COMPARISON BETWEEN MODELICA 2.0 Y ECOSIMPRO/EL 3.2

Luis José Yebra Muñoz
CIEMAT-PSA. Apdo. 22, E-04200, Tabernas, Almería, Spain, luis.yebra@psa.es

Ramón Perez Vara
Empresarios Agrupados, Magallanes 3, 28015 Madrid, Spain
e-mail: rpv@empre.es

Sebastián Dormido Bencomo
Dpto. Informática y Automática, Facultad de Ciencias
UNED, Fac. de Ciencias, Dpto. de Informática y Automática. Avda. Senda del Rey 9, 28040 Madrid, Spain
e-mail: sdormido@dia.uned.es

Manuel Berenguel Soria
Dpto. Lenguajes y Computación, Escuela Politécnica Superior
Universidad de Almería, Ctra. Sacramento s/n, 04120 Almería, Spain
e-mail: beren@ual.es

Abstract

Modelica and EcosimPro/EL are object-oriented tools which use equations for modelling continuous physical systems and discrete events. This paper describes the comparative study which was carried out at a general level to show the different characteristics of the two languages. In this work we have generally tried to pick out the most significant properties of the object-oriented languages and determine which of them are common to both. The paper goes on to describe the incorporation into the model of a simple control system in both languages and discusses the properties used in each implementation in a fair amount of detail. It ends with the conclusions, indicating which characteristics could be included or modified in both languages with a view to increasing the system modelling and simulation capabilities offered by each one.

Key Words: Object-oriented modelling, EcosimPro, EL, Dymola, Modelica.

1 INTRODUCTION

The objective of this work is to carry out a comparative study of two languages, Modelica and EcosimPro/EL (from now on referred to as EL), which are currently being used to model physical systems. The former was developed by “Modelica Association”, a non-profit making organisation with headquarters in Linköping, Sweden. It was created in response to the need to define a standard modelling and simulation language as a result of an agreement between a group of specialists in these disciplines.

EL on the other hand is a language developed by EA International specifically for use with EcosimPro, its tool for modelling and simulating physical systems.

Both languages are basically object-oriented and use declarations. They contain all the elements necessary to model and simulate continuous, discrete and hybrid dynamic systems in a non-causal way.

In the comparison we used Modelica 2.0, based on the last issue of the public document which defines it [1] and in the case of EL we used the documentation supplied with EcosimPro [2]. This analysis does not include details about characteristics which are not related to the modelling and simulation activities. An in-depth, detailed comparative study would generate a huge amount of documentation. This work summarises the system modelling and simulating concepts of both languages and the way each one implements them.

An angular velocity control system is subsequently modelled and simulated using the two languages. The system is particularly interesting and illustrative because it is simple and yet responds to different modelling needs which can be met in different ways in each of the languages. Extensions which will enable the languages to cover further needs are also proposed.

The work concludes with a section which summarises the fundamental differences between both languages and suggests some extensions that could be made to them.

2 MODELICA 2.0 vs ECOSIMPRO/EL 3.2

This section addresses the general comparison between the two languages. An in-depth comparison which included syntactic aspects would result in a document as large as it would be lacking in useful information. The work therefore focuses on aspects that differentiate the two languages on a conceptual level. The comparison will be made in the following order: data types and properties, sentences, terminals, component classes, functions, libraries, partitions and definition of experiments. Tables summarising the properties and characteristics of the two languages will be included in each section.

2.1 DATA TYPES AND PROPERTIES

2.1.1 Basic Data Types

Table 1 sets out the types of fundamental variables for both languages.

Table 1: Types of Fundamental Variables

Type	Modelica	EL
Boolean	Boolean	BOOLEAN
Whole	Integer	INTEGER
Real	Real	REAL
String	String	STRING
Table 1D	NO	TABLE_1D
Table 2D	NO	TABLE_2D
Table 3D	NO	TABLE_3D
Function pointers	NO	FUNC_PTR

EL contemplates two types of tables and function pointers as language base types. Modelica does not include them as such, but it does offer ways to include them as composite types. For example, data tables for interpolation in Modelica can be implemented by coding them or by using existing libraries which offer this function, such as the 'Modelica Additions' library.

EL's function points are not contemplated by Modelica, but there are sufficient means to replace its purpose. In general, whether with regard to programming or modelling languages, function pointers are used to express parametric behaviour which can be passed as an argument for classes or functions. This feature can be incorporated into Modelica with the use of parametric classes and/or components.

In Modelica all the base types are considered as preconstructed classes comprising a series of attributes which model their behaviour. All the base type attributes are specified in [1]. With these attributes it is possible to:

?? Assign a physical quantity to the variable that the base type refers to; eg, voltage, current, temperature, etc
?? Define the units which the variable represents and the units in which the variable is to be displayed; eg, the units represented can be expressed in m/s, whereas the variable can be displayed in km/h
?? Define validity intervals for the variable, beyond which the tool would generate a warning or error message during model simulation
?? Define attributes to condition the tool to assign the initial value and parameters of the variable

In the case of real types, there are attributes which can be used to define nominal values for the variable so that its value can be standardised or the tolerance vectors of the DAE resolvers can be adapted. Other attributes enable the real types, or their subtypes, to condition the symbol [symbolic???] handlers when selecting the status variables in the event of higher index problems.

