EcosimPro and its EL Object-Oriented Modeling Language

Alberto Jorrín César de Prada Pedro Cobas

Department of Systems Engineering and Automatic Control, University of Valladolid, Spain,
{albejor,prada}@autom.uva.es.
Ecosimpro, Empresarios Agrupados, Spain, {Pedro Cobas} pce@ecosimpro.com

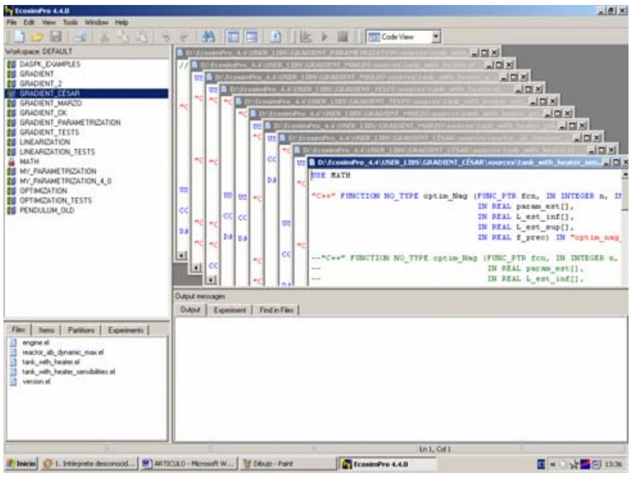
Abstract

This paper introduce the modeling and simulation tool EcosimPro and the possibility in the new version 4 of using object orientation in its modeling language called EL (Ecosimpro Language) which is the *official language of The European Space Agency* (ESA). Also shows the power that the use of classes gives to EcosimPro regarding other simulation environments.

1. Introduction

1.1 What is EcosimPro

EcosimPro is a simulation tool with a user-friendly environment developed by Empresarios Agrupados International for modeling simple and complex physical processes that can be expressed in terms of differential-algebraic equations or ordinary-differential equations and discrete events.



```
USE MATH

"C++" FUNCTION NO_TYPE optim_Nag (FUNC_PTR fcn, IN INTEGER n, IN
                                   IN REAL param_est[],
                                   IN REAL L_est_inf[],
                                   IN REAL L_est_sup[],
                                   IN REAL f_prec) IN "optim_nag_

--"C++" FUNCTION NO_TYPE optim_Nag (FUNC_PTR fcn, IN INTEGER n,
--                                   IN REAL param_est[],
--                                   IN REAL L_est_inf[],
```

Figure 1. General view of the tool.

EcosimPro runs on the various Windows platforms and uses its own graphic environment for model design.

1.2 What is EL

The language used in EcosimPro simulation tool for modeling dynamic systems based on equations and discrete events is named EL, which is the *official language of The European Space Agency* (ESA).

EL has been developed for use in modeling combined continuous-discrete physical systems. It allows mathematical modeling of complex components represented by differential-algebraic equations. One of the design goals for EL was to make it clear and easy to use for engineers doing simulations of this kind.

Models in EL are represented in a natural and intuitive way. EL has a simple but powerful language to prepare experiments on the models created, calculate steady states, transients, perform parametric studies, etc. It can also generate reports, plots and other hard copies from within a classic sequential language, and has the ability to reuse C and FORTRAN functions and C++ classes.

1.3 Key concepts in EcosimPro

The fundamental concepts of EcosimPro are:

- **Component**: This represents a model of the system simulated by means of variables, differential-algebraic equations, topology and event-based behaviour. The component is the equivalent of the "class" concept in object-oriented programming.
- **Port connection type**: This defines a set of variables to be interchanged in connections and the behaviour and restrictions when there are connections between more than two ports. For instance, an electric connection type uses voltage and current as variables to be used in connections. The connection port avoids connecting individual variables; instead, sets of variables are managed together.
- **Partition**: To simulate a component, the user first has to define its associated mathematical model; this is called a partition. A component may have more than one partition. For example, if a component has several different boundary conditions, depending on the set of variables selected, each set of variables produces a different mathematical model, or partition. The next step is to generate experiments for each

partition. The partition defines the causality of the final model.

