to handle variable structure systems, such as ideal diodes, ideal thyristors, Coulomb friction and phase changes.

3.6.2 Graphical representation of models

Modelica offers the possibility to build and represent models using a Graphical User Interface. Some additional information is needed to represent icons, graphical layout and connections. Modelica supports property lists that can be used to store this information. Each component in a model can have list designated by the keyword **annotation**.

An example of this representation can be seen in figure 8.1.

Chapter 4

Helicopter models

4.1 Design history

Except for the helicopter models that have been developed during this thesis work, an earlier Modelica model [7] was developed by Pontus Lidman at the beginning of the WITAS project. The current models have been produced in the following order:

1. Pontus Lidman's model ($Heli_1$)

2. Flat model ($Heli_2$)

3. Object-oriented model ($Heli_3$)

4. SolidWorks-generated model ($Heli_4$)

$Heli_1$ defines a simple helicopter model where the input parameters are the desired velocity in x- and y-direction, desired altitude and desired yaw angle. From these parameters the position, heading, roll, pitch are calculated in a straight forward manner. $Heli_2$ equations have been derived while considering how a real helicopter flies. Therefore it also behaves more like a helicopter. In $Heli_3$ and $Heli_4$ the equations are automatically derived by the compiler. Both these models use the MBS library which makes them more dynamic and realistic in one sense. Different rigid bodies are connected with appropriate joints. For example both these models have a top-rotor and tail-rotor that moves and also a landing gear that secures a safe landing. The difference between these models are that $Heli_4$ is designed in SolidWorks and that $Heli_3$ is developed to be object-oriented. $Heli_3$ is built up using only simple geometrical shapes while $Heli_4$ has a more complex shape. In the following sections $Heli_2$ and $Heli_3$ will be described more in detail. Also $Heli_4$ will be presented.

4.2 Flat helicopter model

The flat helicopter model [8, 9] developed by Erik Skarman[1] and Bourhane Kadmiry[2], consists of a set of equations. The following ten equations describe the physical behavior of the helicopter.

1. $m\frac{dv_x}{dt} - m(\omega_z v_y - \omega_y v_z) = F_{rotor}\sin\beta_y - d_x(v_x) - mF_{F0}^{13}g$
2. $m\frac{dv_y}{dt} - m(\omega_x v_z - \omega_z v_x) = F_{rotor}\sin\beta_x - d_y(v_y) - mF_{F0}^{23}g$
3. $m\frac{dv_z}{dt} - m(\omega_y v_x - \omega_x v_y) = F_{rotor}\cos\beta_y\cos\beta_x - d_z(v_z) - mF_{F0}^{33}g$
4. $I_{xx}\frac{d\omega_x}{dt} - (I_{yy} - I_{zz})\omega_y\omega_z = rF_{rotor}\sin\beta_x - d_{\omega x}(\omega_x)$
5. $I_{yy}\frac{d\omega_y}{dt} - (I_{zz} - I_{xx})\omega_z\omega_x = rF_{rotor}\sin\beta_y - d_{\omega y}(\omega_y)$
6. $I_{zz}\frac{d\omega_z}{dt} - (I_{xx} - I_{yy})\omega_x\omega_y = -M_{rotor} - L_{tail}F_{tail} - d_{\omega z}(\omega_z)$
7. $J\frac{d\Omega}{dt} = M_{rotor} - D(\Theta_0, \Omega)$
8. $\frac{dx}{dt} = F_{F0}^T v_F$
9. $\frac{d\zeta}{dt} = F_{F0}^T\omega_F + \frac{d\sigma}{dt}(\zeta, \omega_F)$
10. $F_{F0} = e^{\zeta}$

The first three equations are force equations for the body of the helicopter in x-, y- and z-direction.The forces that have been taken into account for the model are: lift-force from the rotor, aerodynamic forces and the gravity force. Note that the force from the top rotor is appropriately divided into components considering the tilted rotor plane with angles β_y and β_x.

Equations four to six are moment equations for the cabin of the helicopter. The moments are produced by the forces F_{rotor} and F_{tail}. The gravity force and the aerodynamic forces do not give rise to any moments because they act at the center of mass.

Equation number seven is the moment equation for the rotor.

The eighth equation is in fact three scalar equations which when integrated represent the integration of velocity to position (indexed by 0 because it is done in the inertial frame).

The ninth equation which in fact is also three scalar equations, is a differential equation of the rotational state of the helicopter. The term $\frac{d\sigma}{dt}(\zeta, \omega_F)$ accounts for the non commutativity of rotational motions. This is represented here by the so called turning vector ζ.

Equation number ten relates this turning vector to the transformation between the inertial (0) and the body-fixed (F) frame.

---
[1] Expert in the WITAS project and engineer at *SAAB Dynamics*

[2] Phd student at Linköping University

The turning vector which represents the rotational position of the helicopter is not used in the control laws. Instead the notion of Euler angles (yaw, pitch and roll) is applied. These can be derived from the following equations:

$$\frac{dE}{dt} = H^{-1}\omega_F$$
$$F_{F0} = F_{F0}(E)$$

The vector E stands for the Euler angles (Ψ, Θ, Φ). H is a so called hybrid matrix , which almost is the unit matrix if Θ and Φ are small. The matrix can also become singular if the roll angle is equal to 90 degrees.

Equations number one to ten have been implemented by the author as a flat model in Modelica in a rather straight-forward manner (because of the ability of non-causal, equation-based modeling).

4.3 The object-oriented helicopter model

The motivation to make an object oriented helicopter model is to make it more convenient if you want to replace certain parts of the helicopter. For example if you want to change the top-rotor (to a more detailed model of the top rotor) of the helicopter it should be easy to do that. In order to develop object-oriented helicopter model, following classes have been designed by the author: *Helicopter*, *Rotor*, *Top_ Rotor*, *Tail_ Rotor*, *Heli_ Body*, *One_ Leg_ Landing_ Gear* and *Door*. Additionally a *Ground* class has been implemented so that the helicopter can land.

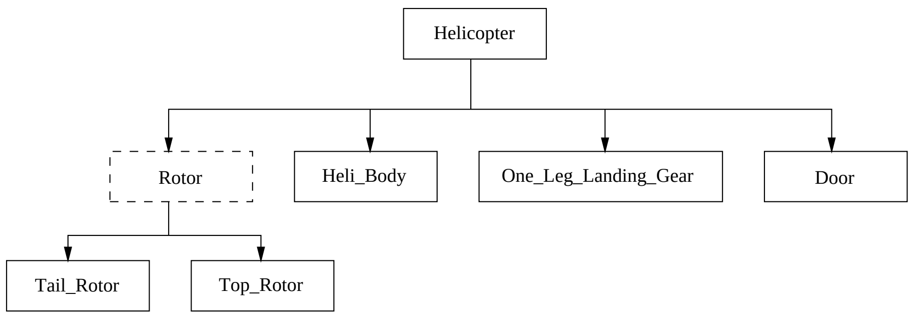


Figure 4.1: Different classes used in the helicopter model.