These attributes are defined so that they can be used by the tools incorporating the Modelica language, although they might not necessarily use them. Dymola is the tool used by the authors with Modelica. The documentation [3] does not specify that all the functions that these attributes offer must be used as indicated in [1].
EL does not treat the fundamental types as classes, but rather as basic language types which will form part of other components.

2.1.2 Types of Enumerated and Derived Data

This section addresses the possibilities which both languages offer with regard to enumerations, subsets and arrays. They are outlined below in Table 2.

Table 2: Enumerated and Derived Data Types

Type	Modelica	EL
Enumerations	YES	YES
Subsets	NO	YES
Arrays	YES	YES

Let us take a look at some of the differences regarding the types discussed. In both cases the enumerations are completely supported. The subsets are only defined for EL.

The most important difference, which is perhaps a clear disadvantage, is the limited elements which can be contained in an array and this is demonstrated by EL. The use of basic and enumerated types as elements in an array is restricted by EL. Modelica, on the other hand, allows classes or models as well as the basic and enumerated types to be elements in any array. This restriction in EL could give rise to limitations in the modelling of systems formed by an array of regular interconnected structures. In general, distributed parameter models can be built with very modular arrays of components.

2.1.3. Data Properties. Variability

The variability of the different types of variables is evident in both languages. The only difference between the two languages lies in the degree of syntax. Both allow variables to be defined with different variability:

?? Constants. Once declared and initialised, their value never changes

?? Parameters. Once declared and initialised their value does not change during a simulation, although it might change from one simulation to another

?? Discrete variables. Their value changes during a simulation only at the instants in which discrete events occur. Their value is maintained in the time intervals between events

?? Continuous variables. Their value can change at any moment during a simulation

2.1 SENTENCES

In general, the sentences that comprise the language code for physical system modelling can be classified into sequential, continuous and discrete sentences.

Sequential sentences are executed in a predefined order within the code section to which they belong. They are the typical sentences used in programming languages like FORTRAN or C++.

Continuous sentences describe the behaviour of continuous dynamic systems. They are normally mathematical ratios which link some variables with others. They are generally sets of equations which are not necessarily continuous.
Discrete sentences describe discontinuities which are explicitly defined by means of relational or logical expressions. With these types of sentences the existence of discontinuities in continuous models can be declared, or systems of discrete events can be modelled directly. The are executed in each integration run calculated by the numerical resolver.

The concepts of continuous, discrete and hybrid systems are quite clearly described in [4] and [8].

Table 3 summarises the types of sentences offered by the two languages. The order is the same as that described in [2].

The sentences in both Modelica and EL are of similar construction and permit the functions described to be implemented.

2.2 TERMINALS

The terminals allow the components of a single system to communicate with each other. In Table 4 we can see the properties the two languages provide in the implementation of these types of components.

With these types of components, EL is more flexible than Modelica. With EL the reserved words ‘IN’ and ‘OUT’ can be used to define the addressability of the ‘Through’ variables of the terminals.

Table 3: Types of Sentences

Type	Sentences	Modelica	EL
Sequential	Assignment	YES	YES
	Function call	YES	YES
	If-then-else	YES	YES
	While	YES	YES
	For	YES	YES
	Assertions	YES	YES
	Return	Implicit	YES
Continuous	Equations	YES	YES
	Symbolic expansion of equations	YES	YES
	Symbolic expansion of groups of equations	YES	YES
	Equations defined by zones	YES	YES
Discrete	Detection of events by condition	YES	YES
	Discrete assertions	YES	YES
	Symbolic expansion of discrete sentences	YES	YES
	Symbolic expansion of blocks of discrete sentences	YES	YES

Table 4: Terminals

Characteristic	Modelica	EL
Parameters	YES	YES
Across variables	YES	YES
Through variables	YES	YES
Programmable addressability	NO	YES
Continuous sentences in the terminal	NO	YES
Connection restrictions due to causality	YES	YES
Additional connection restrictions	NO	YES

The ‘across’ and ‘through’ variables are clearly described in [4], although in summary it could be said that the ‘across’ variables are those which are potentially associated with the component interconnection nodes and the ‘through’ variables are the flows associated with the potential ones and they enter or leave the components via the terminals.

With EL, equations that relate the variables can be introduced and the 'SINGLE' modifier can be used to restrict the number of connections of a given port.
With Modelica, addressability is defined in the language by agreement, and in the Through variables the flow always enters in the component.

It does not allow equations to exist in the terminals. In fact, all models created with Modelica are classes, and the 'connectors' are restricted classes because they do not have code blocks which contain continuous or sequential (or, therefore, discrete) sentences. The only terminal connection limitations are those defined by their causality.