- **Experiment**: The experiments performed for each partition of the component are the different simulation cases. They may be trivial for calculating a steady state or very complex with many steady and transient states changing multiple variables in the model.
- **Library of components**: All the components are classified by disciplines into libraries.

1.4 EL and Object Orientation

EL is a language for automatically solving systems of differential-algebraic equations, as this is how it works internally. However, the complexity of these systems is hidden: all the user has to do is define high-level equations using a high level object-oriented language similar to some programming languages and expressing the equations as algebra does.

EL is object-oriented: components can inherit from one another and can be aggregated to create other more complex, modular components. These features allow the modeller to reuse tried and tested components to create other more complex ones, incrementally.

1.5 The Component in EL

The most important element in EL is the component. A component represents a model by means of variables, topology, equations and an event-based behaviour.

A component may be simple, for example an electrical resistor or capacitor (with a couple of equations); or it may be very complex, for example the pressurized cabin of a space vehicle (with dozens of physics equations and many events).

All components have one block which represents the continuous equations and another which handles all the discrete events. Before the model can be run, it has to be generated. Until then, the components are only theoretical entities waiting to be used by other components or generated into a model.

1.6 The Experiment in EL

Once a model has been generated, experiments can be run on it.

EL has an experiment language similar to the modeling language which is used to integrate the model, calculate steady states, optimize parameters, etc

1.7 EL Basis

EL's design is based on continuous modeling concepts developed in the 1970s, new ideas of the '80s and '90s, and modern object-oriented techniques applied successfully in other fields.

1.8 EL Uses

EL has been designed to be used in industry directly, ranging from simple systems to very complex systems with hundreds of variables and equations. EL has mainly been successfully used for aerospace applications to simulate complex systems in Environmental Control and Life Support Systems (ECLSS) and in power generation to model complex power plants.

1.9 Mathematical capabilities

EcosimPro has Symbolic handling of equations (e.g.: derivation, equations reduction, etc.) and robust solvers for non-linear equations (Newton-Raphson) and DAE systems (DASSL and Runge-Kutta). EcosimPro controls the mechanism to interact with these solvers.Also uses dense and sparse matrix formats depending on the size of the Jacobian matrix. This allows problems with thousands of state variables to be simulated.

EcosimPro has Math wizards for:

- Defining design problems
- Defining boundary conditions
- Solving algebraic loops
- Reducing high-index DAE problems

and clever mathematical algorithms based on graph theory to minimize the number of unknown variables and equations.

Also incorporate Powerful discrete events handler to detect when events occur and powerful root finder mechanism based on Zbrent and Illinois methods. It is completely transparent to the user so the exact moment of the crossover of discrete events can be determined.

2. Object Oriented Modeling

Object-oriented Modeling is a powerful and intuitive paradigm for building models which will outlive the inevitable changes that are part of the growth and ageing of any dynamic system. It is not a panacea, but at least it provides the modeller with powerful features to hide complexity (encapsulation), to enable reuse (inheritance and aggregation) and to create models which are independent and easy to maintain.

Modular development allows a system to be modelled bottom-up. Basic library components can be combined to create increasingly complex components by combining two methods:

- Extension by inheritance from existing components
- Instantiation and aggregation of existing components

These ideas are applied to create a component which represents a complete system. The intermediate components can also be simulated. This greatly reduces development and maintenance time.

2.1 Encapsulation

One of the aims of object-oriented Modeling is encapsulation of complexity. For example, an engineer could model a complex problem with a purpose-built model consisting of hundreds of variables and equations. Although the model works, it will probably be difficult to follow and understand, and will certainly be difficult to reuse for other similar problems.

EL makes the modeller's life easier with its powerful abstraction capability and encapsulation of data and behaviour. Its main elements are libraries and components.

The libraries encapsulate all the elements involved in a particular discipline, and are exposed for use by other libraries. The component is a fundamental item for Modeling to express a dynamic behaviour associated with certain data.

With a conventional object-oriented language such as C++, the public interface is the data and methods declared public.

In EL a component's public interface consists of its ports, construction parameters and data. These elements are unique and are visible from outside during Modeling, reinforcing encapsulation and favouring reuse of the component.