In figure 4.1, the relation between the classes can be seen. Instances of the classes *Tail_ Rotor*, *Top_ Rotor*, *Heli_ Body*, *One_ Leg_ Landing_ Gear* and *Door* are parts in the "main " class *Helicopter*. Each class will be explained more below:

model Helicopter

```
model Helicopter
```

```
Inertial wc;
Door door;
One_Leg_Landing_Gear stand1;
One_Leg_Landing_Gear stand2;
Heli_Body body;
Tail_Rotor tail;
Top_Rotor top; // (betax=BetaX,betay=BetaY);
ExtForce tailforce;
ExtForce topforce;
Ground g1, g2, g3, g4;
my_FreeCardan2S free;
```

Different objects are instantiated. Some of the classes (which has been developed by the author) will be explained below. The class ExtForce is used to instantiate the external forces topforce and tailforce . These are applied on the helicopter according to figure 5.2. The class Inertial has already been explained. The my _FreeCardan2S object is to give the helicopter six degrees of freedom.

```
//visualization of a ground body
BodyV visualground (mass=50, r=[0,0,0],
                    r0=[-50,-0.1,0],
                    I11=1.5, I22=1.5, I33=1.5,
                    Shape="box",
                    Size=[100,0.1,70],
                    Material=[1,0,0,0.5]);
Bar b1 (r = [1.3,0.15,0.07]);
Bar b2 (r = [0.3,0.4,0]);
Bar b3 (r = [0.3,-0.3,0]);
Bar b4 (r = [0,0,0.3]);
Bar b5 (r = [0,0,-0.3]);
Bar b6 (r = [-0.4,0,0]);
Bar b7 (r = [0.4,0,0]);
Bar b8 (r = [-0.4,0,0]);
Bar b9 (r = [0.4,0,0]);
Bar b10 (r = [0.3,-0.15,0.34]);
Bar b11 (r = [0.3103,0.4,0.0007]);
Bar b12 (r = [1.5,-0.0191,0.0007]);
```

To be able to see some ground when the helicopter lands, an object called visualground has been implemented. The several Bar objects are used in putting the helicopter together. They are massless which means that they have any mass and inertia.

```
//input
Real BetaX, BetaY, Frotor, Ftail, Mrotor;
// output
Real P[3], V[3], Omega[3], Euler[3],
```

```
     V0[3], A[3];
Real Chi, Gamma, f, Az;
```

The output and input signals are defined here. As input you get angles β_y and β_x and the two forces F_{rotor} and F_{tail} which are explained in section 5.1. The output are the helicopter's position, velocity, acceleration, Euler angles and rotational velocity .

```
// change coordinate system matrix
Real MoSkar[3,3] = [-1,0,0;0,0,-1;0,-1,0];
```

Because the difference on how the coordinate systems are defined in the control system and the model, a transformation matrix MoSkar is implemented.

```
// help variables
Real ntail[3], ntop[3];
equation
  BetaX = -getbetax(Time, 0.0);
  BetaY = -getbetay(Time, 0.0);
  Frotor = -getfrotor(Time, 0.0);
  Mrotor = -getmrotor(Time, 0.0);
  Ftail = -getftail(Time, 0.0);
```

The input values are defined by using the external function calls; getbetax, getbetay, getfrotor, getmrotor and getftail.

```
//setting qrefs for toprotor
top.betax = BetaX;
top.betay = BetaY;
//change of coordinate system and rotation
  matrix to Euler angles
P = MoSkar * free.rrela;
V = MoSkar * free.Srel * free.vrela;
V0 = MoSkar * free.vrela;
Omega = MoSkar * free.wrelb;
//calculating euler angles from free.Srel
Euler[1] = atan2(free.Srel[3,2],free.Srel[2,2]);
Euler[2] = -asin(free.Srel[1,2]);
Euler[3]= atan2(free.Srel[1,3],free.Srel[1,1]);
```

The output signals are transformed in to the coordinate system used by the control system. The Euler angles are calculated from the rotation matrix Srel defined in the class my_freeCardan2S.

```
// Chi,Gamma
Chi = atan2(V0[2],V0[1]);
Gamma = atan2(V0[3],sqrt(V0[2]^2+V0[1]^2));
```

```
 // connection statements
connect(wc.b,free.a);
connect(free.b,body.a);
connect(free.b,b1.a);
connect(b1.b,tail.a);
connect(free.b, b2.a);
connect(b2.b, top.a);
connect(free.b,b3.a);
connect(b3.b,b4.a);
connect(b3.b,b5.a);
connect(b4.b,stand1.a);
connect(b5.b,stand2.a);
connect(free.b,b11.a);
connect(b11.b,topforce.b);
connect(free.b,b12.a);
connect(b12.b,tailforce.b);
ntail = {0,0,Ftail};
ntop = {Frotor * sin(BetaY),
        Frotor * cos(BetaX) * cos(BetaY),
        Frotor * sin(BetaX)};
tailforce.fb = ntail;
topforce.fb = -ntop;
```

The two forces supposedly created by the helicopter are defined here. The top force depends on how the rotor is tilted (which is defined by the angles β_x and β_y) while the tailforce acts only in one direction .

```
//connections for the ground
connect(stand1.b,b6.a);
connect(stand1.b,b7.a);
connect(b6.b,g1.b);
connect(b7.b,g2.b);
connect(wc.b,g1.a);
connect(wc.b,g2.a);
connect(stand2.b,b8.a);
connect(stand2.b,b9.a);
connect(b8.b,g3.b);
connect(b9.b,g4.b);
connect(wc.b,g3.a);
connect(wc.b,g4.a);
// connections for the door
connect(free.b,b10.a);
connect(b10.b,door.a);
```

Several connect statements are used in putting together the helicopter model. In appendix A there is a diagram on how the different parts are connected together.

The external forces tailforce and topforce are connected to appropriate points on the helicopter.

```
// when to open the door
door.D.qref = if (g1.s>1) then 0
              else -1.57;
```

This condition defines when the door should open. If the distance between the helicopter landing gear and the ground is less then 1 meter, the door will open. Later on this will be controlled with some input signal.

```
 //connect visualground to inertial
 connect(wc.b,visualground.a);
 A = MoSkar * free.Srel * free.arela;
 Az = A[3];
 f = extcomm (Time, P[1], P[2], P[3],
             Euler[1], Euler[2], Euler[3],
             Omega[1], Omega[2],Omega[3],
             V[1], V[2], V[3],
             VO[1], VO[2], VO[3],
             Chi, Gamma, Az,
             free.Srel[1,1], free.Srel[3,1],
             free.Srel[1,3], free.Srel[3,3],
             free.Srel[1,2], free.Srel[3,2],
             free.Srel[2,2]);
end Helicopter;
```

The information acquired by the control system is sent using the external function extcomm.

model Heli_Body

This class defines the body of the helicopter. In this model it is simply a box with a cylinder attached on it.

```
model Heli_Body
  extends MbsOneCut;
  BodyV Heli_bodypart1(mass = 40,
                      I11 = 12.5,I22 = 15,I33 = 15,
                      r = [0.3,0,0],
                      Shape = "box",
                      Size = [0.6,0.6,0.6],
                      Material = [1,0,0,0.5]);
  BodyV Heli_bodypart2(mass = 5,
                      I11 = 0.025,I22 = 0.279,I33 = 0,
                      r = [0.3,0,0],
                      Shape = "cylinder",
```

```
                              Size = [0.8,0.1,0.1],
                              r0 = [0.6,0.15,0],
                              Material = [1,0,0,0.5]);
equation
  connect(a,Heli_bodypart1.a);
  connect(a,Heli_bodypart2.a);
end Heli_Body;
```

Bodypart1 is the box and bodypart2 is the cylinder.

model One_Leg_Landing_Gear

In this model one leg of a helicopter stand is defined. As you can see in figure 4.2 it is a damper and a spring connected in parallel between the helicopter and the visualized stand.