2.3 COMPONENT CLASSES

The classes or components are the fundamental elements of object-oriented languages. They constitute the basic tool for representing the behaviour of systems ranging from simple indivisible ones to very complex ones formed by other subsystems.

2.4.1 Blocks of Code in Classes

Both languages use a set of code sections to define the classes. Each section has a certain objective in both languages. The class sections for the two languages are listed in Table 5.

Table5: Code Sections in Modelica and in EL

Modelica	EL
declarative protected equations initial equations algorithm	PORTS DATA DECLS TOPOLOGY INIT DISCRETE CONTINUOUS

The 'declarative' section in Modelica is the first section which exists after the class name has been defined. It has no label and it is the location where the class variables and components are declared.

There is no unique association between the sections of the two languages. Table 6 clearly summarises what type of code is included in each section.

Table 6: Association between Sections

Declaration of:	Modelica	EL
Parameters	declarative ('parameter')	DATA
Terminals	declarative	PORTS
Protected variables	protected	DECLS
Components	declarative	TOPOLOGY
Connection between components	equation	TOPOLOGY
Initialisation of variables	initial equation or 'start' attributes of variables	INIT
Events	equation and algorithm	DISCRETE
Continuous sentences, equations	equation	CONTINUOUS
Sequential sentences	algorithm	INIT

Some clarifications have to be made in connection with Table 6. In EL the parameters can be declared both in the DATA section and in the component identification definition.

There are two ways to initialise the variables in Modelica: assigning the 'start' and 'fixed' attributes of the variables upon initialisation, or defining equations for the initial instant in the 'initial equation' section. In both cases the sentences are declarative which means the tool will perform the necessary algebraic equations to solve the problem of the initial value which arises in the simulation. EL, on the other hand, deals with initialisation in INIT, sequentially executing the initialisation instructions contained in the INIT block.

Sentences which define events can only be declared in the 'DISCRETE' block in EL. However, in Modelica they can be declared in the 'equation' or 'algorithm' block. In fact, when they appear in the 'algorithm' block the definition of events can be prioritised so that in the light of possible simultaneous events, they could always be handled in a certain order. Details of the different configurations are given in [5].

In Modelica, the use of sequential sentences is dealt with in a section specific to components which include this type of behaviour; that is, in the 'algorithm' section.

2.4.2 Advanced Features

Both EL and Modelica have certain advanced features with which the behaviour between classes can be parametrised or varied. Table 7 lists the features, indicating which of them are incorporated into each of the languages.

Table 6: [Advanced Features???]

Feature	Modelica	EL
Multiple inheritance	YES	YES
Abstract classes	YES	YES

Feature	**Modelica**	**EL**
Virtual equations	NO	YES
Parametric classes	YES	NO

Both Modelica and EL support multiple inheritance and abstract classes.

The virtual equations are a very flexible mechanism for modifying the behaviour of components which belong to the same inheritance. This property is not supported in Modelica, so when an equation is defined in one class it can never be modified in descendant classes.

Parametric classes constitute a mechanism which allows the class to which one or several components may belong to pass as an argument in definition or instantiation time. This is extremely powerful when it comes to reusing libraries which have been designed to exploit this feature. Comparing it with object oriented programming, the parametric classes are the same as the 'templates' in C++ [6]. EL does not have this feature.

2.5 FUNCTIONS

As far as functions are concerning, there are no conceptual differences between the two languages. Both allow the functions of external languages C and FORTRAN to be invoked. EL also allows C++ functions to be invoked.

2.6 LIBRARIES

The libraries are conceptually similar in both languages. Their purpose is to store models or classes which have been developed for subsequent use. The differences can be found in the implementation details.

One clear difference between EL and Modelica is the inclusion of graphic representation properties both in the components themselves and in the libraries where they are stored. Modelica allows an icon to be associated within the actual definition of a component by means of 'annotation' directives. This enables the user to program the graphic information of the component and to model its behaviour. For example, the complete source code of the 'feedback' element in Figure 1 declares not only the operation of difference between the two input variables, but also adds graphic information which enables the figure to be displayed. With Modelica, a library designer could even directly program the graphics primitives which represent the class in a graphic diagram. The graphic description of a component is normally created by the actual simulation tool using outline capturers.

For example, in the case of Dymola, the tool itself has an in-built outline capturer for the user's particular use.

EcosimPro does not have an in-built outline capturer. Instead, an external tool, SmartSketch, can be used to provide the user with the graphical possibilities which EcosimPro does not offer. This method is not as dynamic as that offered by Dymola.

2.7 PARTITIONS

The partition is a concept which is only implemented in EcosimPro and it offers a great deal of flexibility in the design of experiments and simulations. It is a feature which is in-built into the tool and not into the language. Table 7 illustrates the possibilities that partitions offer.

None of the possibilities which EcosimPro offers with its partitions are available with Dymola/ Modelica, at least as far as can be seen from the information supplied with the product [3].