Components are connected by their ports. The port definition includes the variables for a connection and their behaviour and restrictions when there are connections between more than two ports. By defining the behaviour of the ports, the system is able to automatically insert several connection equations, so the user does not need to be concerned about them. To add a new component, the user only has to be concerned about its internal behaviour and not about the connection equations.

2.2 Inheritance

Inheritance is what gives EL its tremendous power for sharing interfaces and behaviour.

In all fields of simulation, when many components are developed it becomes apparent that they share much of the same behaviour. EL can bring together common data and equations in parent components, to be inherited by their child components. This allows the creation of libraries based on parent components with a linear rather than geometric order of complexity. A new component based on another parent will include all its data and behaviours.

EL provides multiple inheritance; i.e., a component can inherit data and behaviour from one or many components which have previously been designed and tested. This capability allows the creation of new components reusing parts of others which have already been created.

2.3 Aggregation

In EL, all Modeling items are components. As described above, a component can inherit from another (or others), and can contain multiple instances or copies of others internally. This concept enforces the capability of reuse because it allows compound components to be created based on others which have already been developed.

This paradigm is applied iteratively and has no limit. The complexity of a final component can be hidden by the aggregation of tried and tested internal instances of other components or classes.

2.4 Data Abstraction

EL provides typical numerical data like REAL and INTEGER as well as other convenient data types like BOOLEAN and STRING. It also offers enumerative types to define the user's own data (very useful in chemical Modeling).

In addition, EL has multidimensional arrays, 1D, 2D and 3D tables and pointers to functions.

3. Classes

3.1 Introduction

Classes in EL are the equivalent to classes in classic object-oriented programming languages such as C++ and Java, but their use is more restricted (and simple). In fact, they are like high-level wrappers for producing a C++ class but bearing in mind that *the final users are engineers and not programmers*.

There are times when the modeller wants to encapsulate data and behaviour in the same item, and later instantiate them and use them by means of certain methods.The main difference between a class and a component in EL is that a component is meant to include dynamic equations and discrete events that the simulation tool arranges and solves, whereas a class represents a set behaviour and only allows the publication of variables and methods.

Classes are normally used in EL to support the Modeling of complex systems where the use of functions is sometimes improved if all the functions referring to the same utility are grouped together and share memory by means of common variables.

The general syntax to define classes is as follows:

```
class_def : CLASS IDENTIFIER ( IS_A
scoped_id_s)?
DESCRIPTION?
( DECLS var_object_decl_s )?
( OBJECTS class_instance_stm_s )?
( METHODS method_def_s )?
END CLASS
```

As all elements are optional, a very simple valid class would be:

```
CLASS math
END CLASS
```

An example of a slightly more complete class would be:

```
CLASS point2D “class reprenting a 2D
point”
DECLS
PRIVATE REAL x
PRIVATE REAL y
METHODS
METHOD NO_TYPE set2D(IN REAL valueX,
IN REAL valueY)
BODY
x = valueX
y = valueY
END METHOD
METHOD NO_TYPE get2D(OUT REAL valueX,
OUT REAL valueY)
BODY
valueX = x
valueY = y
END METHOD
END CLASS
```

This class represents the coordinates of a 2D point. It has a description, two private variables named “x” and “y” and two methods named “set2D()” and “get3D()”.

3.2 Inheritance

As seen above, a class can be inherited from one class (simple inheritance) or more (multiple inheritance) at the same time, in which case it will inherit all the associated variables and methods exactly as if they had been defined in the class itself.