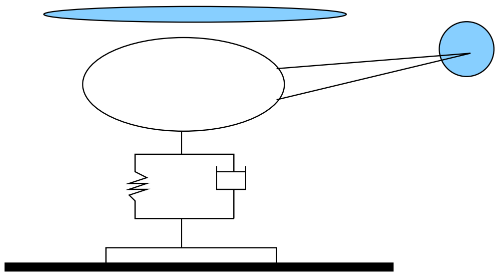


Figure 4.2: Helicopter landing gear.

```
model One_Leg_Landing_Gear
  extends MbsTwoCut;
  parameter Real c = 2000;
  parameter Real srel0 = 0.4;
  parameter Real d = 1000;
```

Some parameters that define spring constant, unstretched spring length and damping constant.

```
  BodyV stand (mass = 1,
               I11 = 0.02,I22 = 0.054,I33 = 0.054,
               r0 = [-0.4,0,0],
               Shape = "box",
               Size = [0.8,0.1,0.1],
               Material = [1,0,0,0.5]);
  PrismaticS p (n = [0,-1,0]);
  TransSpring s (c = c, srel0 = srel0, d = d);
equation
  connect(a,p.a);
```

```
    connect(p.b,stand.a);
    connect(p.pTrans,s.nTrans);
    connect(p.bearing,s.pTrans);
    connect(b,p.b);
  end One_Leg_Landing_Gear;
```

model Door

The class Door is implemented because there is a need to visualize when the robot picks up the cargo from the helicopter. The door is built up by a body and an engine which makes it possible to open the door.

```
model Door
  extends MbsOneCut;
  BodyV Door (mass = 0.001,
              r = [0.1,0.1,0.06],
              Shape = "box",
              Size = [0.2,0.2,0.01],
              Material = [1,0,0,0.5]);
  RevoluteS R (n = [0,1,0]);
  GearT1I D (k = 10, d = 10);
```

GearT1I is a class used for deriving an engine object.

```
equation
  connect(D.nDrive,R.pDrive);
  connect(a,R.a);
  connect(R.b,Door.a);
end Door;
```

model Rotor

This Rotor model is an abstract class, i.e. it is not complete. It can be inherited but not instantiated. In the code, the keyword **partial** is written in front of keyword **model**.

```
partial model Rotor
  extends MbsOneCut;
end Rotor;
```

model Tail_Rotor

The Tail_Rotor model extends the Rotor model, i.e it is a specialization of Rotor. An engine $D1$ is connected to the joint $R3$ so that the rotor can rotate. Tail is body of the rotor.

```
model Tail_Rotor
  extends Rotor;
```

```
    RevoluteS R3 (n = [0,0,1]);
    BodyV Tail (mass = 0.5,
                I11 = 0.002, I33 = 0.002,
                Shape = "box",
                Size = [0.05,0.3,0.02],
                r0 = [0;0.025;0],
                Material = [1,0,0,0.5]);
    GearT1I D1 (k = 10, d = 10);
  equation
    D1.qref = 10;
    connect(D1.nDrive,R3.pDrive);
    connect(a,R3.a);
    connect(R3.b,Tailb.a);
  end Tail_Rotor;
```

model Top_Rotor

The Top_Rotor model is a specialization of the Rotor model. Two engines, $D2$ and $D3$ are used in this model. $D2$ is connected to the joint $R1$ that makes it possible to tilt the rotor and $D3$ is connected to the joint $R2$ so that the rotor can rotate. Top is the body of the rotor.

```
  model Top_Rotor extends Rotor;
    Real betax, betay;
    RevoluteS R1 (n = [0,0,1],q0 = 0.01);
    RevoluteS R2 (n = [1,0,0],q0 = 0.01);
    BodyV Top (mass = 1,
              I11 = 0.005, I22 = 0.005,
              nx = [0,1,0],
              Shape = "cylinder",
              Size = [0.02,1.1,1.1],
              Material = [1,0,0,0.5]);
    GearT1I D2 (k = 1000,d = 100);
    GearT1I D3 (k = 1000,d = 100);
  equation
    D2.qref = betay;
    D3.qref = betax;
```

The angles of the top rotor is set to betax and betay which are received from the control system.

```
    connect(D2.nDrive,R1.pDrive);
    connect(D3.nDrive,R2.pDrive);
    connect(a,R1.a);
    connect(R1.b,R2.a);
    connect(R2.b,Top.a);
  end Top_Rotor;
```

Another important class made is the ground model that makes it possible to land the helicopter. Basically it is a damper and linear spring connected in parallel.

model Ground

```
model Ground
  extends MbsTwoCut;
  parameter Real n[3] = {0,1,0};
```

The parameter n declares the normal vector to the ground plane.

```
  parameter Real c = 100000 "Spring constant [N/m]";
  parameter Real s0 = 0 "Ground level";
  parameter Real d = 1000 "Damping constant [N*s/m]";
  parameter Real n2[3] = {1,0,0} "x-dir";
  parameter Real n3[3] = {0,0,1} "z-dir";
```

The parameter $n2$ and $n3$ declares directions for the friction force.

```
  ExtForce force;
  Sensor sensor;
```

A sensor object determines the relative kinematics between two points. In this case we are interested in what height the helicopter is in and what speed it approaches the ground.

```
  Real s, sd, f1, f2, f3;
  Real my = 15 "friction coefficient";
equation
  connect(a,sensor.a);
  connect(b,sensor.b);
  connect(b,force.b);
  s = n' * sensor.rrela;
  sd = n' * sensor.vrela;
  f1 = if (s<s0) then c * (s-s0) + d * sd else 0;
  f2 = if (s<1) and abs(sensor.vrela[1])>0
       then my*sensor.vrela[1] else 0;
  f3 = if (s<1) and abs(sensor.vrela[3])>0
       then my*sensor.vrela[3] else 0;
  force.fb = -f * Sb' * n + (-f2 * Sb' * n2) +
             (-f3 * Sb' * n3);
end Ground;
```

$f1$ is the force inflicted when the helicopter lands and $f2$ and $f3$ are friction forces on the ground so that the helicopter does not slide when it lands with some speed. All these forces are put together in a force vector element. s and sd are the distance respective the speed in y-direction between two objects which will be connected to the ground object.