Table 7: Possibilities based on Partitions

Feature	**Modelica**	**EL**
Choice of surrounding conditions	NO	YES
Choice of variable tearing in algebraic loops	NO	YES
Choice of high index problem states	NO	YES
Change parameter variability	NO	YES

2.8 DEFINITION OF EXPERIMENTS

EL explicitly offers the possibility of defining experiments for the systems to be simulated using blocks of sequential code with which the user can: declare variables, initialise algebraic variables in the models, define surrounding conditions, calculate steady states, select integration methods, select tolerances, generate reports, invoke external functions, etc.

With Modelica, all these possibilities depend greatly on the tool with which it is working. The functions that EL offers through the experiments can be attained through the scripts of Modelica, but to a limited extent.

3 EXAMPLE OF MODELLING AND SIMULATION IN BOTH LANGUAGES

3.1 INTRODUCTION

In this section we are going to describe a particularly interesting example when it comes to practical comparisons between the two languages. The proposed system has been taken from [7] which suggests studying the model of a simple control system in which one of the components is gradually modified to add more real behaviour, and at the same time observing how overall system behaviour changes.

The proposed system is that of Figure 7.7 of [7] (page 107), which the authors have titled “Our Sensor Benchmark” [Nota de traductor: Banco de Pruebas = Test Bench; Benchmark = marca de referencia, benchmark test = prueba de referencia]. It is a system which, though small, incorporates continuous and discrete components. By defining polymorphic components and parametric classes, it is possible to describe different component models and, in parallel, study the variations in the simulation results.

3.2 DESCRIPTION OF THE SYSTEM TO BE SIMULATED

The system to be simulated is an angular velocity control system (Figure 1). It comprises a PI controller, an ideal actuator and the plant to be controlled. The plant is formed by an inertial load connected to a fixed casing by a friction. The flange velocity is the variable which is to be controlled. This velocity is obtained through a sensor whose behaviour will affect overall system behaviour.

The aim is to study the results of the different simulations in which different sensor models have been used. To this end, three different models of the sensor are built: ideal, with sampler and zero order retainer; and with sampler-retainer and A/D converter.

Of all the system components, the only one that will vary is ‘sensor’, whose behaviour will successively be refined adding more realism.

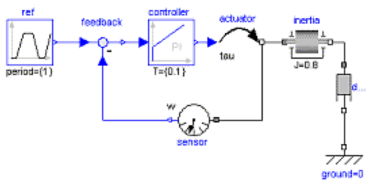

Figure 1: Benchmark

3.3 IMPLEMENTATION IN MODELICA

To implement the system in Modelica all the components used will be those contained in the Modelica Standard Library (MSL). The different implementations of the velocity sensor are based on components from the MSL, although the models vary as characteristics are introduced.

The following is the Modelica code of the system shown in Figure 1:

```
model SensorBenchmark "Banco de pruebas"
  import Modelica.Mechanics.Rotational;
  import Modelica.Blocks;
  Blocks.Continuous.PI controller(k={100},T={0.1});
  Blocks.Math.Feedback;
  Blocks.Sources.Trapezoid ref(offset={50}, rising={0.2},
width={0.25}, falling={0.2}, amplitude={50});
  Rotational.Inertia inertia(J=0.8);
  Rotational.Damper damper(d=2);
  Rotational.Fixed ground;
  Rotational.Torque actuator ;
  replaceable Rotational.Sensors.SpeedSensor sensor
extends Rotational.Interfaces.AbsoluteSensor;
equation
  connect(ref.outPort, feedback.inPort1)
  connect(feedback.outPort, controller.inPort)
  connect(controller.outPort, actuator.inPort)
  connect(actuator.flange_b, inertia.flange_a);
  connect(inertia.flange_b, damper.flange_a);
  connect(damper.flange_b, ground.flange_b);
  connect(actuator.flange_b, sensor.flange_a);
  connect(sensor.outPort, feedback.inPort2);
end SensorBenchmark;
```

Attention is drawn to the declaration of the parametrisable component ‘sensor’, forerunner of the specifier ‘replaceable’. To define different control system classes with different sensor models it would be necessary to create them as classes derived from ‘SensorBenchmark’, redeclaring the component ‘sensor’. The classes to which the sensors can belong must always be subclasses of ‘Modelica.Mechanics. Rotational.Interfaces.AbsoluteSensor’.

New models are defined below for the different sensors to be implemented:

3.3.1 Ideal Sensor

This sensor measures the absolute flange angular velocity without introducing any type of error into the measurement.

The following is the Modelica code of this component:

```
model SpeedSensor  "Ideal sensor to measure the absolute
flange angular velocity"
   extends Interfaces.Absolute.Sensor;
   SIunits.AngularVelocity w "Absolute angular velocity of
flange";
equation
   w = der(flange_a.phi);
   w = outPort.signal[1];
   0 = flange_a.tau;
end SpeedSensor;
```

This component belongs to the MSL and has been reproduced to that we can briefly comment on it. It is a model derived from the base class ‘Modelica. Mechanics.Rotational.Interfaces.AbsoluteSensor’.