To give an example of simple inheritance we can define a new class point3D as:

```
CLASS point3D IS_A point2D "a 3D
point"
DECLS
PRIVATE REAL z
METHODS
METHOD NO_TYPE set3D(IN REAL valueX,
IN REAL valueY,IN REAL valueZ)
“Method to init a 3D point”
BODY
set2D(valueX,valueY)
z = valueZ
END METHOD
METHOD NO_TYPE get3D(OUT REAL valueX,
OUT REAL valueY,OUT REAL valueZ)
“Method to obtain a 3D point”
BODY
get2D(valueX,valueY)
valueZ = z
END METHOD
END CLASS
```

This example shows how inheritance is used to inherit "point3D" class from "point2D" class and thus inherit their variables “x” and “y” and the methods “set2D()” and “get2D()”.To see an example of multiple inheritances, another class can be created.

```
CLASS statusClass “a status class”
DECLS
PRIVATE BOOLEAN status
METHODS
METHOD NO_TYPE setStatus(IN BOOLEAN
sta)
BODY
status= sta
END METHOD
METHOD BOOLEAN getStatus()
BODY
RETURN status
END METHOD
END CLASS
```

Then it can be inherited in the definition of point3D.

```
CLASS point3D IS_A point2D,
statusClass “a 3D point”
....
END CLASS
```

Here, the child class will inherit all variables and methods from parent classes; in other words, every point3D object will have access to all the public parts of classes “point2D”,“statusClass” and “point3D”.

Variables or methods cannot be inherited with the same name as this would produce an error in the compiler.

3.3 DECLS Block

With the DECLS block of a class, any kind of basic EL variable can be defined, whether this be a simple variable or a multidimensional array. For example, the following declarations are valid:

```
CLASS testClass
DECLS
REAL x = 9.9
REAL z,y = 1.1
INTEGER v[3] = {1, 2, 4}
BOOLEAN stat
STRING str = "hello world"
END CLASS
```

Defined in it are variables such as REAL, INTEGER, BOOLEAN and STRING. Initial values are also assigned. The general syntax for defining variables in classes is as follows:

```
var : PRIVATE? CONST? data_type name_s
( '=' init_expression )? STRING_VALUE?
```

Examples of declarations are:

```
PRIVATE REAL x = 1

PRIVATE CONST REAL x = 1

REAL speed= 1 “speed of the aircraft
(m/s)”
```

3.4 OBJECTS Block

This section allows us to declare objects that are instances of classes. The objects are written in the OBJECTS block of classes, components, functions and experiments. The general syntax of the objects declaration is as follows:

```
PRIVATE? names STRING_VALUE?
```

Valid declarations are:

```
OBJECTS
point3D p1
PRIVATE statusClass object1, object2
Point2D points[3,4]
```

3.5 METHODS Block

Methods define the functional interface of a class. They are subroutines connected to a definition of a class. They are always declared within a class in the METHODS block and can only be invoked from instances of that class.

Like functions, a method can return a basic EL type and has a number of call arguments which are defined when the method is written.

The general syntax of a method is:

```
method_def : PRIVATE? METHOD data_type
IDENTIFIER
'(' EOL* func_arg_decl_s ')'
STRING_VALUE?
( DECLS var_decl_s )?
( OBJECTS class_instace_stm_s )?
( BODY seq_stm_s )?
( END METHOD )?
```

An example of its use is:

```
CLASS example
DECLS
PRIVATE REAL x
METHODS
-- method to increment x with value v
(returns nothing)
PRIVATE METHOD NO_TYPE incr(IN REAL v)
BODY
x= x + v -- increase the class
variable x
END METHOD
-- method to return the value of x
after increasing it with value v
METHOD REAL popValue(IN REAL v)
BODY
incr ( v )
RETURN x
END METHOD
END CLASS
```

3.6 Using Classes

Classes defined in EL can be used in functions, components, experiments and in other classes.

Instances in classes are always defined in the OBJECTS block of each statement, as described above.These objects are used the same way as in other object-oriented programming languages. All their variables of instance and public methods (eg, those which are not tagged PRIVATE in their definition) are directly accessible by using the point (.) operator. The general rule is:

```
OBJECTREF.METHOD(....)
OBJECTREF.VARIABLE
```

For example:

```
object1.setValue(5)
object1.speed= 4
object1.foo.setValue(4)
```

The following class uses objects of “example” class defined in the previous section:

```
CLASS useExample
METHODS
```

```
METHOD NO_TYPE use()
DECLS
REAL v
OBJECTS
example ex -- declare object named ex
BODY
ex.popValue(2)
RETURN v
END METHOD
END CLASS
```

3.7 Class Associated With a Partition

When generating a partition, the tool can automatically generate an internal class representing the mathematical model generated. This provides a number of advantages:

- Any partition can be encapsulated in a single class
- This class provides an interface for interacting with the partition. For instance, initialization of variables, steady and transient calculations, get values of variables,etc
- Simulations can be embedded in components, functions, experiments and classes, since they are programmed with the class interface
- Multiple experiments can be executed in the same run
- Child classes (inherited from the partition classes) can be created by adding new variables and methods. In fact, a child class could provide complex experiments embedded in a single method.