4.4 Solid Works generated helicopter model

Vadim Engelson[3] has designed a helicopter in the SolidWorks environment. It has then been exported to Modelica. Some of the code can be seen in appendix B. This helicopter consists of 10 parts, 4 revolute and 2 prismatic joints from the MBS library. The class BodyME is general class used for defining bodies of various shapes. Mass, inertia, center of mass and so forth is calculated by SolidWorks and this information can be found in these body objects.

The generated Modelica code can not be used directly in simulation runs. Every model must include an inertial object (see section 3.5 about MBS library). In our example, the forces have to be connected (rotorforce, tailforce, ground forces), as well as engines to appropriate joints. These Modelica objects cannot be automatically generated from SolidWorks.

Figure 4.3 shows a screen-shot of this helicopter when visualized.

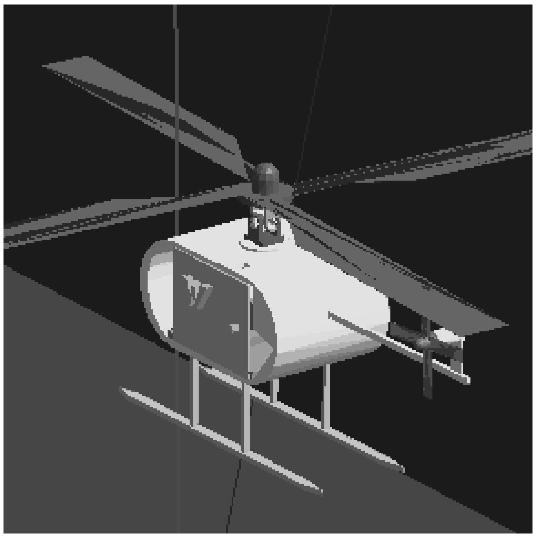


Figure 4.3: Screen-shot of the helicopter.

[3]Phd student and member of the Pelab group at Linköping University

Chapter 5

Helicopter control system

The current helicopter control system [10] enables the user of the helicopter to control it by using a command language called FCL (*Flight Command Language*). The control system is based on a helicopter model which has already been discussed in section 4.2. Basically it is a set of equations, i.e. a mathematical model that describes the behavior of a helicopter. Before we go any further explaining the details, we should determine how the origin frame (O) and the body-fixed frame (F) are situated. In Figure 5.1 you can see how the body-fixed frame is situated on the helicopter and how each Euler angle represents rotation around each axis. Euler angles Ψ, Φ and Θ are the so called yaw, roll and pitch angles.

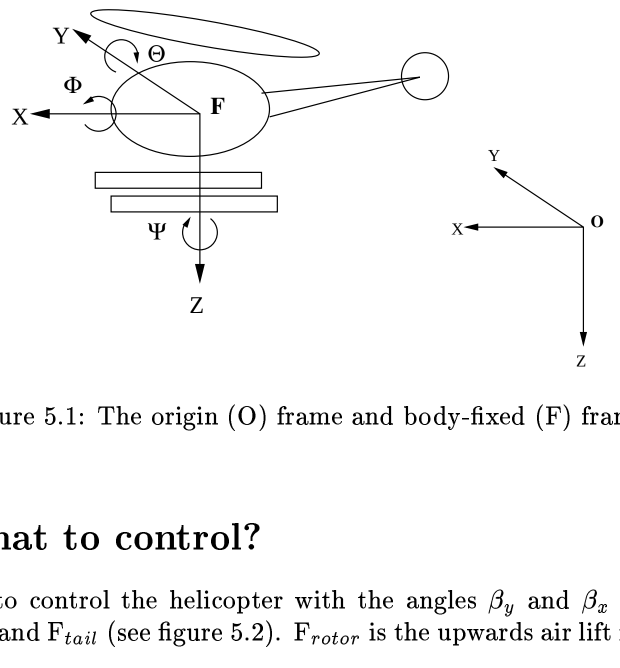


Figure 5.1: The origin (O) frame and body-fixed (F) frame.

5.1 What to control?

The idea is to control the helicopter with the angles β_y and β_x and the two forces F_{rotor} and F_{tail} (see figure 5.2). F_{rotor} is the upwards air lift force applied

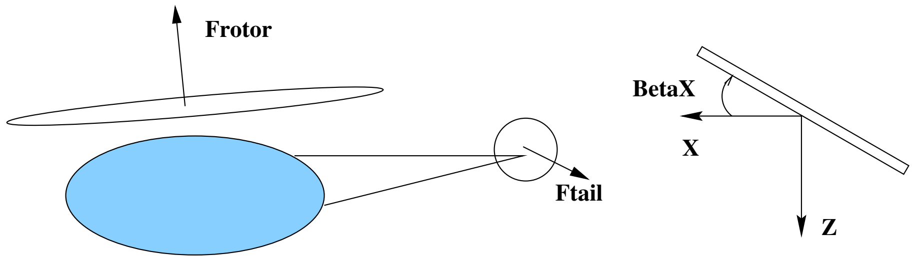


Figure 5.2: Positions of the forces Frotor and Ftail on the helicopter. The figure also shows how the BetaX angle is defined.

approximately over the center of gravity of the helicopter. Usually its value stays around $-mg$ (pointed downwards in z-direction). F_{tail} is the force from the tail rotor. β_y and β_x are the angles of the seemingly tilted top rotor plane. Why the top rotor is seemingly tilted is actually the result of a synchronized motion of up and down of the individual rotor blades. If these angles are not zero, the helicopter will rotate faster and faster. Therefore these angles are mostly zero and if non zero, then during short intervals. The force that makes the helicopter go forward and sideways comes from the gravity. The helicopter "falls forward" . This can be regulated by controlling the angles Φ and Θ. The third angle, Ψ is easily controlled by F_{tail}. The key idea of the control system is then to control the attitude angles Φ and Θ.

5.2 Control structure

As mentioned above, the helicopter is controlled by sending FCL commands. Every time a new FCL command is sent to the system an event occurs. An event can also be generated internally in the system. These events then controls the switching between the different modes in each channel. Which modes are available for each channel are described in table 5.1. Each mode corresponds in a certain way to a control law (Control laws are discussed below in section control laws). How these modes are combined in each channel, is determined by rules. In other words what actions these events cause are determined by rules. The actions can be:

1. Selecting control modes from the table above;
2. planning;
3. making observations.

Planning has to do with the initialization of the control system, which includes reading the parameters from the FCL language commands and setting the various parameters that are mentioned in the control law equations.

Observations are made about the world and what it does. They are mainly used to gate the checking of events so that only relevant events are checked.

So the rules discussed earlier actually have one input (event) and seven outputs (four flight modes, one plan and two observations).

To get a description on which rules and plans are used for different FCL commands, see Skarman [10].

xmode channel	ymode channel	zmode channel	Ψmode channel
velocity	velocity	altitude	external command
slow	position	altitude plan	clean
brake	waypoint	land	
position	align		

Table 5.1: Different control modes in different channels.