Add a continuous variable of the angular velocity type (subtype of real) 'w' and it will be defined as a derivative of the angular position 'phi' contained in the connector of the base class (flange_a). The base class appears in the fragment of code shown below:

```
partial model AbsoluteSensor   "Base class to measure a
single absolute flange variable"
  extends Modelica.Icons.RotationalSensor   "(left) flange to
be measured (flange axis directed INTO cut plane)";
   Modelica.Blocks.Interfaces.OutPort outPort(final n=1);
end AbsoluteSensor;
```

This class comprises two connectors, one is mechanical (Flange_a) and the other is a type of output signal (OutPort).

3.3.2 Sensor with Sampler and Retainer

The code is shown below:

```
model SampleHoldSensor
import Modelica.Mechanics.Rotational;

   extends Rotational.Interfaces.AbsoluteSensor;
   Modelica.SIunits.AngularVelocity w ;
   parameter Modelica.SIunits.Time sample_interval=0.1;
equation
   w = der(flange_a.phi);
   flange_a.tau = 0;
algorithm
   when sample(0, sample_interval) then
      outPort.signal[1] := w ;
   end when;
end SampleHoldSensor;
```

This sensor is derived from 'Modelica.Mechanics. Rotational.Interfaces.AbsoluteSensor' to fulfil the restriction imposed on the control system. It also adds the code necessary to model the behaviour of a sampler and zero order retainer. This code forces an event every 'sample_interval' seconds via the Modelica internal function 'sample()', and assigns the value of the angular velocity of the mechanical connectors to the output signal. Because the output signal is being assigned within a 'when' sentence, its variabiltiy becomes implicitly converted to discrete, maintaining its value until the next event which fulfils the condition in the 'when' sentence (its next sample). The sampling period is defined as a parameter of the model.

3.3.3 Sensor with Sampler, Retainer and A/D Converter

This sensor models the additional behaviour of a data acquisition system based on the A/D function. This adds more realism to the sensor and enables us to study the errors made by behaviours which have not been modelled in the previous models. In this case the quantification noise is added to the measurement from the sampler and retainer. The following is the code of the new model:

```
model QuantizedSensor
   import Modelica.SIunits;
   import Modelica.Mechanics.Rotational;
extends Rotational.Interfaces.AbsoluteSensor;

   parameter Integer bits=4;
   parameter SIunits.Time sample_interval=0.02;
   parameter SIunits.AngularVelocity min=-150;
   parameter SIunits.AngularVelocity max=150;
   SIunits.AngularVelocity w ;
protected
   parameter Real delta=(max - min)/2^bits;
   Integer level;
equation
   w = der(flange_a.phi);
   flange_a.tau = 0;
algorithm
   when sample(0, sample_interval) then
      level := integer((w - min)/delta);
   end when;
   if level < 0 then
      outPort.signal[1] := min;
   elseif level >= 2^bits then
      outPort.signal[1] := max;
   else
      outPort.signal[1] := level*delta + min;
   end if;
end QuantizedSensor;
```

This model is again derived from 'Modelica. Mechanics.Rotational.Interfaces.AbsoluteSensor' to fulfil the condition imposed by the system it represents??? sistema contenedor. It has a parameter, the number of bits from the A/D for the quantification, so that in different instantiations of the component with different values of the number of bits it is possible to observe the effect of this parameter on the final behaviour of the model.

With the code, we can easily follow the implementation of the operation of an A/D converter in which the extremes of the analogue signal to be converted (max and min) and the internal integer variable 'level' which stores the value of the conversion are defined. Based on this value, the 'outPort.signal[1]' continuous output variable is assigned as a function of the 'level' variable.

3.4 IMPLEMENTATION IN ECOSIMPRO/EL

There are two properties which are used by Modelica in the above code that EL does not have: replaceable components and the internal function 'sample()'. However this is not an obstacle, it is simply a difference in the design of the component. To implement the function of the three sensors, the sensor model has to be changed for each system simulation. In order that the sensor can sample, a component which implements the sample is created by the generation of discrete events. The system code, EL in Figure 1, is described below:

```
USE MECH_ROTATIONAL
USE TILLER_C04

COMPONENT SensorBenchmark
TOPOLOGY
   MECH_ROTATIONAL.RotationalInertia inertia (J=0.8)
   MECH_ROTATIONAL.RotationalFixed ground
   MECH_ROTATIONAL.RotationalDamper damper (d=0.2)
   MECH_ROTATIONAL.RotationalTorque actuator
   MECH_ROTATIONAL.RotationalSpeedSensor sensor
   TILLER_C04.PIController controller (Kp = 100, Ti = 0.1)
   CONNECT ground.p TO damper.flange_b
   CONNECT damper.flange_a TO inertia.flange_b
   CONNECT actuator.flange_b TO inertia.flange_a
   CONNECT sensor.flange_a TO inertia.flange_a
   CONNECT controller.driver TO actuator.inPort
   CONNECT sensor.outPort TO controller.sensor
END COMPONENT
```