To automatically generate the class associated with the partition, the user must select the option:

"Generate an associated class for a partition"

located in GUI option Edit->Options->General. In this case, each time the modeller makes a partition, an internal class will be generated with the name:

"ComponentName_PartitionName"

Use the underscore (_) separator to create a joint name from the component and partition names, such as aircraft_transient.

Once the internal class has been created, the modeller can declare an object of that class from any OBJECTS block, such as:

```
OBJECTS
aircraft_transient air
```

3.7.1 Access to Variables During Simulation

The user can access any model variable of the partition in these special classes. Since the variables can be of different types, there are different methods that allow the user to access the type of variable, to see if the variable exists, etc. Here is an example of available methods:

```
INTEGER getNumberVars ()
REAL getValueReal (IN STRING name)
BOOLEAN setValueReal (IN STRING name,
IN REAL v)
...
```

3.7.2 Operations Allowed with Classes of Models

By using these types of classes, you can perform any calculation with the partitions as if you were writing an experiment. For example, you can perform steady state and transient calculations on the same model.

Let's look at an example of an experiment carried out on the component aircraftGear from the DEFAULT_LIB library. The experiment exp1 carries out the following transient study:

```
EXPERIMENT exp1 ON
aircraftGear.default
INIT
-- Dynamic variables
y3 = 0.
y3' = 0.
y2 = 0.
y2' = 0.
x = 0.
x' = 60.96
BODY
REPORT_TABLE("reportAll", " * ")
TIME = 0.
TSTOP = 10.
CINT = 0.05
INTEG()
END EXPERIMENT
```

This same experiment can be written using these special classes. Suppose we create a new component called useAircraft and that in its *initialization* (INIT block) we want it to perform a transient calculation of the partition aircraftGear_default just as we did in the experiment. The code would be:

```
COMPONENT useAircraft
OBJECTS
aircraftGear_default air
INIT
-- set initial values
air.setTraceProgramme(TRUE)
air.setValueReal("y3", 0)
air.setValueReal("y3'", 0)
air.setValueReal("y2", 0)
air.setValueReal("y2'", 0)
air.setValueReal("x", 0)
air.setValueReal("x'", 60.96)
-- integrates the model
air.REPORT_TABLE("rAir", " * ")
air.TIME = 0.
air.TSTOP = 10.
air.CINT = 0.05
air.INTEG()
END COMPONENT
```

When using this component, a transient study will be executed for the aircraftGear model at the beginning of the calculation. This way, we have managed to embed a calculation of a mathematical model into another component. This makes the language very powerful for embedding mathematical models inside others.

3.7.3 Activation of Flags

By default, performing calculations on any class associated with a partition will not produce any on-screen messages or log files.

There is a set of functions to activate and deactivate these flags which print simulation messages when running, return the actual status of flag for tracing the simulation, print simulation messages in the log file, check assertions when running, ...

3.7.4 Creation of Classes Based on Partition Classes

We can also create our own classes by inheriting them from the automatically generated class of partition. This way, a more user-friendly interface can be created for operating with a mathematical model.