5.3 Control laws

You might say that the control layer contains seven different control laws. The structure and dependence between the control laws can be seen in figure 5.3.

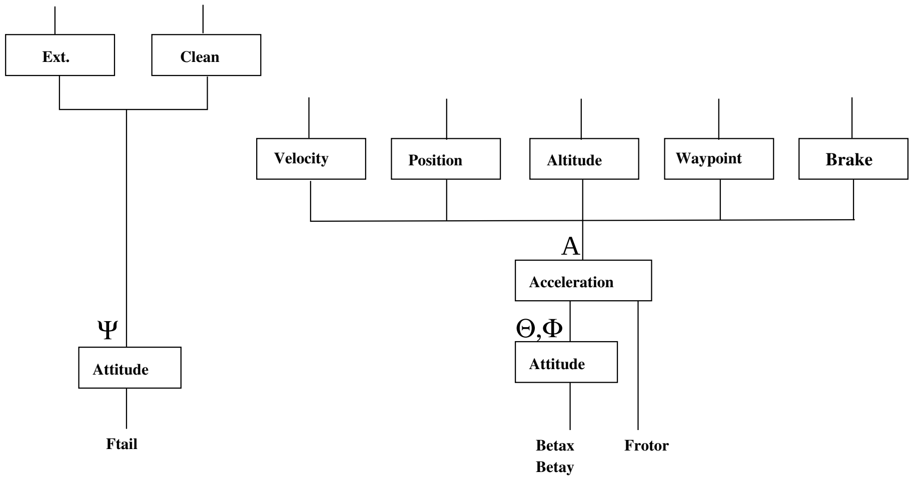


Figure 5.3: Structure of the control laws.

- *Attitude control* — This control contains a feedback loop of the rotation and the attitude angles of the helicopter. As a result you get β_y and β_x. F_{tail} is derived from a feedback loop containing the rotation and the yaw angle (Ψ) of the helicopter.
- *Acceleration control* — By using an inverse kinematics scheme, the attitude angles can be solved as functions of the desired accelerations in

x- and y-directions. F_{rotor} is evaluated by applying a feedback mechanism taking the difference between the desired acceleration in z-direction and the current acceleration of the helicopter in same direction.

- *Velocity control* — In the velocity control loop the difference between the required velocity and the current velocity is considered while calculating the desired acceleration.
- *Position control* — The feedback loop here contains current position and velocity of the helicopter.
- *Altitude control* — Altitude control is a special case of the position control. The same feedback pattern and design is used.
- *Brake control* — A non linear algorithm is applied here which can stop the helicopter with a constant acceleration and at a given position.
- *Waypoint navigation* — Waypoint navigation corresponds to take the helicopter to a certain remote point (waypoint) from its current position. A method called "line of sight" [10] is used as feedback.

Have in mind that some of the control loops are made in an inertial frame while some are made in body-fixed coordinate systems. For example, it doesn't make sense talking about position in a helicopter fixed frame. It is also important to point out that constants, integrations ??? have been left out and only the feedback mechanism has been considered. For more details see Skarman[10].

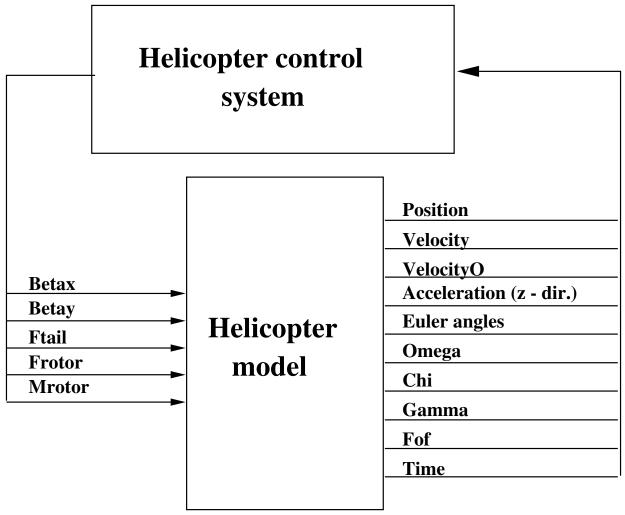


Figure 5.4: Input and output signals to the helicopter model.

5.4 Helicopter control system interaction

The current control system for the helicopter is coded in Ada and the Modelica models are compiled into C as mentioned before. In order to use Dymola's simulation platform DymoSim, an interface has been built to take care of the communication between the models and the control system. External functions calls are implemented in the models to handle the update of input and output signals. These functions are implemented in C. This C program "glues" the model and the control system together. It also includes functions for getting/setting values in the control system.

For the helicopter we have the input and output signals shown in figure 5.4.

Chapter 6

Robot control and model

6.1 Robot control system

The control system for the robot is very much simplified. It was decided to first design a control system that just performs one of the required functions: unloading the helicopter. The robot strictly follows the following procedures. It does not go to the next point until it is finished with the former.

1. The position of the cargo is sent to the robot.
2. The robot moves to appropriate position.
3. The robot turns its arm to appropriate angle.
4. The robot moves its arm to appropriate height.
5. The robot arm is moved forward.
6. The robot arm grabs the cargo.
7. The robot arm is moved backward.
8. The robot moves to new position.
9. The robot drops the cargo.

Depending on where the helicopter lands, the robot has to move in that direction. Figure 6.1 shows which situations are handled at this stage. Also the door has to be pointed towards origin (of coordinate system) so the robot does not have to move around the helicopter. This have to be taken into account when landing the helicopter.

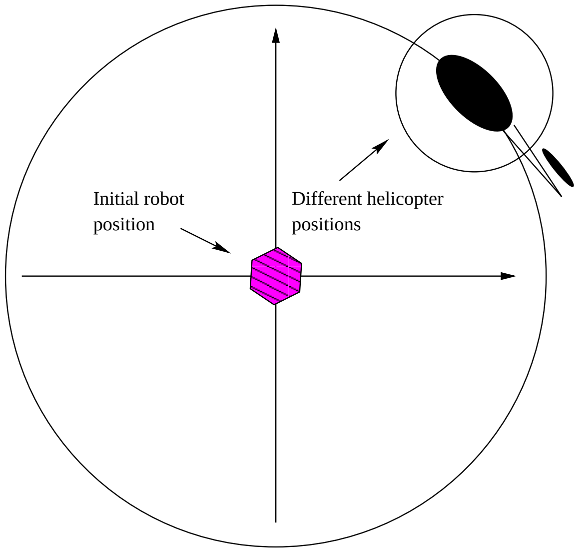


Figure 6.1: Situations handled during maintenance.

6.2 Solid Works generated robot model

The robot has been designed by Vadim Engelson in SolidWorks and then automatically converted into Modelica code. It is built by five body parts, four prismatic and one revolute joints from the MBS library. As for the SolidWorks generated helicopter model, this model needs to be connected to an inertial object before it can be compiled and simulated. Also, the joints have to be connected to appropriate engines so that you can control it movements. Some force elements have to be connected on the robot hand so it can grab/pickup cargo.