3.4.1 Ideal Sensor

The following is the EL code obtained from the MECH_ROTATIONAL library developed by EA International:

```
ABSTRACT COMPONENT RotationalAbsoluteSensor
    PORTS
        OUT MECH_ROTATIONAL.Flange flange_a
        OUT CONTROL.analog_signal outPort
END COMPONENT

COMPONENT RotationalSpeedSensor IS_A
RotationalAbsoluteSensor
    DECLS
        REAL w
    CONTINUOUS
        w  = flange_a.phi'
        w  = outPort.signal
        0 = flange_a.tau
END COMPONENT
```

Following the methodology of implementation in Modelica, the component ‘RotationalSpeedSensor’ has been developed based on the abstract base class ‘RotationalAbsoluteSensor’.

3.4.2 Sensor with Sampler and Retainer

To implement this component in EL the behaviour of the sampler has to be modelled by the periodical generation of events. The ‘sampler’ component is implemented as the base class of other components which need to incorporate this behaviour, which could be achieved through the inheritance mechanism which the two languages have.

The following is the EL code of the ‘sampler’ component:

```
COMPONENT Sampler
DATA
   REAL sample_interval = 0.1
DECLS
   BOOLEAN sample = FALSE
INIT
  sample = TRUE AFTER sample_interval
DISCRETE
   WHEN(sample==TRUE)  THEN
      sample = FALSE AFTER 0
      sample = TRUE AFTER sample_interval
   END WHEN
END COMPONENT
```

The component to be implemented for replacement in the test bench / benchmark is ‘SampleHoldSensor’. It must be based on ‘sampler’ and it must include code to maintain the value of the continuous variable sampled until the next sampling instant.

The following is the EL code of the ‘SampleHoldSensor’ component:

```
COMPONENT SampleHoldSensor IS_A Sampler,
MECH_ROTATIONAL.RotationalAbsoluteSensor
DECLS
   REAL w
DISCRETE
   WHEN(sample==TRUE)  THEN
      outPort.signal = w
   END WHEN
CONTINUOUS
   w  = flange_a.phi'
   0 = flange_a.tau
END COMPONENT
```

By means of multiple inheritance the component incorporates the behaviour of the ‘Sampler’ which did not exist in the original ‘sensor’ component.

3.4.3 Sensor with Sampler, Retainer and A/D Converter

The component is the same as that which is coded in the Modelica section. The following is the EL code of this component:

```
USE MECH_ROTATIONAL

COMPONENT QuantizedSensor IS_A
MECH_ROTATIONAL.RotationalAbsoluteSensor, Sampler
DATA
   INTEGER bits  = 4
   REAL min = -150
   REAL max = 150
DECLS
   REAL delta
   REAL w
   INTEGER level
DISCRETE
   WHEN (sample) THEN
      level = (w  - min) / delta
      IF (level < 0) THEN
         outPort.signal = min
      ELSEIF (level >= 2**bits) THEN
         outPort.signal = max
      ELSE
         outPort.signal = level*delta + min
      END IF
   END WHEN
CONTINUOUS
   delta = (max - min) / 2**bits
   w  = flange_a.phi'
   flange_a.tau = 0
END COMPONENT
```

3.5 SIMULATION WITH DYMOLA/ MODELICA

To compare system behaviour with respect to the three sensor models, a simulation was run with the three components simultaneously. The following is the Modelica code of the system to be simulated:

```
model Comparacion

model SamplingCase = SensorBenchmark (redeclare
SampleHoldSensor sensor(sample_interval=0.01));
```

```
 model QuantizedCase=SensorBenchmark(redeclare
QuantizedSensor sensor(sample_interval=0.01, bits=8));

  SensorBenchmark ideal;
  SamplingCase sampling(sensor(sample_interval=0.015));
  QuantizedCase quantized(sensor(sample_interval=0.015));
end Comparacion;
```

To run the simulation of the three components simultaneously, a class which contains all three is defined. Use has been made beforehand of Modelica’s capability to define new ‘SamplingCase’ and ‘QuantizedSensor’ classes, both based on the initial model ‘SensorBenchmark’ but passing the sensor class to be used as an argument. ‘SamplingCase’ is a class derived from ‘SensorBenchmark’ in which the sensor class is a sampler and retainer. ‘QuantizedCase’ is a class derived from ‘SensorBenchmark’ in which the sensor class is a sampler, retainer and A/D function.

Subsequently, three components ‘ideal’, ‘sampling’ and ‘quantized’ of the ‘SensorBenchmark’, ‘SamplingCase’ and ‘QuantizedCase’ types, respectively, are declared. Parameters of instantiation in time are passed to the last two components to fix the sampling period at 15 ms, instead of the 10 ms defined by the classes to which they belong. The results of the simulation are shown in Figures 2 and 3.