For example, a new class can be created by inheriting it from the class aircraftGrear_default and writing the experiment written in the INIT block of the example in the previous section in a method of the class.

```
CLASS aircraftTransient IS_A
aircraftGear_default
METHODS
METHOD NO_TYPE run()
BODY
-- set initial values
setTraceProgramme(TRUE)
setValueReal("y3", 0)
setValueReal("y3'", 0)
setValueReal("y2", 0)
setValueReal("y2'", 0)
setValueReal("x", 0)
setValueReal("x'", 60.96)
-- integrates the model
REPORT_TABLE("rAir", " * ")
TIME = 0.
TSTOP = 10.
CINT = 0.05
INTEG()
END METHOD

END CLASS
```

In this example, it has been embedded the same experiment in a method of a new class called “aircraftTransient”. Thus, we can now re-write the class useAircraft as:

```
COMPONENT useAircraft
OBJECTS
aircraftTransient air
INIT
-- run the experiment
air.run()
END COMPONENT
```

This, as can be seen, is a great simplification, since it only calls the method “run()” which encapsulates the whole experiment.These classes have all the properties of a normal class and can define new variables and methods, be inherited by others, etc. This gives great flexibility for encapsulating experiments in methods of classes and reusing them later.

3.7.5 An ilustrative example: Initialization of models

The main objective of this problems is to allow to start a simulation from stationary conditions.To formulate this kind of problems in EL, classes are used.

For instance:

In a problem of an eletrical engine:

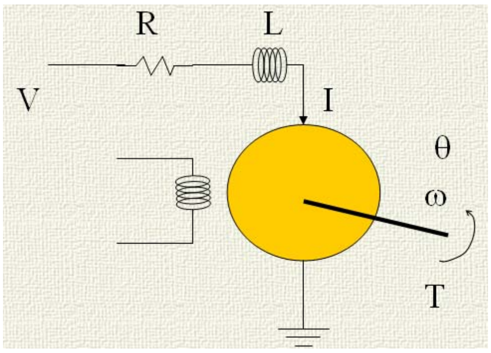


Figure 2. Electrical engine schematic.

whose model is given by:

$$J\frac{d\omega}{dt} = k_1 i - f\omega - T \qquad \text{Newton Law}$$

$$V = Ri + L\frac{di}{dt} + k_2\omega \qquad \text{Ohm Law}$$

ω angular velocity

J moment of inertia

V source voltage

I armatura current

T external torque

$k_2\omega$ f.c.e.m

Figure 3. Electrical engine dynamic equations.

The codification in EL is for instance:

```
COMPONENT engine

   DATA

      REAL R = 0.2       -- electric resistance (ohmios)
      REAL L = 0.01      -- electric inductance (H)
      REAL k1 = 0.006    -- armature constant (lbs-pie/A)
      REAL f             -- damping ratio of the mechanical system(lbs-pie/rad/seg)
      REAL J = 0.001     -- moment of inertia of the rotor (slug-pie2)
      REAL k2 = 0.055    -- speed constant (V.seg/rad)

   DECLS

      REAL V             -- source voltage (V)
      REAL omega         -- angular velocity
      REAL i             -- armatura current
      REAL T             -- motor torque applied to the shaft


   CONTINUOUS

      J * omega' = k1*i - f *omega - T
      L*i' = V - R*i - k2*omega


END COMPONENT
```

Figure 4. Electrical engine dynamic codification in EL.

The partition to work with that model for example could be:

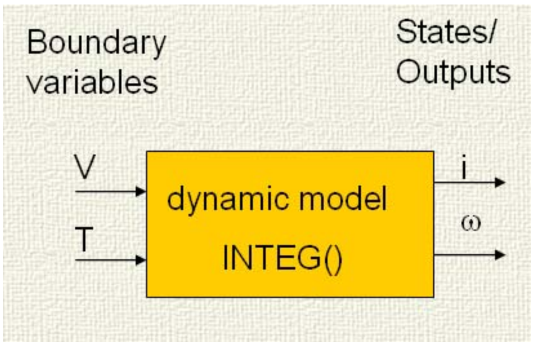


Figure 5. Partition of the system.

where the variables "V" and "T" are the boundary variables and "i" and "w" are the states and outputs of the process.

Intuitively, if the model were easy, to start with stationary conditions, only would be necessary write the equations in the INIT block by this way:

$$i(0) = \frac{V(0) - k_2\omega(0)}{R}$$

$$T(0) = k_1 i(0) - f\omega(0)$$

Figure 6. Initializations.

turning the derivatives into null and clearing the outputs up.

But this operation only could be done if the model equations are easy.