6.3 Robot control system interaction

The interface between the robot and the control system is built in the same way as for the helicopter and its control system. The model is compiled into C and the communication with the control system (which is written in C) is handled by external function calls.

Chapter 7

Maintenance operating environment

In the WITAS project there is need for maintenance environment system that provides the UAV with different type of services like refueling, unloading, etc. These services should be performed by an autonomous ground robot in the form of an industrial robot. In this thesis work a maintenance environment has been implemented in Modelica. The structure and classes that have been used can be seen in figure 7.1. Each class will be presented below. For the helicopter and the robot there are control systems (see chapter 5 and section 6.1) which communicate with these modules through external function calls. They work independently.

class World

The *World* class is the "main" class. In this class instances from the classes *Robot*, *Cargo*, *Ground*, *Helicopter*, *Pickup* and *Inertial* are created. As was pointed out earlier, there always has to be one Inertial instance in the model.

- One helicopter, one robot and one cargo object are instantiated and connected to inertial object.
- Eight Pickup objects are created. Four of them are connected between the cargo and the robot. These are needed so that the robot can pickup the cargo. Four of the same type is connected between the helicopter and the cargo. These are needed so that the cargo stays in place in the helicopter.
- Twelve Ground objects are created. Four of them are connected between the helicopter (under the landing gear) and the inertial so that the helicopter can land (i.e. the forces in the ground objects gets activated when the helicopter reaches the ground. This first makes the helicopter stop and then stay on the ground). Eight of them is connected between the inertial

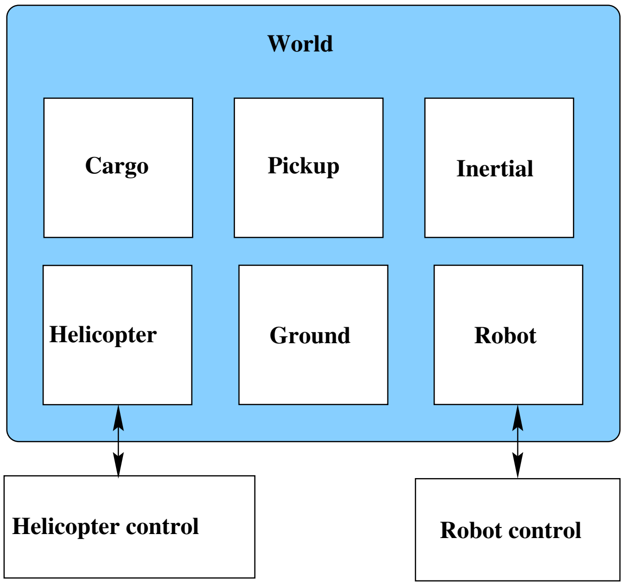


Figure 7.1: The World class and what classes are instantiated here.

and the cargo (on different positions on the cargo). So if the cargo, in this case a box, is dropped it will bounce on the ground more realistically.

- Two sensor objects are connected between the robot and the inertial so that the robot and the robot hand can be located in space. This information is sent to the robot control system.

class Helicopter

The SolidWorks generated helicopter is in cooperated in this class. Apart from what is said in section 4.4, some additional "glue coding" has been done. A freecardan object is connected to the helicopter which gives it six degrees of freedom.

class Robot

The SolidWorks generated robot is in cooperated in this class.

class Pickup

This class contains a force that is activated depending on a condition (i.e. flag). If the condition is true the force is activated, otherwise not.

class Cargo

This class includes a SolidWorks generated box. A freecardan object is connected to give the cargo six degrees of freedom. There is external function call that sends information about the cargo's position to the robot control system.

Chapter 8

Tools

This chapter gives a brief description of different tools used in the project. First the Dymola tool — *Dynamic modeling Laboratory* will be described. The Modelica language which is used for modeling in this work, is supported by the Dymola environment. SolidWorks have been used to describe some of the models and then automatically generate it to Modelica code. MVIS has been helpful when visualizing the animations to see for example if the helicopter model behaves like an real helicopter.

8.1 Dymola

Dymola is a commercial product from Dynasim AB [11]. It is a development environment for modeling and simulating complex, multi domain, dynamic systems. It consists of the following parts:

- The Dymola language translator;
- the Modelica language translator;
- DymoDraw editor;
- DymoViewer;
- the Dymola translator;
- Dymosim.

These parts will be described in the following sections. Some parts will be described more thoroughly than others depending on their use for the work in this thesis.

8.1.1 The Dymola language

Just like Modelica the Dymola language is an object oriented, equation based, non causal, multi domain modeling language. Reuse of modeling knowledge is supported by using class libraries and by using inheritance. Model details are given in differential equations and algebraic equations. The system then compiles the equations into assignment statements. Hybrid modeling as well as symbolic model manipulation is supported. Dymola is general purpose language which makes is suitable for different physical domains such as electrical, mechanical, thermo dynamical, etc.

8.1.2 The Modelica language

As have been pointed out earlier, Modelica is included in the Dymola environment. It is a successor to Dymola. The implementation of Modelica used in this thesis (version 0.91) transforms the Modelica models into Dymola models and then compiles it into the final simulation application. A later implementation directly transforms the Modelica models to executable C code.

Modelica has a more flexible object model than Dymola. Its array construct is much more powerful than Dymola's. It includes functions and algorithms, which are missing in Dymola. In addition, the Modelica class construct also includes functionality as C++ templates.

For a Modelica language description see chapter 3.

8.1.3 DymoDraw editor

This object diagram editor allows creation and editing of model classes. Structural properties, like connections between submodules are edited graphically. Equations, parameters, definitions and so forth are edited with a built in text editor. Figure 8.1 shows a screen shot of the top rotor diagram DymoDraw editor.

8.1.4 DymoViewer

DymoViewer is used to present the simulation results. This can either be done in a graph by plotting variables or by 3D visualization of the dynamic behavior of the model. In the present implementation only simple graphical objects such as spheres, cylinders and cones are supported. These have to predefined while defining the class model.