It can be seen how the performance of the velocity control system becomes degraded as the sampler, retainer and A/D converter are added. At the beginning the behaviour is very oscillatory because the system is on standby (w=0), the reference starts with a value of 50, and the sampling period is excessively high. As the sampling period decreases and the number of bits from the A/D converter increases, the behaviour gets closer to that of the control system based on an ideal sensor.

3.6 SIMULATION WITH ECOSIMPRO/EL

The following is the EL code required for the simulation:

```
USE MECH_ROTATIONAL
USE TILLER_C04
COMPONENT SamplingCase
TOPOLOGY
  MECH_ROTATIONAL.RotationalInertia inertia (J=0.8)
  MECH_ROTATIONAL.RotationalFixed ground
  MECH_ROTATIONAL.RotationalDamper damper (d=0.2)
  MECH_ROTATIONAL.RotationalTorque actuator
  SampleHoldSensor sensor(sample_interval = 0.015)
  TILLER_C04.PIController controller (Kp = 100, Ti = 0.1)
  CONNECT ground.p TO damper.flange_b
  CONNECT damper.flange_a TO inertia.flange_b
  CONNECT actuator.flange_b TO inertia.flange_a
  CONNECT sensor.flange_a TO inertia.flange_a
  CONNECT controller.driver TO actuator.inPort
  CONNECT sensor.outPort TO controller.sensor
END COMPONENT

COMPONENT QuantizedCase
TOPOLOGY
  MECH_ROTATIONAL.RotationalInertia inertia (J=0.8)
  MECH_ROTATIONAL.RotationalFixed ground
  MECH_ROTATIONAL.RotationalDamper damper (d=0.2)
  MECH_ROTATIONAL.RotationalTorque actuator
  QuantizedSensor sensor(sample_interval = 0.015, bits=8)
  TILLER_C04.PIController controller (Kp = 100, Ti = 0.1)
  CONNECT ground.p TO damper.flange_b
  CONNECT damper.flange_a TO inertia.flange_b
  CONNECT actuator.flange_b TO inertia.flange_a
  CONNECT sensor.flange_a TO inertia.flange_a
  CONNECT controller.driver TO actuator.inPort
  CONNECT sensor.outPort TO controller.sensor
END COMPONENT

COMPONENT Comparacion
TOPOLOGY
          SensorBenchmark ideal
          SamplingCase sampling
          QuantizedCase quantized
END COMPONENT
```

The library TILLER_C04 has been developed by EA International. In this case, because there are no replaceable components in EL, the SamplingCase’ and ‘QuantizedCase’ classes have to be defined explicitly. The appropriate type of sensor is included in each one.

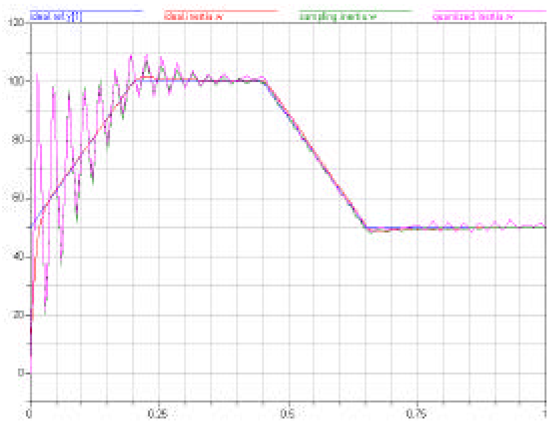

Figure 2: Simulation of the three Systems

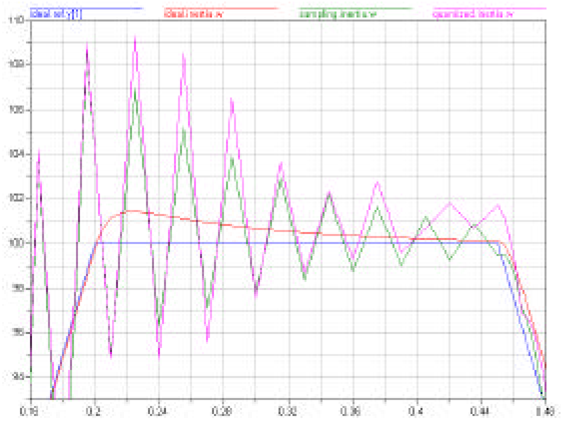

Figure 3: Detail of the Simulation

After the partition is generated by defect from EcosimPro, the following experiment is created:

```
EXPERIMENT exp1 ON Comparacion.default

DECLS
   TABLE_1D trapezoid = {{0., 0.2, 0.45, 0.65, 1},{0., 1., 1.,
0., 0.}}
INIT    -- set initial values for variables
     -- Dynamic variables
   ideal.inertia.phi = 0
   ideal.inertia.w  = 0
   ideal.controller.IntegratorBlock.s_out.signal = 0
   sampling.inertia.phi = 0
   sampling.inertia.w  = 0
   sampling.controller.IntegratorBlock.s_out.signal = 0
   quantized.inertia.phi = 0
   quantized.inertia.w  = 0
   quantized.controller.IntegratorBlock.s_out.signal = 0

BOUNDS    -- set expressions for boundary variables: v =
f(t,...)
   ideal.controller.command.signal = 50 + 50 *
periodTimeTableInterp(TIME, trapezoid, 1., 0)
   quantized.controller.command.signal = 50 + 50 *
periodTimeTableInterp(TIME, trapezoid, 1., 0)
   sampling.controller.command.signal = 50 + 50 *
periodTimeTableInterp(TIME, trapezoid, 1., 0)

BODY
   REPORT_TABLE("reportAll", " * ")
   TIME = 0
   TSTOP = 1
   CINT = 0.002
   INTEG()
END EXPERIMENT
```