In general this problem could be solved using classes and creating a static partition with the component. In this case the static partition would be for example:

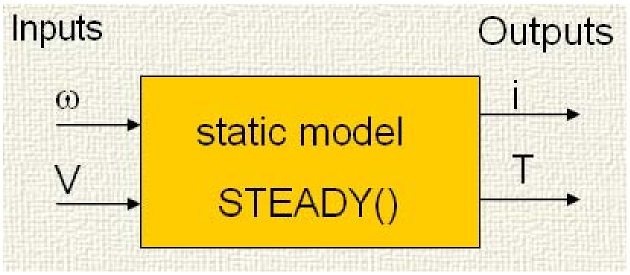


Figure 7. An example of a partition to initialize the dynamic model at stationary conditions.

where V and W are the boundary variables and inputs. With "w" and "v" values could be computed the i(0) value.

$$i(0) = \frac{V(0) - k_2\omega(0)}{R}$$

Figure 8. First calculation.

and with the i(0) value, could be computed the T(0) value.

$$T(0) = k_1 i(0) - f\omega(0)$$

Figure 9. Second calculation.

This partition could be used in the INIT zone of the dynamic component, so another component would have been created with the static equations.

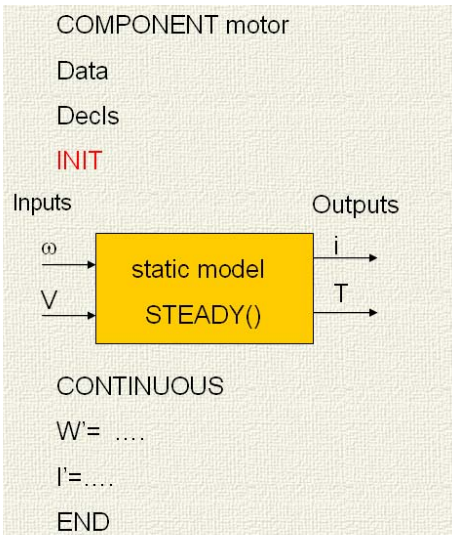
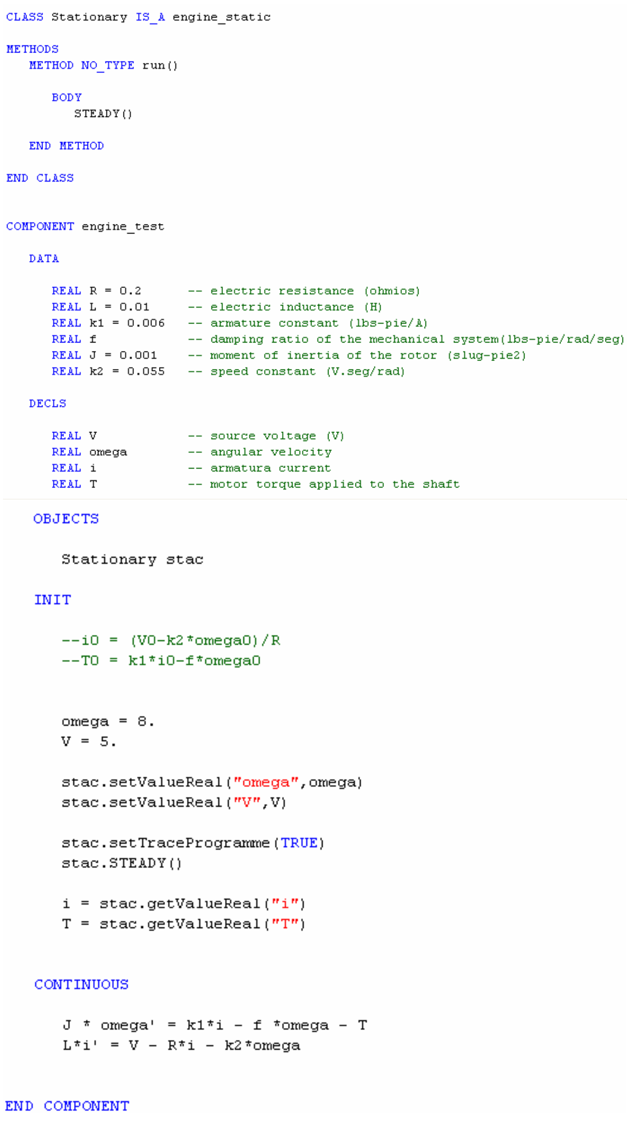


Figure 10. New component structure schematic.