8.1.5 The Modelica translator

The Dymola program translates the model code into equations for DymoSim to solve. First these equations are preprocessed in ways such as common subexpression elimination, differentiation, solving algebraic loops, etc. In order to put

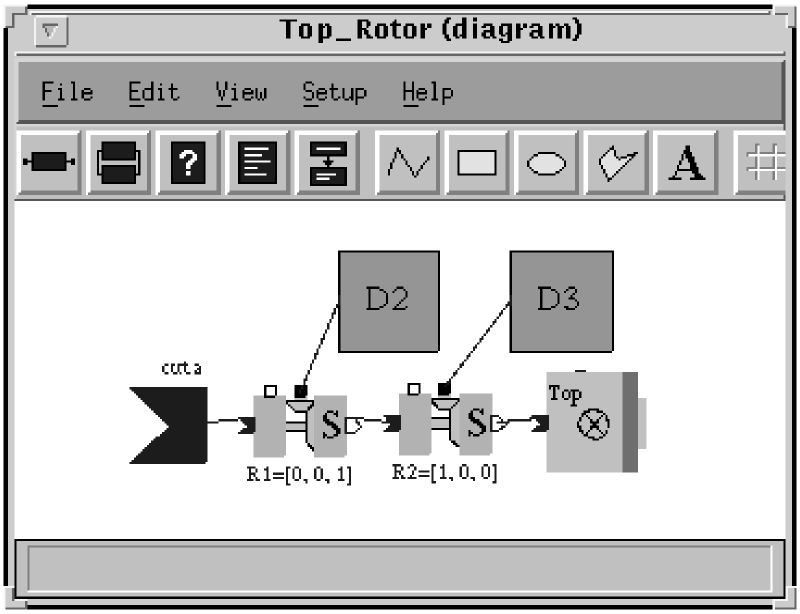


Figure 8.1: DymoDraw editor.

the equations in explicit form (ODE and DAE) symbolic manipulation, tearing and numerical solutions are used.

8.1.6 DymoSim

The Dynamic Model Simulator is a powerful simulation program for solving *ordinary differential equations* (ODE) and *differential algebraic equations* (DAE). It uses different types of numerical integration methods depending on the problem. Discrete events can also be handled providing that information about possible events is transferred to the DAE/ODE solver. The structure of DymoSim is shown in figure 8.2. The model to be simulated is defined as a set of C-functions according to the standard DSblock specification [12]. DSblock specifies when different routines are called, what the parameter lists are and what should be calculated. DSblock routines are generated by Dymola which then compiles and links these together with the Dymola Library to create a DymoSim executable.

8.2 Solid Works

SolidWorks is a mechanical design automation software that makes it easy to produce models and drawings. Like other 3D CAD systems SolidWorks uses the concept of parts and assemblies. Each solid component is modeled as a separate part document while in the assembly document the complete model's

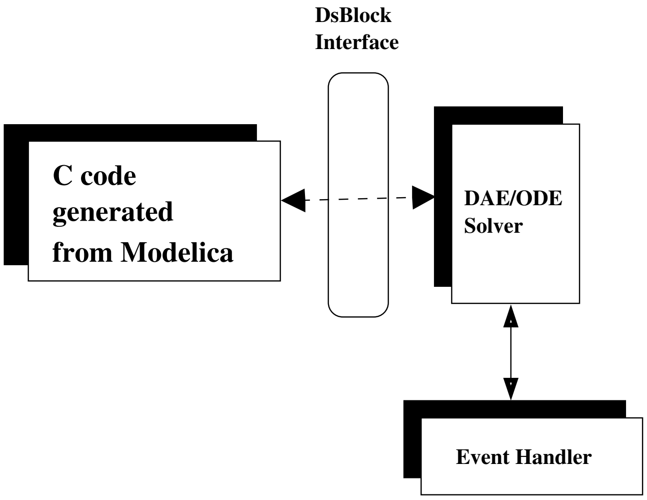


Figure 8.2: Diagram over DymoSim.

different parts are put together. In the assembly document the mobility between different parts is also defined. This mobility can be constrained by connecting one or more so called mates, each adding an extra constraint to the mobility.

SolidWorks has been used by Vadim Engelson to describe some of the models. The assembly which consists of a set of parts and stores a set of mates are validated and translated to a set of Modelica MBS class instances and appropriate connect statements between them.

8.3 MVIS

MVIS is a visualizer developed by Vadim Engelson. The Visualizer uses the STL format [13] which is a very simple format suitable for visualization. All the surfaces are divided into triangles. Every triangle's vertices and normal vector are listed in the STL file. The visualizer loads corresponding STL file for each part and optimizes it for rendering. Rendering is performed by OpenGL [13] library functions.

The visualizer can control various options by either using pop up menus, keyboard shortcuts or a string command. Some options are for example to rotate the camera in 2 degrees of freedom, display of parts as wire frame or filled color, display of application specific landscape, display of origin and coordinate systems, etc.

Chapter 9

Conclusions and future work

9.1 Discussion

The models developed in this thesis, the flat helicopter model, the object oriented helicopter model, the robot model and the maintenance operating environment have worked to some extent. The flat model, only based on equations, worked well with the Ada control system due to the fact that the control system was developed on top of these equations. All parameters such as mass and moments of inertia could be declared to some suitable value.

The object oriented and the SolidWorks generated helicopter models are more dynamic in the sense that they have moving parts (e.g. rotors) and a landing gear that allows a "smooth" landing. But because of usage of the MBS library it might be a bit hard to estimate for example the center of gravity where the forces should act on according to the control system (in the flat model, all the mass is concentrated to a point which makes it easier). This might be one of the reasons why these models sometimes are a bit unstable in flight. For some maneuvers such as sharp turns the models back part starts to oscillate. Another reason can be that the control system is not general enough.

The idea of a maintenance environment system was initiated late during the work for this thesis. Consequently we built up a simple environment were a robot interacts with an helicopter by unloading it. The scenario works but some parameters have to be tuned so that the robot goes to the right position, grabs the package in the right way.

Some models have been slow while simulating but some efforts were made to make them faster. In the future these models should be simulated in real time which can be a bit tricky (but similar things have been done).

Modelica has been pleasant tool to work with for developing the models. In the beginning it took some time to get comfortable with how Modelica works. There were also some troubles because of the fact that the Modelica language is still under development. I should point out that I didn't use the graphical tools but instead wrote Modelica code directly. Still you could see the beauty of Modelica when representing different physical systems equations.

9.2 Future work

Below I have listed some things that might be interesting to consider for the future. Some work on these proposals have already been started.