This experiment aims to define the excitations to the system and initialise the system initially on standby. Figures 4 and 5 illustrate the simulation results obtained through EcoMonitor.

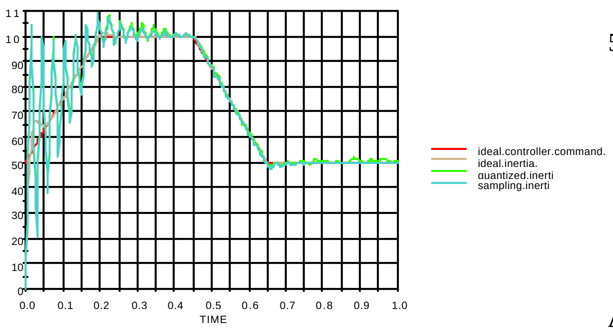

Figure 4: Simulation Results

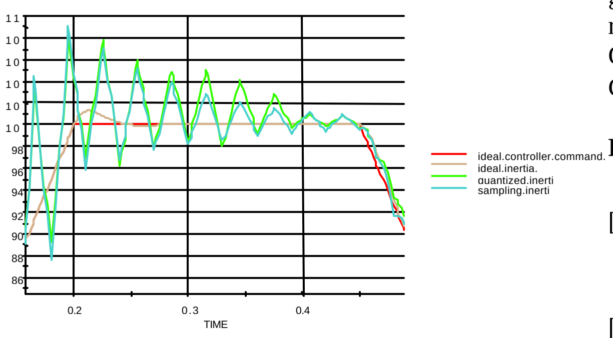

Figure 5: Simulation Details

We can see that the simulation results shown in Figures 4 and 5 are not the same as those in Figures 2 and 3. This is because a PI controller with different internal functions has been used in the implementation of the control system in each of the languages. In the case of Modelica the PI controller from the MSL has been used, and in the case of EL that described in chapter 4 of [7] has been used. In both cases we can clearly appreciate the effect of the sample and of the A/D converters used.

4 CONCLUSIONS

In general, both languages are perfectly tailored for the methodology of object-oriented modelling of continuous, discrete and hybrid systems.
The following are the authors' recommendations with respect to possible extensions to the two languages:

1 The modelling possibilities of EL would increase if the arrays could contain components in addition to simple types and derivatives.
2 The modelling possibilities of EL would increase with the inclusion of parametric classes.
3 Modelling components from the same inheritance graph with Modelica would be more flexible if it allowed the use of virtual equations which specialised the behaviour in successive generations of components.
4 The interface for simulating model experiments with EcosimPro by means of programming languages (C++) is a very flexible capability which Dymola/Modelica does not offer.
5 The use by EcosimPro/EL of complementary graphics tools such as SmartSketch, makes the task of modelling more uncomfortable than it would be if it were built into the tool itself. Because of this drawback, graphic modelling is much more comfortable with Dymola/Modelica than it is with EcosimPro.

Acknowledgements

The authors wish to thank the CICYT for the funds granted through FEDER to finance part of the research carried out within the scope of projects QUI99-0663-C02-02, DPI2000-1218-C04-01, DPI2001-2380-C02-02 and DPI2002-04375-C03-02/03.

References

[1] Modelica Association, (2002): "Modelica Language Specification, Version 2.0". http://www.modelica.org

[2] EA International. EcosimPro 3.2. Documentation

[3] Dynasim AB. Dymola 5.0a. Documentation

[4] Andersson, M. (1994). "Object-Oriented Modeling and Simulation of Hybrid Systems". PhD thesis. ISRN LUTFD2/TFRT—1043—SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden

[5] Modelica Association, (2002): "Modelica Tutorial, Version 1.4".http://www.modelica.org

[6] Stroustrup, B., (1997): "The C++ Programming Language. Third Edition". Addison-Wesley

[7] Tiller, M.M. (2001): "Introduction to Physical Modeling with Modelica". Kluwer Academic Publishers

[8] Urquía Moraleda, A. (2000). "Modelado Orientado a Objetos y Simulación de Sistemas Híbridos en el ámbito del Control de Procesos Químicos" (*Object-oriented Modelling and Simulation of Hybrid Systems in the Field of Chemical Process Control*). PhD thesis. Department of Information Technology and Automatic Control, Faculty of Science, UNED