So, the codification of the component using classes would be:



```
CLASS Stationary IS_A engine_static

METHODS
   METHOD NO_TYPE run()

      BODY
         STEADY()

   END METHOD

END CLASS


COMPONENT engine_test

   DATA

      REAL R = 0.2      -- electric resistance (ohmios)
      REAL L = 0.01     -- electric inductance (H)
      REAL k1 = 0.006   -- armature constant (lbs-pie/A)
      REAL f            -- damping ratio of the mechanical system(lbs-pie/rad/seg)
      REAL J = 0.001    -- moment of inertia of the rotor (slug-pie2)
      REAL k2 = 0.055   -- speed constant (V.seg/rad)

   DECLS

      REAL V            -- source voltage (V)
      REAL omega        -- angular velocity
      REAL i            -- armatura current
      REAL T            -- motor torque applied to the shaft

   OBJECTS

      Stationary stac

   INIT

      --i0 = (V0-k2*omega0)/R
      --T0 = k1*i0-f*omega0


      omega = 8.
      V = 5.

      stac.setValueReal("omega",omega)
      stac.setValueReal("V",V)

      stac.setTraceProgramme(TRUE)
      stac.STEADY()

      i = stac.getValueReal("i")
      T = stac.getValueReal("T")


   CONTINUOUS

      J * omega' = k1*i - f *omega - T
      L*i' = V - R*i - k2*omega


END COMPONENT
```

Figure 11. New Electrical engine dynamic codification in EL using classes to initialize .

Finally using that component and that class the simulation will start from stationary conditions.

4. Conclusions

The *object-oriented programming language* of *EcosimPro*, known as EL, is therefore one of the ***pioneer languages*** that has to deal with this new way of Modeling physical systems.

So finally the advantages of this type of methodology that offers the modeller over the others are going to be list:

- The modeller encapsulates data and behaviour into individual components (minimise global data)
- A component hides the complexity by making public only a certain part of the component
- The public interface comprises parameters, data and ports, while the local variables, discrete events and equations remain private
- The complexity grows in a linear rather than a geometrical way
- Reuse of tested components
- Inheritance simplifies the Modeling by sharing many common data and equations. Less code, more productivity and easier maintainability
- A component can contain equations that can be inserted at the time of simulation or not, depending on certain parameters passed to the component
- Use of virtual equations. Some equations change from parents to children. Users can decide to overwrite a parent equation with their own
- Equations format is declarative. Algorithms will symbolically transform the equations so that they fit in the best possible way to be solved numerically
- In general, it be can quoted the following considerations in the revolution of object-oriented Modeling:
- Modeling is non-causal. In other words, when a component is modelled, the causality of equations is not given. This will be solved during the final moment of simulation. Libraries of generic and reusable components can therefore be created in all situations
- Tried and tested components are constantly reused through simple or multiple inheritance and aggregation
- There is extensive use of hidden information and encapsulated data to deal with the complexity, solving each Modeling problem in its associated component and not taking other global data into account
- It is easier to make changes in the models because everything is divided into parts

Acknowledgements

A few words of thanks to the Empresarios Agrupados staff for giving me kindly all the information that I need.

References

[1] EcoEC: EcosimPro External Conections Version 4.4. EA International. 2008

[2] EcoEML: EcosimPro EL Modeling Language Version 4.4. EA International. 2008

[3] EcoIGS: EcosimPro Installation and Getting Started Version 4.4. EA International. 2008

[4] EcoMASG: EcosimPro Mathematical Algorithms and Simulation Guide Version 4.4. EA International. 2008

[5] EcoUM: EcosimPro User Manual Version 4.4. EA International. 2008

[6] EcoVU: EcosimPro Version 4.4 Upgrade. EA International. 2008

[7] www.ecosimpro.com