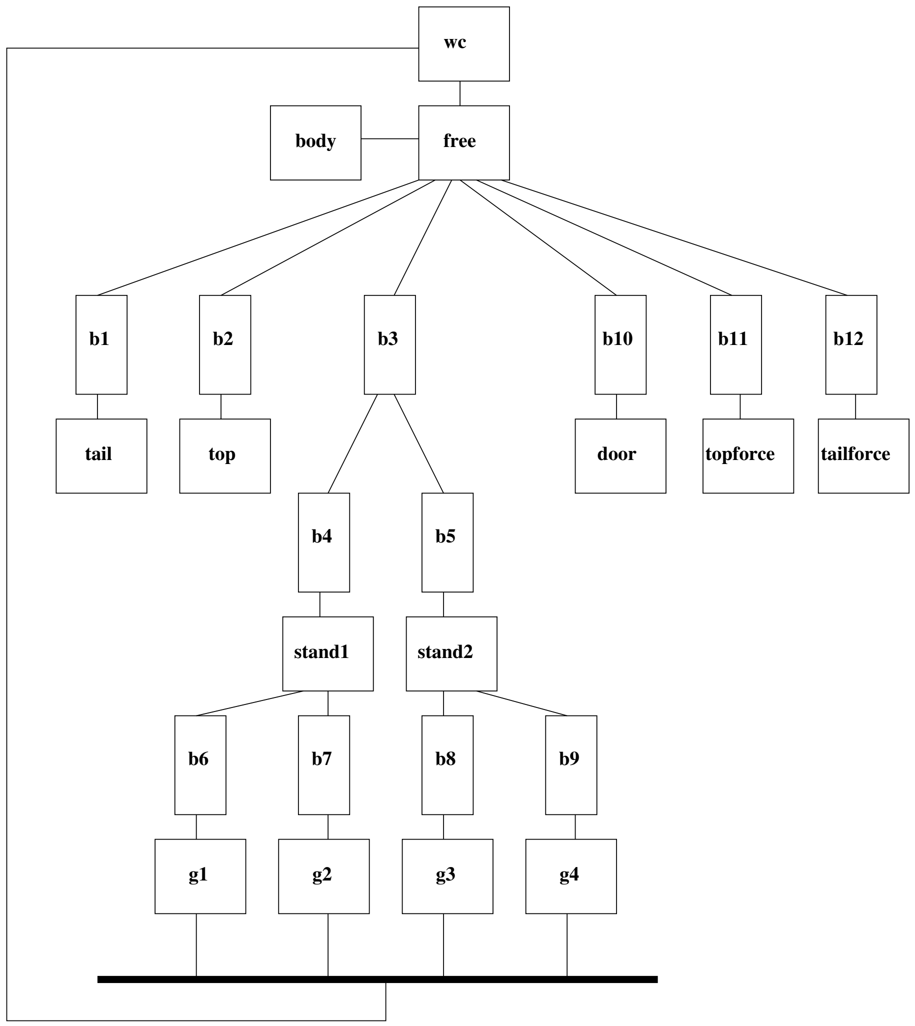
- *More realistic models* — The models in these thesis can of course be more developed into more realistic ones. The helicopter's rotor system can for example be modeled instead of like now, approximate it with a resulting force in the hub. But more realistic models often lead to more complex models which takes more time to simulate. So there is a trade-off between having realistic models and fast simulation runs.

- *Better control system for the robot* — The control system at the moment is only made for a specific task. Planning is needed for the robot so it can avoid obstacles in its way when it is approaching the helicopter. When the robot is reaching for the box, a technique called inverse kinematics can be used for calculating the trajectory of the robot arm. A similar command language like FCL should be developed for the robot model.

- *Faster models* — The aim is to run the helicopter, robot and the WMES models in real time. To achieve this, several approaches should be considered. The Modelica generated C code could be more optimized. The code could also be made parallel, so it can be run on different processors at the same time.

Appendix A

Object oriented helicopter connect diagram



Appendix B

SolidWorks generated helicopter model code

```
// ..\chop21.mo
import "BodyME.mo";
model chop21
        parameter Real nx[3]=[1;0;0];
        parameter Real ny[3]=[0;1;0];
        Real ac[24], th[6];
        MbsCutA a(across=ac, through=th);
        SubInertial I(nx=nx, ny=ny);
        BodyME nose_1(r=[0.0662952; 0.00707633; 0.00450532],
                I11=7.63407e-005, I22=8.90642e-005, I33=8.90642e-005,
                I12=0, I23=0, I13=0, mass=0.169646,
                r0=[-0.00450532; 0.00707633; 0.0362952],
                nx=[0; 0; -1],
                ny=[0; 1; 0],
                Material=[1; 0; 0; 1], parmStlIndex=16);
        Bar nose_1Bar0(r=[0.0962952;0.00707633;0.00450532]);
        BodyME hbod6_1(r=[0.482968; 0.0190422; 0.000638917],
                I11=1.42728, I22=4.12359, I33=4.09798,
                I12=-0.22473, I23=-0.00432753, I13=-0.0225688, mass=36.142,
                r0=[0.22; -0.18085; -0.25],
                nx=[1; 0; 0],
                ny=[0; 1; 0],
                Material=[0.87451; 1; 0; 1], parmStlIndex=11);
        Bar hbod6_1Bar1(r=[0.42;0.22915;0]);
```

...

```
        equation connect(a,I.a);
        equation connect(I.b,nose_1.a);
        equation connect(I.b,nose_1Bar0.a);
        equation connect(nose_1Bar0.b,hbod6_1.a);
```

...

```
        equation connect(nose_1Bar0.b,hbod6_1Bar12.a);
        equation connect(J5.pTrans,S0);
        equation connect(hbod6_1Bar12.b,J5.a);
        equation connect(J5.b,land1_1.a);
        equation connect(J5.b,land1_1Bar13.a);
        equation connect(land1_1Bar13.b,land1_1_F);
        equation connect(J5.b,land1_1Bar14.a);
        equation connect(land1_1Bar14.b,land1_1_B);
        equation connect(nose_1Bar0.b,hbod6_1Bar15.a);
        equation connect(J6.pTrans,S1);
        equation connect(hbod6_1Bar15.b,J6.a);
        equation connect(J6.b,land1_2.a);
        equation connect(J6.b,land1_2Bar16.a);
        equation connect(land1_2Bar16.b,land1_2_F);
        equation connect(J6.b,land1_2Bar17.a);
        equation connect(land1_2Bar17.b,land1_2_B);
        equation connect(nose_1Bar0.b,hbod6_1Bar18.a);
        equation connect(hbod6_1Bar18.b,hbod6_1_RotForceCutB);
        equation connect(nose_1Bar0.b,hbod6_1Bar19.a);
        equation connect(hbod6_1Bar19.b,hbod6_1_TailForceCut);
end chop21;
```

Bibliography

[1] Pelab homepage, `http://www.ida.liu.se/~pelab`

[2] WITAS Project homepage, `http://www.ida.liu.se/ext/witas`

[3] PATRICK DOHERTY (1998), *Project Status and Integration Report: The WITAS Integrated Software System Architecture*, IDA, Linköping University, Sweden

[4] Modelica homepage, `http://www.modelica.org`

[5] Documentation on kinematics loops and inverse models???

[6] HÅKAN LARSSON (1999), *Translation of 3D CAD models to Modelica*, IDA, Linköping University, Sweden

[7] PONTUS LIDMAN (1998), *Virtual Reality Environment for Simulation of Autonomous Aircraft*, IDA, Linköping University, Sweden

[8] ERIK SKARMAN (1998), *WITAS, An aircraft model*, IDA, Linköping University, Sweden

[9] ERIK SKARMAN (1998), *WITAS, A helicopter model*, IDA, Linköping University, Sweden

[10] ERIK SKARMAN (1999), *WITAS, A helicopter control system*, IDA, Linköping University, Sweden

[11] Dynasim AB homepage, `http://www.dynasim.se`

[12] DSblock specification???

[13] VADIM ENGELSON, PETER FRITZSON and HÅKAN LARSSON(1999), *Design, Simulation and Visualization Environment for Object-oriented Mechanical and Multi-domain Models*, IDA, Linköping University, Sweden

[14] ELMQVIST HILDING et al. (1998), *Modelica — A Unified Object-oriented Language for Physical Systems Modeling Tutorial and Rationale*

[15] MARTIN OTTER Modelica multi-body system library